Alphabill Yellowpaper

Ahto Buldas, Märt Saarepera, Risto Laanoja, Ahto Truu

Specification

Version TestNet V1 (main/af6934ab)

February 28, 2024

Contents

Notation

$\mathbb{A}, \mathbb{B}, \ldots$	types
$a \in \mathbb{A}$	variable or constant a is of type $\mathbb{A}$
$\mathbb{N}_k$	k-bit unsigned integers $(0 \ldots 2^k - 1)$
$\mathbb{N}_k^{\pm}$	k-bit signed integers $(-2^{k-1} \ldots 2^{k-1} - 1)$
$\mathbb{A}[\mathbb{B}]$	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 $\mathbb{A}$
$\{0,1\}^*$	the set of all finite length bitstrings, including the empty string denoted by $\lfloor\rfloor$
$\{0,1\}^k$	the set of all bitstrings of length k
$\{0,1\}^{\leq k}$	the set of all bitstrings of length $\leq k$, including $\lfloor\rfloor$
$\{0,1\}^{\geq k}$	the set of all bitstrings of length $\geq k$
$\mathbb{A}^*$	the class of all finite arrays of elements of type $\mathbb{A}$, including the empty array
$\mathbb{A}^k$	the class of all arrays of length k of 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; $a \in \mathbb{A}^{\lvert a \rvert}$
$a \lVert b$	concatenation of lists or bitstrings a and b
$\wedge$	logical AND operation
$\vee$	logical OR operation

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 identifier ι, owner condition φ, and unit data D
- **transactions** that create new units, delete units, or change the data of the units

Alphabill Framework:

- defines *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* α to transaction systems
- provides *unicity certificate service* for the registered transaction systems: unique state root hash h_α and transaction root hash $h_{B\alpha}$ and summary value V_α for every pair (n, α), where n is the round number of transaction system.

1.2 Alphabill Architecture

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

Partitions are 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.

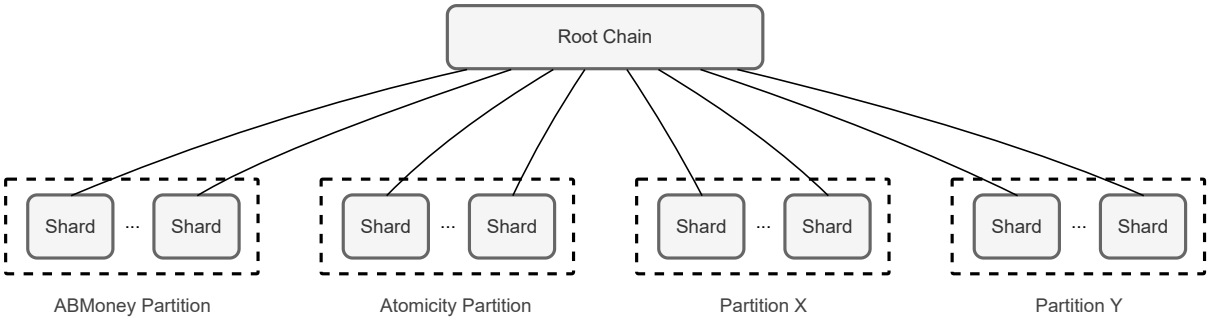

Figure 1. Alphabill functional model

2 Framework Data Structures and Functions

2.1 Parameters, Types, Constants, Functions

2.1.1 Parameters

sidlen $= 32$ – system identifier length (global)
tidlen – unit type identifier length of type $\mathbb{N}$ (per transaction system)
uidlen – unit identifier length of type $\mathbb{N}$ (per transaction system)

2.1.2 Types

$\mathbb{A} = \mathbb{N}_{\text{sidlen}}$ – system identifiers
$\mathbb{IT} = \{0,1\}^{\text{tidlen}}$ – unit type identifiers
$\mathbb{IU} = \{0,1\}^{\text{uidlen}}$ – unit identifiers
$\mathbb{I} = \mathbb{IU} \times \mathbb{IT} = \{0,1\}^{\text{uidlen+tidlen}}$ – extended identifiers, combining the type and the unit identifiers
$\mathbb{L}$ – owner condition type (represents byte-codes of ownership conditions)
$\mathbb{H}$ – hash value type
$\mathbb{B}$ – unicity trust base type
$\mathbb{SP}$ – unit (state) proof type
$\mathbb{XP}$ – transaction (execution) proof type
$\mathbb{T}$ – transaction type identifiers
$\mathbb{TO}$ – abstract transaction order type, including union of transaction attributes over all valid transaction types, client-side metadata, and authorization proofs
$\mathbb{TR}$ – abstract transaction record type, including union of transaction attributes over all valid transaction types, client-side metadata, authorization proofs, and server-side metadata
$\mathbb{MC}$ – client-side transaction metadata type
$\mathbb{MS}$ – server-side transaction metadata type
$\mathbb{SD}$ – system description type
$\mathbb{U}$ – 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 $\mathbb{H}$

2.1.4 Functions

H – hash function of type $\{0,1\}^* \to \mathbb{H}$
VerifyUnicitySeal – unicity seal verification function of type $\mathbb{H} \times \mathbb{U} \times \mathbb{B} \to \{0,1\}$
VerifyOwner – ownership verification function of type $\mathbb{L} \times \mathbb{TO} \times \{0,1\}^* \to \{0,1\}$, where the last argument is the input to satisfy the owner condition (a common example is a digital signature to be verified with a public key embedded in the owner condition)
VerifyUnitProof – unit proof verification function of type $\mathbb{SP} \times \mathbb{B} \times \mathbb{SD} \to \{0,1\}$
VerifyTxProof – transaction proof verification function of type $\mathbb{XP} \times \mathbb{TR} \times \mathbb{B} \times \mathbb{SD} \to \{0,1\}$

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
uidlen – the unit identifier length

The extended identifiers are the concatenation of the unit identifier part and the type identifier part, with the unit identifier in the uidlen most significant bits and the type identifier in the tidlen least significant bits 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 System Description Record

Every transaction system registered in the Alphabill Framework with system identifier α is described by a data structure SD[α] of type $\mathbb{SD}$ that is a tuple $(\text{tidlen}, \text{uidlen}, \mathcal{U}, \mathbb{D}, \mathbb{V}, V_0, F_S, V_S, \gamma, \mathcal{V}, F_C, \iota_{FC})$, where:

1. tidlen – unit type identifier length
2. uidlen – unit identifier length
3. $\mathcal{U}$ – list of known unit type identifiers
4. $\mathbb{D}$ – abstract unit data type (union of unit data types for all known unit types)
5. $\mathbb{V}$ – summary value type
6. V_0 – summary value of the data related to the unit with zero-identifier $0_{\mathbb{I}}$
7. F_S – node summary function of type $(\mathbb{V} \cup \{\bot\}) \times \mathbb{V} \times \mathbb{V} \to \mathbb{V}$
8. V_S – data summary function of type $\mathbb{D} \to \mathbb{V}$
9. γ – summary check predicate of type $\mathbb{V} \times \mathbb{V} \to \{0,1\}$
10. $\mathcal{V}$ – summary trust base of type $\mathbb{V}$
11. F_C – transaction cost function of type $\mathbb{TO} \times \mathbb{S} \to \mathbb{N}_{64}$
12. ι_{FC} – identifier of the bill in the money partition that represents the fee credits users have in this transaction system not yet paid out to the validators

Table 1. System description record.

No	Field	Notation	Type
1.	Type identifier length	tidlen	$\mathbb{N}$
2.	Unit identifier length	uidlen	$\mathbb{N}$
3.	Valid unit types	$\mathcal{U}$	$(\{0,1\}^{\text{tidlen}})^*$
4.	Unit data type	$\mathbb{D}$	
5.	Summary value type	$\mathbb{V}$	
6.	Summary value of the data related to the unit with zero-identifier $0_{\mathbb{I}}$	V_0	$\mathbb{V}$
7.	Node summary function	F_S	$(\mathbb{V} \cup \{\bot\}) \times \mathbb{V} \times \mathbb{V} \to \mathbb{V}$
8.	Data summary function	V_S	$\mathbb{D} \to \mathbb{V}$
9.	Summary check predicate	γ	$\mathbb{V} \times \mathbb{V} \to \{0,1\}$
10.	Summary trust base	$\mathcal{V}$	$\mathbb{V}$
11.	Transaction cost function	F_C	$\mathbb{TO} \times \mathbb{S} \to \mathbb{N}_{64}$
12.	Fee credit bill identifier	ι_{FC}	$\mathbb{I}$

2.4 Transaction Orders and Records

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

1. α – system identifier of type $\mathbb{A}$
2. τ – transaction type identifier of type $\mathbb{T}$
3. ι – unit identifier of type $\mathbb{I}$
4. A – transaction attributes of type $\mathbb{AT}_\tau$
5. M_C – client metadata for the transaction, of type $\mathbb{MC}$, consisting of
   - 5.1 T_0 – timeout of type $\mathbb{N}_{64}$
   - 5.2 f_m – maximum fee the user is willing to pay for the execution of this transaction, of type $\mathbb{N}_{64}$
   - 5.3 ι_f – optional identifier of the fee credit record of type $\mathbb{I}$
6. s – owner proof of type $\{0,1\}^*$
7. s_f – optional fee authorization proof of type $\{0,1\}^*$ (omitted when the main owner proof s also satisfies the fee owner condition)

Each transaction order P has an associated *set of target units* targets(P). In most cases targets(P) = $\{P.\iota\}$, 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 $P' = \langle P, M_S\rangle$, with P a transaction order as defined above and $M_S = (f_a, r, R)$, where:

1. M_S – service metadata for the transaction, of type $\mathbb{MS}$, consisting of
   - 1.1 f_a – actual fee charged for the processing of this transaction
   - 1.2 r – indicates whether the transaction was executed successfully; currently only successful transactions (with $M_S.r = 1$) are recorded in blocks; however, in the

future also unsuccessful transactions (with $M_S.r = 0$) may be recorded and charged for

1.3 R – optional processing result details, of type $\mathbb{RT}_\tau$

Table 2. Fields of a transaction order.

No	Field	Notation	Type
1.	System identifier	α	$\mathbb{A}$
2.	Transaction type identifier	τ	$\mathbb{T}$
3.	Unit identifier	ι	$\mathbb{I}$
4.	Transaction attributes	A	$\mathbb{AT}_\tau$
5.	Timeout	$M_C.T_0$	$\mathbb{N}_{64}$
6.	Maximum transaction fee	$M_C.f_m$	$\mathbb{N}_{64}$
7.	Fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\perp\}$
8.	Owner proof	s	$\{0,1\}^*$
9.	Fee authorization proof	s_f	$\{0,1\}^* \cup \{\perp\}$

Table 3. Fields of a transaction record.

No	Field	Notation	Type
1.	System identifier	α	$\mathbb{A}$
2.	Transaction type identifier	τ	$\mathbb{T}$
3.	Unit identifier	ι	$\mathbb{I}$
4.	Transaction attributes	A	$\mathbb{AT}_\tau$
5.	Timeout	$M_C.T_0$	$\mathbb{N}_{64}$
6.	Maximum transaction fee	$M_C.f_m$	$\mathbb{N}_{64}$
7.	Fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\perp\}$
8.	Owner proof	s	$\{0,1\}^*$
9.	Fee authorization proof	s_f	$\{0,1\}^* \cup \{\perp\}$
10.	Actual transaction fee	$M_S.f_a$	$\mathbb{N}_{64}$
11.	Success indicator	$M_S.r$	$\{0,1\}$
12.	Processing details	$M_S.R$	$\mathbb{RT}_\tau$

2.5 Unicity Tree and Unicity Certificate

2.5.1 Unicity Tree Certificate

Unicity Tree Certificate is a tuple $C^{\text{uni}} = (\alpha, \text{dhash}; h^s_1, \ldots, h^s_{\text{sidlen}})$, where:

1. α – system identifier of type $\mathbb{A}$
2. dhash – system description hash of type $\mathbb{H}$
3. $h^s_1, \ldots, h^s_{\text{sidlen}}$ – sibling hashes of type $\mathbb{H}$

2.5.2 Unicity Seal

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

1. n_r – Root Chain Round number
2. e_r – Root Chain Epoch number
3. t_r – Round creation time (wall clock value specified and verified by the Root Chain), with one-second precision. See Appendix B for encoding
4. r_- – Root hash of previous round's Unicity Tree
5. r – Root hash of Unicity Tree (denoted as r^{root} if necessary for clarity)
6. 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.

2.5.3 Unicity Certificate

Unicity Certificate is a tuple $UC = (IR, C^{\text{uni}}, C^{\text{r}})$. In some contexts it may include a shard certificate: $(C^{\text{shard}}, IR, C^{\text{uni}}, C^{\text{r}})$. Elements of the tuple form an authenticated chain, e.g., the verification of fields in IR looks like:

- $r \leftarrow \text{CompUnicityTreeCert}(C^{\text{uni}}, IR)$
- $1 = \text{VerifyUnicitySeal}(r, C^{\text{r}}, \mathcal{T})$.

2.6 State Tree and Unit State Proof

2.6.1 State Tree Certificate

State Tree Certificate C^{state} consists of the following components:

1. An initial tuple $(h_L, V_L; h_R, V_R)$
2. A list of tuples $(\iota_1, z_1, V_1; h_1^s, V_1^s), \ldots, (\iota_m, z_m, V_m; h_m^s, V_m^s)$

See Sec. 4.11 for more details.

2.6.2 Unit Tree Certificate

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

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

See Sec. 4.10 for more details.

2.6.3 Unit State Proof

Unit State 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 $\mathbb{H} \cup \{\bot\}$

3. C^{unit} – unit tree certificate
4. V_0 – data summary of type SD.$\mathbb{V}$
5. C^{state} – state tree certificate
6. UC – unicity certificate

Creation and verification of unit state proofs is specified in Sec. 2.8.4 and 2.8.5.

2.7 Transaction Execution Proof

Transaction Execution Proof for a transaction record P is a tuple $\Pi^{\text{tx}} = \langle h_h, C, UC \rangle$, where:

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

Transaction proof creation and verification is specified in Sec. 2.8.6 and 2.8.7.

2.8 Functions

2.8.1 Compute Unicity Tree Certificate: CompUnicityTreeCert

Input:

1. $(\alpha, \text{dhash}; h^s_1, \ldots, h^s_{\text{sidlen}})$ – unicity tree certificate
2. IR – Input Record

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

Computation:

$r \leftarrow H(IR, \text{dhash})$
for $i \leftarrow$ sidlen downto 1 **do**
  if $\alpha_i = 0$ **then** $r \leftarrow H(r, h^s_i)$
  if $\alpha_i = 1$ **then** $r \leftarrow H(h^s_i, r)$
end for
return r

2.8.2 Compute State Tree Certificate: 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 SD.$\mathbb{V}$
4. $C^{\text{state}} = \langle (h_L, V_L; h_R, V_R); (\iota_1, z_1, V_1; h^s_1, V^s_1), \ldots, (\iota_m, z_m, V_m; h^s_m, V^s_m) \rangle$ – state tree certificate of type $\mathbb{CS}$
5. SD – system description of type $\mathbb{SD}$

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 SD.$\mathbb{V}$

Computation:

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

2.8.3 Compute Unit Tree Certificate: CompUnitTreeCert

Input:

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

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

Computation:

$z \leftarrow H(H_0(x_-, t), s)$ ▹ H_0 defined in Sec. 4.9
return PLAIN_TREE_OUTPUT$(\langle (b_1, y_1), \ldots, (b_m, y_m) \rangle; z)$ ▹ Sec. C.1.3

2.8.4 Create Unit Proof: 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. SD – system 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
$x_- \leftarrow \bot$ ▹ Unit was created, no previous state
end if
$C^{\text{unit}} \leftarrow$ CreateUnitTreeCert(ι, i, N) ▹ Sec. 4.10.2
$C^{\text{state}} \leftarrow$ CreateStateTreeCert$(\iota, N, \iota_r, \text{SD})$ ▹ Sec. 4.11.2
return $(\iota, x_-, C^{\text{unit}}, N[\iota].V, C^{\text{state}}, UC)$

2.8.5 Verify Unit Proof: VerifyUnitProof

Input:

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

Output: True or False

Computation:

$z \leftarrow$ CompUnitTreeCert(x_-, C^{unit})
$(h, v) \leftarrow$ CompStateTreeCert$(\iota, z, V_0; C^{\text{state}}, \text{SD})$
$r \leftarrow$ CompUnicityTreeCert$(UC.C^{\text{uni}}, UC.IR)$
return $UC.C^{\text{uni}}$.dhash $= H(\text{SD}) \ \wedge$ VerifyUnicitySeal$(r, UC.C^{\text{r}}, \mathcal{T})$
$\wedge \ \gamma(V, \text{SD}.\mathcal{V}) \ \wedge \ UC.IR.h = h \ \wedge \ UC.IR.v = v$

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 bearer predicate φ and data D:

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

2. At some point during the round n, the unit ι had the transaction in the record P' applied to it:

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

3. At some point during the round n, the unit ι had the transaction in the record P' applied to it and this set the unit's bearer predicate to φ and data to D (essentially the conjunction of the two previous conditions):

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

4. The state of the unit ι did not change from the beginning of round n_1 to the end of round n_2:

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

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

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

2.8.6 Create Transaction Proof: CreateTxProof

Input:

1. $B = \langle(\alpha, \sigma, h_{\_}, \nu); P_1, \ldots, P_k; UC\rangle$ – block
2. i – index of the transaction to generate proof for

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

Computation:

assert $1 \leq i \leq k$
$h_h \leftarrow H(\alpha, \sigma, h_{\_}, \nu)$
$C \leftarrow$ PLAIN_TREE_CHAIN$(\langle H(P_1), \ldots, H(P_k)\rangle, i)$ ▹ Sec. C.1.2
return (h_h, C, UC)

2.8.7 Verify Transaction Proof: VerifyTxProof

Input:

1. $\Pi^{\mathsf{tx}} = (h_h, C, UC)$ – transaction proof
2. P – transaction record
3. $\mathcal{T}$ – trust base
4. SD – system description of type $\mathbb{SD}$

Output: TRUE or FALSE

Computation:

$h \leftarrow$ PLAIN_TREE_OUTPUT$(C, H(P))$ ▹ Sec. C.1.3

$h \leftarrow H(h_h, h)$
$r \leftarrow$ CompUnicityTreeCert$(UC.C^{\text{uni}}, UC.IR)$
return $UC.C^{\text{uni}}$.dhash $= H(\text{SD}) \ \wedge$ VerifyUnicitySeal$(r, UC.C^{\text{r}}, \mathcal{T})$
$\wedge \ UC.IR.h_B = h$

3 Root Partition

3.1 State of the Root Partition

State of the root partition is a tuple $(n, e, \mathcal{T}, \mathcal{A}, \mathcal{H}, \mathcal{SD})$, where:

1. n – round number of type $\mathbb{N}_{64}$
2. e – epoch number of type $\mathbb{N}_{64}$
3. $\mathcal{T}$ – unicity trust base of type $\mathbb{B}$
4. $\mathcal{A}$ – set of system identifiers, with elements of type $\mathbb{A}$
5. $\mathcal{N}$ – round numbers n_α of the registered transaction systems (type: $\mathcal{A} \to \mathbb{N}_{64}$)
6. $\mathcal{H}$ – last-certified root hashes h_α of the registered transaction systems (type: $\mathcal{A} \to \mathbb{H}$)
7. $\mathcal{SD}$ – system descriptions of type $\mathbb{SD}[\mathcal{A}]$ for all registered transaction systems

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

Table 4. State of the root partition.

No	Field	Notation	Type
1.	Round number of the root partition	n	$\mathbb{N}_{64}$
2.	Epoch number of the root partition	e	$\mathbb{N}_{64}$
3.	Unicity trust base	$\mathcal{T}$	$\mathbb{B}$
4.	Registered system identifiers	$\mathcal{A}$	Subset of $\mathbb{A}$
5.	Round numbers	$\mathcal{N}$	$\mathcal{A} \to \mathbb{N}_{64}$
6.	Last-certified root hashes	$\mathcal{H}$	$\mathcal{A} \to \mathbb{H}$
7.	System descriptions	$\mathcal{SD}$	$\mathbb{SD}[\mathcal{A}]$

3.2 System Input Record

System input record (IR) of a transaction system is a tuple $(n, e, h', h, v, h_B, f_B)$, where:

1. n – transaction system's round number of type $\mathbb{N}_{64}$
2. e – transaction system'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 $\mathbb{V}^* = \cup_{\alpha\in\mathcal{A}}\mathcal{SD}[\alpha].\mathbb{V}$
6. h_B – hash of the block B computed over all fields except certificates, type $\mathbb{H}$; computation is specified by the function BLOCK_HASH()
7. f_B – sum of the actual fees over all transaction records in the block, of type $\mathbb{N}_{64}$

The field descriptions above apply to a committed (certified) Input Record. In the state certification *request*, the fields are interpreted as follows: h' - root hash to be extended (that is, of the last certified round), h, v, h_B - values proposed for certification and n is incremented for the proposed round.

3.3 Unicity Tree and Unicity Tree Certificate

Let $\mathcal{A} \subseteq \mathbb{A}$ be the set of system identifiers of the registered transaction systems, i.e. $\mathcal{A} = \{\alpha \in \mathbb{A}\colon\ \mathcal{SD}[\alpha] \neq \perp\}$.

Let $\mathcal{IR}$ be system input records of type $(\mathbb{N}_{64}\times\mathbb{N}_{64}\times\mathbb{H}\times\mathbb{H}\times\mathbb{V}^*\times\mathbb{H}\times\mathbb{N}_{64})[\mathcal{A}]$, i.e. for every $\alpha\in\mathcal{A}$, either $\mathcal{IR}[\alpha] = (n_\alpha, e_\alpha, h'_\alpha, h_\alpha, v_\alpha, h_{B_\alpha}, f_{B_\alpha})$, or $\mathcal{IR}[\alpha] = \perp$.

Let $\overline{\mathcal{A}}$ be the closure of $\mathcal{A}$ as a prefix-free code (Appendix A). This means that, by definition, $\overline{\mathcal{A}} = \{\alpha \in \{0,1\}^*\colon\ \exists c \in \{0,1\}^*\colon\ \alpha c \in \mathcal{A}\}$.

Unicity tree is a sparse Merkle tree, i.e. a function of type $\overline{\mathcal{A}} \to \mathbb{H}$ such that:

- If $\alpha \in \mathcal{A}$, then $\chi(\alpha) = H(\mathcal{IR}[\alpha]||H(\mathcal{SD}[\alpha]))$, where we assume that $\perp||H(\mathcal{SD}[\alpha]) = H(\mathcal{SD}[\alpha])$;
- if $\alpha \in \overline{\mathcal{A}}\backslash\mathcal{A}$ and $\alpha 0 \notin \overline{\mathcal{A}}$, then $\chi(\alpha) = H(0_\mathbb{H}||\chi(\alpha 1))$,
- if $\alpha \in \overline{\mathcal{A}}\backslash\mathcal{A}$ and $\alpha 1 \notin \overline{\mathcal{A}}$, then $\chi(\alpha) = H(\chi(\alpha 0)||0_\mathbb{H})$,
- for any other $\alpha \in \overline{\mathcal{A}}$: $\chi(\alpha) = H(\chi(\alpha 0)||\chi(\alpha 1))$.

3.4 Functions of the Root Partition

3.4.1 Create Unicity Tree: CreateUnicityTree

Input:

1. $\mathcal{A}$ – set of system identifiers of type $\mathbb{A}$
2. $\mathcal{SD}$ – system description of type $\mathbb{SD}[\mathcal{A}]$
3. $\mathcal{IR}$ – system input records of type $(\mathbb{N}_{64}\times\mathbb{N}_{64}\times\mathbb{H}\times\mathbb{H}\times\mathbb{V}^*\times\mathbb{H}\times\mathbb{N}_{64})[\mathcal{A}]$, where $h'_\alpha = \mathcal{H}[\alpha]$, whenever $\mathcal{IR} = (n_\alpha, e_\alpha, h'_\alpha, h_\alpha, v_\alpha, h_{B_\alpha}, f_{B_\alpha}) \neq \perp$

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

Computation: gentree(⌊⌋)

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

gentree(α):

1. **if** $\alpha \in \mathcal{A}$, **then store** $\chi(\alpha) \leftarrow H(\mathcal{IR}[\alpha]||H(\mathcal{SD}[\alpha]))$ and **return** $\chi(\alpha)$
2. **else if** $\alpha \in \overline{\mathcal{A}}\backslash\mathcal{A}$ and $\alpha 0 \notin \overline{\mathcal{A}}$, **then store** $\chi(\alpha) \leftarrow H(0_\mathbb{H}||\text{gentree}(\alpha 1))$ and **return** $\chi(\alpha)$

3. **else if** $\alpha \in \overline{\mathcal{A}} \backslash \mathcal{A}$ and $\alpha 1 \notin \overline{\mathcal{A}}$, **then store** $\chi(\alpha) \leftarrow H(\text{gentree}(\alpha 0) || 0_{\mathbb{H}})$ and **return** $\chi(\alpha)$
4. **else store** $\chi(\alpha) \leftarrow H(\text{gentree}(\alpha 0) || \text{gentree}(\alpha 1))$ and **return** $\chi(\alpha)$

3.4.2 Create Unicity Tree Certificate: CreateUnicityTreeCert

Input:

1. α – system identifier of type $\mathbb{A}$
2. χ – unicity tree of type $\overline{\mathcal{A}} \rightarrow \mathbb{H}$
3. $\mathcal{SD}$ – system description of type $\mathbb{SD}[\mathcal{A}]$

Output: unicity tree certificate $(\alpha, \text{dhash}; h^s_1, \ldots, h^s_{\text{sidlen}})$

Computation:

dhash $\leftarrow H(\mathcal{SD}[\alpha])$
for $i \leftarrow 1$ to sidlen **do**
  $h^s_i \leftarrow \chi(\alpha_1 \alpha_2 \ldots \alpha_{i-1} \overline{\alpha_i})$
end for
return $(\alpha, \text{dhash}; h^s_1, \ldots, h^s_{\text{sidlen}})$

where $\alpha_1 \alpha_2 \ldots \alpha_{\text{sidlen}}$ is the binary representation of α, and $\overline{\alpha_i}$ is the Boolean inverse of α_i

4 Generic Transaction System

4.1 Parameters

Every (shard of a) transaction system in the Alphabill network is completely defined by the following parameters:

1. α – system identifier of type $\mathbb{A}$
2. SD[α] = (tidlen, uidlen, $\mathcal{U}, \mathbb{D}, \mathbb{V}, F_S, V_S, \gamma, \mathcal{V}, F_C$) – system description of type $\mathbb{SD}$
3. $\mathcal{SH}$ – sharding scheme of type $\mathbb{SH}$
4. $\sigma \in \mathcal{SH}$ – shard identifier
5. PrndSh – function of type $\mathbb{IU} \times \{0,1\}^* \to \mathbb{IU}$ such that $f_{\mathcal{SH}}(\text{PrndSh}(\iota, X)) = f_{\mathcal{SH}}(\iota)$ for every ι and X
6. S_0 – initial state of type $\mathbb{S} = \mathbb{A} \times \{0,1\}^* \times \mathbb{SH} \times \mathbb{N}_{64} \times \mathbb{I} \times \mathbb{ND}[\mathbb{I}] \times \mathbb{B} \times \mathbb{SD}[\mathbb{A}]$, where $\mathbb{ND} = ((\mathbb{H} \cup \{\bot\}) \times \mathbb{H} \times \mathbb{L} \times \mathbb{D})^* \times \mathbb{H} \times \mathbb{L} \times \mathbb{D} \times \mathbb{V} \times \mathbb{H} \times \mathbb{I} \times \mathbb{I} \times \mathbb{N}_{32} \times \mathbb{N}_8^{\pm}$ is the node type
7. RInit – round initialization procedure of type $\mathbb{S} \to \mathbb{S}$
8. RCompl – round completion procedure of type $\mathbb{S} \to \mathbb{S}$
9. $\mathbb{T}$ – transaction identifier type (a finite set)
10. For every $\tau \in \mathbb{T}$:
    - 10.1 $\mathbb{AT}_\tau$ – attributes type
    - 10.2 $\mathbb{TO}_\tau = (\mathbb{A} \times \mathbb{T} \times \mathbb{I} \times \mathbb{AT}_\tau \times \mathbb{MC}) \times \{0,1\}^* \times (\{0,1\}^* \cup \{\bot\})$ – derived transaction order type
    - 10.3 $\mathbb{TR}_\tau = \mathbb{TO}_\tau \times \mathbb{MS}$ – derived transaction record type
    - 10.4 ψ_τ – predicate of type $\mathbb{TO}_\tau \times \mathbb{S} \to \{0,1\}$
    - 10.5 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(ι, X) = PrndSh(ι, X')
2. *Uniformity* – for any ι, and sufficiently large n, if $X \leftarrow \{0,1\}^n$ is uniformly distributed in $\{0,1\}^n$, 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

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

where $\sigma_1\sigma_2\ldots\sigma_\ell$ is the binary representation of $f_{\mathcal{SH}}(\iota)$ and $\chi_1\chi_2\ldots\chi_{\text{uidlen}}$ is the output of a collision-resistant hash function $H^*\colon \{0,1\}^* \to \{0,1\}^{\text{uidlen}}$. 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 ($H\colon \{0,1\}^* \to \{0,1\}^k$) 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 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 uidlen leftmost bits of the concatenation $H(X,0)\|H(X,1)\|\ldots$

4.2 Sharding Scheme

Sharding scheme $\mathcal{SH} : \mathbb{SH}$ 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.

4.3 Core

4.3.1 State of the Core

State of the Core of a transaction system is a tuple $(\alpha, \mathcal{SH}, n, \mathcal{SD})$, where:

1. α – system identifier of type $\mathbb{A}$
2. $\mathcal{SH}$ – sharding scheme of type $\mathbb{SH}$
3. n – round number of type $\mathbb{N}_{64}$
4. e – epoch number of type $\mathbb{N}_{64}$
5. $\mathcal{SD}$ – system descriptions of type $\mathbb{SD}[\mathbb{A}]$ for all registered transaction systems (including SD$[\alpha]$)

4.3.2 Shard Tree

Shard tree is a function $\chi\colon \overline{\mathcal{SH}} \to \mathbb{H} \times \mathbb{V}$ such that:

1. If $\sigma \in \mathcal{SH}$, then $\chi(\sigma) = (h_\sigma, V_\sigma)$, where (h_σ, V_σ) is the output of CompStateTreeCert in shard σ.
2. If $\sigma \in \overline{\mathcal{SH}} \backslash \mathcal{SH}$, then $\chi(\sigma) = (H(\sigma, 0_{\mathbb{H}}, \emptyset; h_0, V_0; h_1, V_1), F_S(\emptyset, V_0, V_1))$, where $(h_0, V_0) = \chi(\sigma\|0)$ and $(h_1, V_1) = \chi(\sigma\|1)$.

4.3.3 Shard Certificate

Shard certificate is a sequence $(s_1, h_1^s, V_1^s), \ldots, (s_m, h_m^s, V_m^s)$, where:

1. s_i – a bitstring of type $\{0,1\}^{m-i}$
2. h_i^s – sibling hash value of type $\mathbb{H}$
3. V_i^s – sibling summary value of type $\mathcal{SH}[\alpha].\mathbb{V}$

4.3.4 Shard Certificate Creation: CreateShardCert

Input:

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

Output: shard tree certificate $C^{\text{shard}} = ((s_1, h_1^s, V_1^s), \ldots, (s_m, h_m^s, V_m^s))$

Computation:

$m \leftarrow |\sigma|$
$C^{\text{shard}} \leftarrow ()$
for $i \leftarrow m$ downto 1 **do**
  $(h_i^s, V_i^s) \leftarrow \chi(\sigma_1\sigma_2 \ldots \sigma_{i-1}\overline{\sigma_i})$
  $C^{\text{shard}} \leftarrow C^{\text{shard}} || (\sigma_1\sigma_2 \ldots \sigma_{i-1}, h_i^s, V_i^s)$
end for
return C^{shard}

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.

4.4 Shard

4.4.1 State of a Shard

State of a (shard of a) transaction system is a tuple $(\alpha, \sigma, \mathcal{SH}, n, e, \iota_r, N, \mathcal{T}, \mathcal{SD})$, where:

1. α – system identifier of type $\mathbb{A}$
2. σ – shard identifier of type $\{0,1\}^{\leq \text{SD}[\alpha].\text{uidlen}}$
3. $\mathcal{SH}$ – sharding scheme of type $\mathbb{SH}$
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 $\mathbb{B}$
9. $\mathcal{SD}$ – system descriptions of type $\mathbb{SD}[\mathbb{A}]$ for all registered transaction systems (including SD[α])

4.4.2 Node of the State Tree

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

Table 5. State of a transaction system.

No	Field	Notation	Type
1.	System identifier	α	$\mathbb{A}$
2.	Shard identifier	σ	$\{0,1\}^{\leq \mathrm{SD}[\alpha].\mathrm{uidlen}}$
3.	Sharding scheme	$\mathcal{SH}$	$\mathbb{SH}$
4.	Round number	n	$\mathbb{N}_{64}$
5.	Epoch number	e	$\mathbb{N}_{64}$
6.	Root node identifier	ι_r	$\mathbb{I}$
7.	State tree	N	$\mathbb{ND}[\mathbb{I}]$
8.	Unicity trust base	$\mathcal{T}$	$\mathbb{B}$
9.	System descriptions	$\mathcal{SD}$	$\mathbb{SD}[\mathcal{A}]$

1.2 x_i – the new head hash of the unit ledger, of type $\mathbb{H}$

1.3 φ_i – the new bearer condition of type $\mathbb{L}$

1.4 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. φ – current bearer condition of type $\mathbb{L}$
4. D – current unit data of type $\mathbb{D}$
5. V – summary value of the sub-tree rooted at this node, of type $\mathbb{V}$
6. h – summary hash of the sub-tree rooted at this node, of type $\mathbb{H}$
7. ι_L – left child node identifier of type $\mathbb{I}$
8. ι_R – right child node identifier of type $\mathbb{I}$
9. d – depth of the subtree of type $\mathbb{N}_{32}$
10. b – balance factor of type $\mathbb{N}_8^{\pm}$

Table 6. Node of the State Tree.

No	Field	Notation	Type
1.	Transaction record hash	$t_i = S_i.t$	$\mathbb{H} \cup \{\bot\}$
2.	New unit ledger hash	$x_i = S_i.x$	$\mathbb{H}$
3.	New bearer condition	$\varphi_i = S_i.\varphi$	$\mathbb{L}$
4.	New unit data	$D_i = S_i.D$	$\mathbb{D}$
5.	State log hash	h_s	$\mathbb{H}$
6.	Current bearer condition	φ	$\mathbb{L}$
7.	Current unit data	D	$\mathbb{D}$
8.	Subtree summary value	V	$\mathbb{V}$
9.	Subtree summary hash	h	$\mathbb{H}$
10.	Left child node identifier	ι_L	$\mathbb{I}$
11.	Right child node identifier	ι_R	$\mathbb{I}$
12.	Depth of the subtree	d	$\mathbb{N}_{32}$
13.	Balance factor	b	$\mathbb{N}_8^{\pm}$

4.4.3 Invariants of the State Tree

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

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

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

$x_i = H_0(x_{i-1}, t_i)$ for all $i \in \{2, \ldots, |S|\}$ (Sec. 4.9)
$h_s = \textsc{plain\_tree\_root}(\langle z_1, \ldots, z_{|S|}\rangle)$, where $z_i = H(x_i, H(\varphi_i, D_i))$
$\varphi = \varphi_{|S|}$
$D = D_{|S|}$
$V = F_S(V_s(D), N[\iota_L].V, N[\iota_R].V)$
$h = H(\iota, h_s, V; N[\iota_L].h, N[\iota_L].V; N[\iota_R].h, N[\iota_R].V)$
$d = \max\{N[\iota_L].d, N[\iota_R].d\} + 1$
$b = N[\iota_R].d - N[\iota_L].d$

If $N[\iota] = \perp$, then we define $N[\iota].\varphi \equiv 1$ i.e. owner conditions for non-existing items are true.

This default value is important for unit-creating transactions, where the owner condition of the newly created item is not yet defined. The validity of unit-creating transactions are defined by other parameters, such as ψ_τ (see the validity conditions of transaction types).

4.5 Fee Credit Records

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.

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 $D = (b, \lambda, \ell, t)$, where:

1. b – current balance, of type $\mathbb{N}_{64}^{\pm}$; 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; for consistency, all user interfaces should display all credit balances with 8 digits after the decimal point
2. λ – hash of the last "add fee credit", "close fee credit", "lock fee credit", or "unlock fee credit" operation that targeted this record, of type $\mathbb{H}$; note that spending fee credit when executing other transactions does not affect this "backlink" value
3. ℓ – lock status of the record, of type $\mathbb{N}_{64}$; allows locking of the record at the beginning of a multi-step protocol that needs the record 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 transactions, it only applies to user-initiated actions like adding or reclaiming fee credits
4. t – the earliest time when this record may be "garbage collected" when the balance goes to zero, expressed as the round number of type $\mathbb{N}_{64}$

Table 7. Fee Credit Record.

No	Field	Notation	Type
1.	Current balance	b	$\mathbb{N}_{64}^{\pm}$
2.	Hash of last increment	λ	$\mathbb{H}$
3.	Locking status	ℓ	$\mathbb{N}_{64}$
4.	Minimum lifetime	t	$\mathbb{N}_{64}$

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

4.6 Valid Transaction Orders

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

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

1. $P.\alpha = S.\alpha$ – transaction is sent to this system
2. $f_{\mathcal{SH}}(P.\iota) = S.\sigma$ – target unit is in this shard
3. $n < T_0$ – transaction has not expired
4. $N[\iota] = \bot \ \vee$ VerifyOwner$(N[\iota].\varphi, P, P.s) = 1$ – owner proof verifies correctly
5. ExtrType$(\iota_f) =$ fcr $\ \wedge \ N[\iota_f] \neq \bot$ – the fee payer has credit in this system
6. $s_f = \bot \ \vee$ VerifyOwner$(N[\iota_f].\varphi, P, P.s_f)$ – if the transaction has a fee authorization proof, it must satisfy the owner condition of the fee credit record
7. $s_f \neq \bot \ \vee$ VerifyOwner$(N[\iota_f].\varphi, P, P.s)$ – if the transaction does not have a separate fee authorization proof, the owner proof of the whole transaction must also satisfy the owner condition of the fee credit record

8. $f_m \leq N[\iota_f].D.b$ – the maximum permitted transaction cost does not exceed the fee credit balance
9. $\mathcal{SD}.F_C(P,S) \leq f_m$ – the actual transaction cost does not exceed the maximum permitted by the user
10. $\psi_\tau(P,S)$ – type-specific validity condition holds

The "transfer to fee credit" and "reclaim fee credit" transactions in the money partition (see 5.2.7.3 and 5.2.7.6) and the "lock fee credit", "unlock fee credit", "add fee credit", and "close free credit" transactions in all application partitions (see 5.2.7.1, 5.2.7.2, 5.2.7.4, and 5.2.7.5) 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 8 above do not apply.

4.7 Executing Transactions

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

1. $M_S \leftarrow (S.\mathcal{SD}[\alpha].F_C(P,S), 1, \bot)$ – initialize the transaction processing metadata (these initial values may be overwritten by Action$_\tau$)
2. Action$_\tau$ – execute the type-specific actions
3. Append the transaction record and the new state to the change logs of all units affectected by the transaction; for each $\iota \in$ targets(P):
   - 3.1 $t \leftarrow H(P, M_S)$ – compute the hash of the transaction record
   - 3.2 If $|N[\iota].S| = 0$: – this is a freshly created unit
     - 3.2.1 $x \leftarrow H(\bot, t)$ – 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 $x \leftarrow H(x, t)$ – compute the new head hash of the unit ledger
   - 3.4 $N[\iota].S \leftarrow N[\iota].S \| (t, x, N[\iota].\varphi, N[\iota].D)$ – append to the change log
4. DecrCredit$(P.M_C.\iota_f, M_S.f_a)$ – decrease the balance of the corresponding fee credit record

The "transfer to fee credit" and "reclaim fee credit" transactions in the money partition (see 5.2.7.3 and 5.2.7.6) and the "lock fee credit", "unlock fee credit", "add fee credit", and "close free credit" transactions in all application partitions (see 5.2.7.1, 5.2.7.2, 5.2.7.4, and 5.2.7.5) are special cases: fees are handled intrinsically in those transactions; therefore, step 4 above is skipped when processing those transactions.

4.8 Round Initialization and Completion

4.8.1 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 $N[\iota].S \leftarrow \langle(\bot, x, N[\iota].\varphi, N[\iota].D)\rangle$

2. Delete all fee credit records with zero remaining balance and expired lifetime:

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

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

3. RInit$_\alpha$ – execute the transaction system specific initialization steps

4.8.2 Round Completion: RCompl

The round completion procedure consists of the following steps:

1. RCompl$_\alpha$ – execute the transaction system specific completion steps

4.9 Unit Ledger

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

Unit record is a tuple $R_i = (P_i, C_i^{\text{unit}}, C_i^{\text{tree}}, UC_i)$, where

1. P_i – optional transaction record of type $\mathbb{TR} \cup \{\bot\}$,
2. C_i^{unit} – unit tree certificate,
3. C_i^{tree} – tree certificate,
4. UC_i – unicity certificate.

The unit tree certificate C_i^{unit} is computed from the current parameters φ_ι, D_ι, and the ledger state hash x_ι of the unit ι. The certificate contents also depend on the state values $\varphi'_\iota, D'_\iota, x'_\iota$ of other states of the same unit.

The tree certificate $C_i^{\text{tree}} = C_i^{\text{state}} || C_i^{\text{shard}}$ is concatenation of a state tree certificate C_i^{state} (created by a shard) and a shard tree certificate C_i^{shard} (created by the Core).

The 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

$$x_i = \begin{cases} H_0(x_{i-1}, H_d(P_i)) & \text{if } i > 0 \\ \bot & \text{if } i = 0 \end{cases}$$

where

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

and

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

The structure of a unit ledger is depicted in Fig. 2

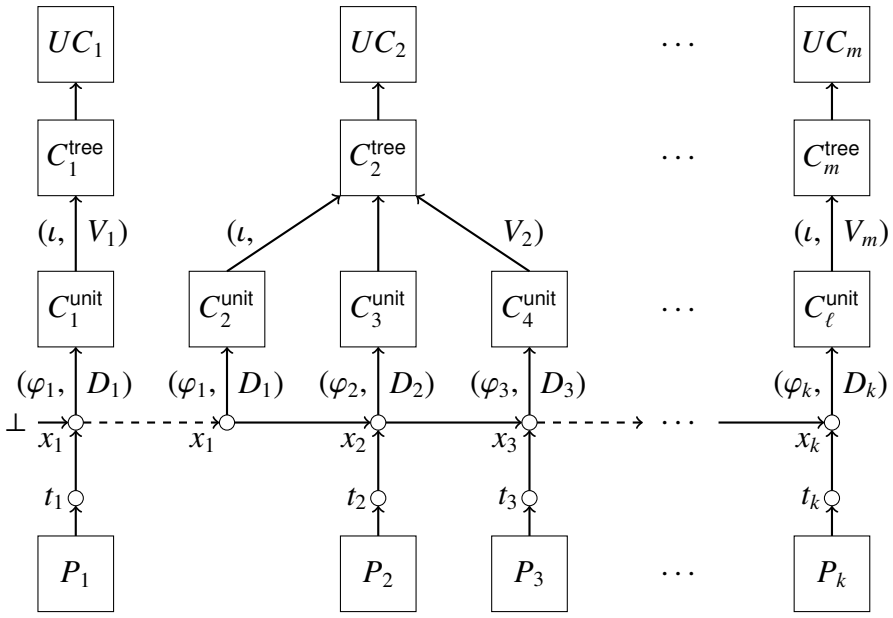

Figure 2. The structure of a unit ledger. In round 1, the unit was created by transaction P_1. In round 2, the unit started in the state copied from round 1 and was then updated by transactions P_2 and P_3. The three states in which the unit was during round 2 have distinct unit tree certificates but they share a common state tree certificate and a common unicity certificate. In round m, the unit was brought to its current state by transaction P_k.

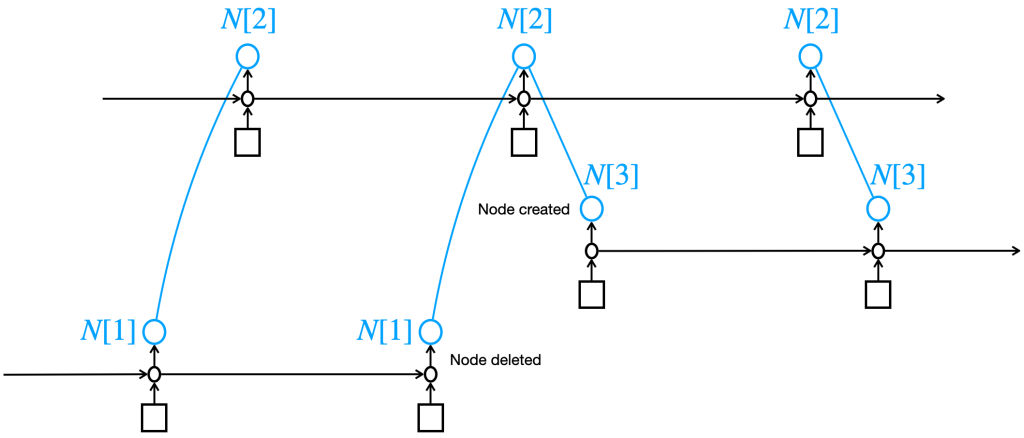

Figure 3. Evolution of the state tree and unit ledgers.

4.10 Unit Tree Certificate

4.10.1 Definition

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

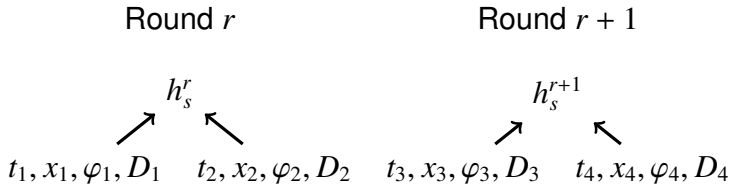

Figure 4. Evolution of a unit's state and the unit trees in two rounds.

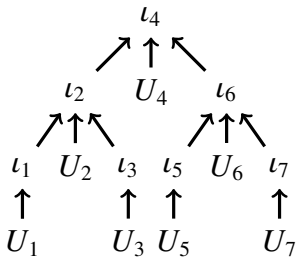

Figure 5. Embedding of unit trees in the state tree: ι_i are unit nodes in the state tree (these form a binary hash tree), U_i are the unit trees linked to each unit node.

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.

4.10.2 **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

Output: Unit tree certificate C^{unit}

Computation:

$n \leftarrow |N[\iota].S|$
for $j \leftarrow 1$ to n **do**
  $y_j \leftarrow H(N[\iota].S_j.\varphi, N[\iota].S_j.D)$
  $z_j \leftarrow H(N[\iota].S_j.x, y_j)$
end for
return $((N[\iota].S_i.t, y_i),$ PLAIN_TREE_CHAIN$(\langle z_1, \ldots, z_n \rangle, i))$

4.11 State Tree Certificate

4.11.1 Definition

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

4.11.2 **Creation:** CreateStateTreeCert

Input:

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

Output: State tree certificate C^{state}

Computation:

$C \leftarrow ()$
$\iota' \leftarrow \iota_r$
while $\iota' \notin \{\iota, 0_{\mathbb{I}}\}$ **do**
  $V \leftarrow \text{SD}.F_S(N[\iota'].D)$
  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

4.11.3 **Create Tree Certificate:** CreateTreeCert

Input:

1. C^{state} – state tree certificate
2. C^{shard} – shard tree certificate

Output: Tree certificate C

Computation: **return** $C^{\text{state}} || C^{\text{shard}}$

4.12 State Mutation Functions

4.12.1 Generic Unit Functions

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

1. AddItem(ι, φ, D) – Adds a unit with identifier ι, owner condition φ, and data D; $N[\iota] \leftarrow (\bot, \langle\rangle, \bot, \varphi, D, V, h, 0_{\mathbb{I}}, 0_{\mathbb{I}}, 1, 0)$, 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
3. SetOwner(ι, φ) – Sets the owner of unit ι to φ; $N[\iota].\varphi \leftarrow \varphi$
4. UpdateData(ι, f) – Applies the update function f to the data of unit ι; $N[\iota].D \leftarrow f(N[\iota].D)$

4.12.2 Fee Credit Functions

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

1. AddCredit$(\iota_f, \varphi_f, v, \lambda, t)$ – Calls AddItem$(\iota_f, \varphi_f, (v, \lambda, 0, t))$; in other words, adds a new credit record $(v, \lambda, 0, t)$ with the owner condition φ_f and 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, λ, t) – Calls UpdateData(ι_f, f), where $f(D) = (D.b + v, \lambda, 0, \max(D.t, t))$; in other words, sets $N[\iota_f].D.b \leftarrow N[\iota_f].D.b + v$, $N[\iota_f].D.\lambda \leftarrow \lambda$, $N[\iota_f].D.\ell \leftarrow 0$, $N[\iota_f].D.t \leftarrow \max(N[\iota_f].D.t, t)$
4. DecrCredit(ι_f, v) – Calls UpdateData(ι_f, f), where $f(D) = (D.b - v, D.\lambda, D.\ell, D.t)$; in other words, sets $N[\iota_f].D.b \leftarrow N[\iota_f].D.b - v$; note that $N[\iota_f].D.\lambda$, $N[\iota_f].D.\ell$, and $N[\iota_f].D.t$ remain unchanged in this operation

4.13 Blocks

4.13.1 Block of a Shard

Block is a tuple $B = \langle(\alpha, \sigma, h_-, v_{\text{prop}}); P_1, \ldots, P_k; UC\rangle$, where:

1. α – system identifier
2. σ – shard identifier
3. h_- – hash of the previous block
4. v_{prop} – block proposer's identifier
5. $P_1, \ldots, P_m$ – transaction records of the block
6. UC – unicity certificate

Existence of a transaction in a block is the proof of execution 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$(B) \neq 0_{\mathbb{H}}$.
If $k = 0$ then the block is *empty block* and BLOCK_HASH$(B) = 0_{\mathbb{H}}$. Empty blocks might still change the state due to internal housekeeping (e.g. state tree pruning, dust bills deletion, etc).

4.13.2 Genesis Block of a Shard

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

1. α – system identifier of type $\mathbb{A}$
2. σ – shard identifier of type $\{0,1\}^{\leq \mathcal{SH}.k}$
3. $\mathcal{SH}$ – sharding scheme of type $\mathbb{SH}$
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 $\mathbb{B}$
9. $\mathcal{SD}$ – system descriptions of type $\mathbb{SD}[\mathbb{A}]$ for all registered transaction systems (including SD$[\alpha]$)

4.13.3 Block Creation: CreateBlock

Input:

1. State $(\alpha, \sigma, \mathcal{SH}, n, e, \iota_r, N, \mathcal{T}, \mathcal{SD})$
2. Sequence of transaction records $P_1, \ldots, P_m$
3. Block hash of the previous block h_-, obtained as $h_- \leftarrow B'.UC.IR.h_B$ or $h_- \leftarrow$ BLOCK_HASH(B'), where B' is the preceding block.

Output: Block $B = \langle (\alpha, \sigma, h_-, \nu); P_1, \ldots, P_k; UC \rangle$

The procedure changes the state.

Computation:

Execute the round initialization procedure RInit
for $i \leftarrow 1 \ldots k$ **do**
  if P_i is valid **then**
    add P_i to the block
    execute P_i
  end if
end for
Execute the round completion procedure RCompl
Record Block Proposer identifier ν_{prop} (defined and verified by the underlying consensus mechanism)
Obtain UC certifying the block
return $B = \langle (\alpha, \sigma, h_-, \nu_{\text{prop}}); P_1, \ldots, P_k; UC \rangle$

4.13.4 Block Verification: VerifyBlock

Input:

1. Block $B = \langle (\alpha, \sigma, h_-, \nu); P_1, \ldots, P_k; UC \rangle$

2. Unicity trust base $\mathcal{T}$

Output: TRUE or FALSE

Computation:

$x \leftarrow$ BLOCK_HASH(B)
if $UC.C^{\text{shard}} \neq \emptyset$ **then**
  $r \leftarrow$ CompUnicityTreeCert$(UC.C^{\text{uni}}$, CompShardTreeCert$(UC.C^{\text{shard}}, UC.IR))$
else
  $r \leftarrow$ CompUnicityTreeCert$(UC.C^{\text{uni}}, UC.IR)$
end if
return VerifyUnicitySeal$(r, UC.C^{\text{r}}, \mathcal{T}) \ \wedge \ UC.IR.h_B = x$

4.13.5 Block Hash: BLOCK_HASH

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

Input: Block $B = \langle(\alpha, \sigma, h_{-}, \nu); P_1, \ldots, P_k; UC\rangle$

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

Computation:

function BLOCK_HASH(B)
  if $k = 0$ **then** ▷ Empty block
    return $0_{\mathbb{H}}$
  else
    for $i \leftarrow 1$ to k **do**
      $h_i \leftarrow H(P_i)$
    end for
    return $H(H(\alpha, \sigma, h_{-}, \nu)$, PLAIN_TREE_ROOT$(\langle h_1, \ldots, h_k\rangle))$
  end if
end function

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. 6) that change the ownership conditions 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 exchange are needed.

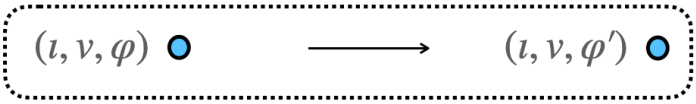

Figure 6. Bill transfer.

5.1.2 Extended Bill Money Scheme

Extended bill money scheme addresses the shortcomings of pure bill schemes by introducing split type payments (Fig. 7) that make exact payments always possible.

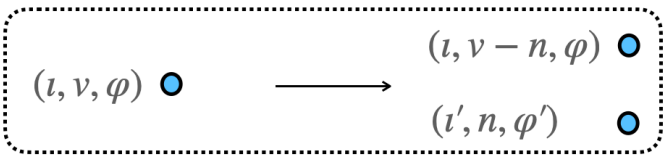

Figure 7. Bill split.

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. 8) and get a proof of having done so.

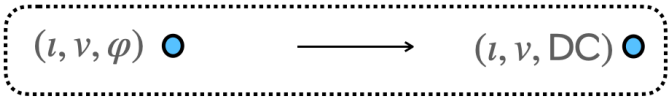

Figure 8. Transfer of dust bills to Dust Collector.

By presenting those proofs to the system, users can then obtain a new, larger-value bill via swapDC transactions (Fig. 9).

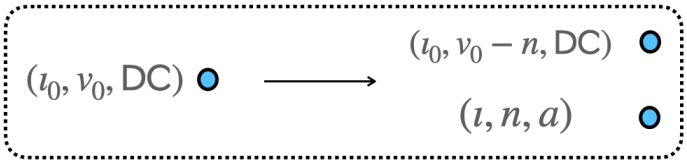

Figure 9. Swap with Dust Collector.

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_0. 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. 10, 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 (ι_0, 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.

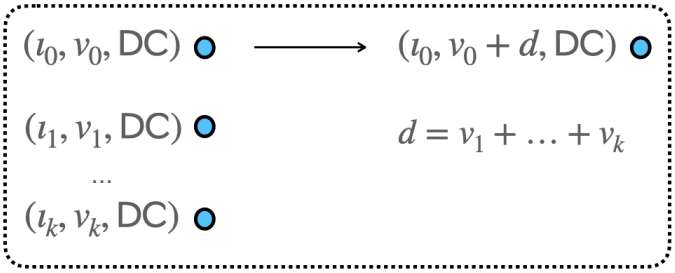

Figure 10. Dust collection.

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 $v_{\text{swapDC}} = 0$.

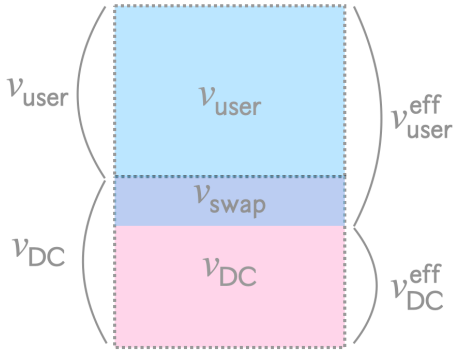

Figure 11. Types of money and money invariants.

5.2 Specification of the Money Partition

5.2.1 Parameters, Types, Constants, Functions

System identifier: α_{money}

Type and unit identifier lengths: tidlen $= 8$, uidlen $= 256$

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

Data types:

- $\mathbb{D}_{\text{bill}}$: tuples (v, t, λ, ℓ) where:
  1. 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
  2. t – the partition round number of the last transaction with the bill of type $\mathbb{N}_{64}$
  3. λ – backlink of type $\mathbb{H}$; hash value computed over all fields of the previous transaction order with the same bill; $\lambda = H(P)$
  4. 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
- $\mathbb{D}_{\text{fcr}} = (b, \lambda, \ell, t)$, where
  1. $b \in \mathbb{N}_{64}^{\pm}$ is the current balance of this record, in fixpoint format with 8 fractional decimal digits
  2. $\lambda \in \mathbb{H}$ is the hash of the last addFC, closeFC, lockFC, or unlockFC transaction for this record
  3. $\ell \in \mathbb{N}_{64}$ is the lock status of the record; $\ell = 0$ means the record is not locked, any other value means it's locked
  4. $t \in \mathbb{N}_{64}$ is the minimum lifetime of this record

Summary functions:

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

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

Transaction types: $\mathbb{T} = \{$trans, split, lock, unlock, transDC, swapDC, lockFC, unlockFC, transFC, addFC, closeFC, reclFC$\}$ (transfer a bill, split a bill, lock a bill, unlock a bill, transfer to dust collector, swap with dust collector, lock a fee credit record, unlock a fee credit record, transfer to fee credit, add fee credit, close fee credit, reclaim fee credit)

5.2.2 Transfer

Transaction order $P = \langle(\alpha, \text{trans}, \iota, A, M_C), s, s_f\rangle$ with $A = (v, \varphi, \eta, \lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. v – amount to transfer, of type $\mathbb{N}_{64}$
2. φ – new bearer condition, of type $\mathbb{L}$
3. η – optional nonce, of type $\{0, 1\}^*$
4. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

$$\begin{aligned}&\psi_{\text{trans}}(P, S) \equiv \\ &\quad \text{ExtrType}(P.\iota) = \text{bill} \ \wedge \ S.N[P.\iota] \neq \bot \ \wedge \\ &\quad S.N[P.\iota].\ell = 0 \ \wedge \\ &\quad P.A.v = S.N[P.\iota].D.v \ \wedge \\ &\quad P.A.\lambda = S.N[P.\iota].D.\lambda\end{aligned}$$

That is,

- ι identifies an existing bill
- the bill is not locked
- the value to be transferred is the value of the bill
- the transaction follows the previous valid transaction with the bill

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

1. SetOwner$(\iota, A.\varphi)$
2. UpdateData(ι, f), where $f(D) = (D.v, S.n, H(P), D.\ell)$

Table 8. Data fields of the trans transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_{money}
2.	transaction type	τ	$\mathbb{T}$	trans
3.	unit identifier	ι	$\mathbb{I}$	-
4.	amount to transfer	$A.v$	$\mathbb{N}_{64}$	-
5.	new owner condition	$A.\varphi$	$\mathbb{L}$	-
6.	nonce	$A.\eta$	$\{0,1\}^* \cup \{\bot\}$	-
7.	backlink	$A.\lambda$	$\mathbb{H}$	-
8.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
9.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
10.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	-
11.	owner proof	s	$\{0,1\}^*$	-
12.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	-
13.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

5.2.3 Split

Transaction order $P = \langle(\alpha, \text{split}, \iota, A, M_C), s, s_f\rangle$ with $A = (v_1, \ldots, v_m; \varphi_1, \ldots, \varphi_m; \nu', \eta, \lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. $v_1, \ldots, v_m$ – amounts to transfer, of type $\mathbb{N}_{64}$

2. $\varphi_1, \ldots, \varphi_m$ – new bearer conditions, of type $\mathbb{L}$
3. v' – remaining value, of type $\mathbb{N}_{64}$
4. η – optional nonce, of type $\{0,1\}^*$
5. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

$$\begin{aligned}&\psi_{\mathsf{split}}(P,S) \equiv \\ &\quad \mathsf{ExtrType}(P.\iota) = \mathsf{bill} \ \wedge \ S.N[P.\iota] \neq \bot \ \wedge \\ &\quad S.N[P.\iota].\ell = 0 \ \wedge \\ &\quad P.A.v_1 + \ldots + P.A.v_m + P.A.v' = S.N[P.\iota].D.v \ \wedge \\ &\quad P.A.v_1 > 0 \ \wedge \ \ldots \ \wedge \ P.A.v_m > 0 \ \wedge \ P.A.v' > 0 \ \wedge \\ &\quad P.A.\lambda = S.N[P.\iota].D.\lambda\end{aligned}$$

That is,

- ι identifies an existing bill
- the bill is not locked
- the sum of the values to be transferred plus the remaining value equals the value of the bill
- the values to be transferred and also the remaining value are all non-zero
- the transaction follows the previous valid transaction with the bill

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

1. for $i = 1, \ldots, m$:
   1.1 $\iota_i \leftarrow \mathsf{NodeID}(\mathsf{bill}, \mathsf{PrndSh}(\mathsf{ExtrUnit}(P.\iota), P.\iota \| P.A \| P.M_C \| i))$, i.e. generate a new bill identifier in the same shard
   1.2 $\mathsf{AddItem}(\iota_i, P.A.\varphi_i, (P.A.v_i, S.n, H(P,i), 0))$ – create a new bill ι_i with value v_i and owner condition φ_i
2. $\mathsf{UpdateData}(\iota, f)$, where $f(D) = (P.A.v', S.n, H(P), D.\ell)$

Targets: For split transaction P, $\mathsf{targets}(P) = \{P.\iota, \iota_1, \iota_2, \ldots, \iota_m\}$.

5.2.4 Lock

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

1. ℓ – new lock status, of type $\mathbb{N}_{64}$
2. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

$$\begin{aligned}&\psi_{\mathsf{lock}}(P,S) \equiv \\ &\quad \mathsf{ExtrType}(P.\iota) = \mathsf{bill} \ \wedge \ S.N[P.\iota] \neq \bot \ \wedge \\ &\quad S.N[P.\iota].\ell = 0 \ \wedge \\ &\quad P.A.\ell > 0 \ \wedge \\ &\quad P.A.\lambda = S.N[P.\iota].D.\lambda\end{aligned}$$

Table 9. Data fields of the split transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_{money}
2.	transaction type	τ	$\mathbb{T}$	split
3.	unit identifier	ι	$\mathbb{I}$	-
4.	amounts to transfer	$A.v_1, \ldots, A.v_m$	$\mathbb{N}_{64}$	-
5.	new owner conditions	$A.\varphi_1, \ldots, A.\varphi_m$	$\mathbb{L}$	-
6.	remaining value	$A.v'$	$\mathbb{N}_{64}$	-
7.	nonce	$A.\eta$	$\{0,1\}^* \cup \{\bot\}$	-
8.	backlink	$A.\lambda$	$\mathbb{H}$	-
9.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
10.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
11.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	-
12.	owner proof	s	$\{0,1\}^*$	-
13.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	-
14.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

That is,

- ι identifies an existing bill
- the bill is not locked
- the new status is a “locked” one
- the transaction follows the previous valid transaction with the bill

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

1. UpdateData(ι, f), where $f(D) = (D.v, S.n, H(P), P.A.\ell)$

5.2.5 Unlock

Transaction order $P = \langle(\alpha, \text{unlock}, \iota, A, M_C), s, s_f\rangle$ with $A = (\lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\text{unlock}}(P, S) \equiv \\
&\quad \text{ExtrType}(P.\iota) = \text{bill} \ \wedge \ S.N[P.\iota] \neq \bot \ \wedge \\
&\quad S.N[P.\iota].\ell > 0 \ \wedge \\
&\quad P.A.\lambda = S.N[P.\iota].D.\lambda
\end{aligned}
$$

That is,

- ι identifies an existing bill
- the bill is locked
- the transaction follows the previous valid transaction with the bill

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

1. UpdateData(ι, f), where $f(D) = (D.v, S.n, H(P), 0)$

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:

- 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 lock transaction.
- 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.
- The value of the dust bills is added to the target bill using a swapDC transaction. As this transaction completes the dust collection process, it also unlocks the target bill.

5.2.6.1 Transfer to Dust Collector

Transaction order $P = \langle(\alpha, \text{transDC}, \iota, A, M_C), s, s_f\rangle$, with $A = (v, \iota', \lambda', \lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. v – target value, of type $\mathbb{N}_{64}$
2. ι' – identifier of the target bill, of type $\mathbb{I}$
3. λ' – current backlink of the target bill, of type $\mathbb{H}$
4. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

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

That is,

- ι identifies an existing bill
- the bill is not locked
- the target value equals the value of the bill
- the transaction follows the previous valid transaction with the bill

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

1. SetOwner(ι, DC)
2. UpdateData(ι_0, f'), where $f'(D) = (D.v + P.A.v, S.n, H(P), D.\ell)$ – increase DC money supply
3. UpdateData(ι, f), where $f(D) = (0, S.n, H(P), D.\ell)$ – decrease bill value

Table 10. Data fields of the transDC transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_{money}
2.	transaction type	τ	$\mathbb{T}$	transDC
3.	unit identifier	ι	$\mathbb{I}$	-
4.	target value	$A.v$	$\mathbb{N}_{64}$	-
5.	target identifier	$A.\eta$	$\mathbb{I}$	-
6.	target backlink	$A.\lambda'$	$\mathbb{H}$	-
7.	backlink	$A.\lambda$	$\mathbb{H}$	-
8.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
9.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
10.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	-
11.	owner proof	s	$\{0,1\}^*$	-
12.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	-
13.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

5.2.6.2 Swap with Dust Collector

Transaction order $P = \langle(\alpha, \text{swapDC}, \iota, A, M_C), s, s_f\rangle$, with $A = (v, \varphi; P_{k_1}, \ldots, P_{k_\ell}; \Pi_{k_1}, \ldots, \Pi_{k_\ell})$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. v – target value, of type $\mathbb{N}_{64}$
2. φ – target owner condition, of type $\mathbb{L}$
3. $P_{k_1}, \ldots, P_{k_\ell}$ – sequence of bill transfer records, of type transDC
4. $\Pi_{k_1}, \ldots, \Pi_{k_\ell}$ – transaction execution proofs of $P_{k_1}, \ldots, P_{k_\ell}$, of type $\mathbb{XP}$

Transaction-specific validity condition:
$\psi_{\text{swapDC}}(P, S)$ is logical conjunction of the following conditions:

1. $P.A.v = P_{k_1}.A.v + \ldots + P_{k_\ell}.A.v$ – target value is the sum of the values of the transDC payments
2. $P.A.v \leq N[\iota_0].D.v$ – there is sufficient DC-money supply
3. $\text{ExtrType}(P.\iota) = \text{bill} \wedge N[\iota] \neq \bot$ – ι identifies an existing bill
4. $P_{k_1}.\alpha = \ldots = P_{k_\ell}.\alpha = \alpha_{\text{money}}$ – transfers were in the money partition
5. $P_{k_1}.\tau = \ldots = P_{k_\ell}.\tau = \text{transDC}$ – bills were transferred to DC

Table 11. Data fields of the swapDC transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_{money}
2.	transaction type	τ	$\mathbb{T}$	swapDC
3.	unit identifier	ι	$\mathbb{I}$	-
4.	target value	$A.v$	$\mathbb{N}_{64}$	-
5.	owner condition	$A.\varphi$	$\mathbb{L}$	-
6.	transfer records	$A.P_{k_1}, \ldots, A.P_{k_\ell}$	transDC	-
7.	transaction proofs	$A.\Pi_{k_1}, \ldots, A.\Pi_{k_\ell}$	$\mathbb{XP}$	-
8.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
9.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
10.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	-
11.	owner proof	s	$\{0,1\}^*$	-
12.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	-
13.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

6. $P_{k_1}.\iota < \ldots < P_{k_\ell}.\iota$ – transfer orders are listed in strictly increasing order of bill identifiers (in particular, this ensures that no source bill can be included multiple times)
7. $P_{k_1}.A.\iota' = \ldots = P_{k_\ell}.A.\iota' = \iota$ – bill transfer orders contain correct target identifiers
8. $P_{k_1}.A.\lambda' = \ldots = P_{k_\ell}.A.\lambda' = S.N[P.\iota].D.\lambda$ – bill transfer orders contain correct target backlinks
9. VerifyTxProof$(\Pi_{k_1}, P_{k_1}, S.\mathcal{T}, S.\mathcal{SD}) \wedge \ldots \wedge$ VerifyTxProof$(\Pi_{k_\ell}, P_{k_\ell}, S.\mathcal{T}, S.\mathcal{SD})$ – transaction proofs of the bill transfer orders verify

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

1. UpdateData(ι_0, f), where $f(D) = (D.v - P.A.v, S.n, H(P), D.\ell)$ – reduce DC money supply by $P.A.v$
2. UpdateData(ι, f), where $f(D) = (D.v + P.A.v, S.n, H(P), 0)$ – increase the value of ι by $P.A.v$

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:

- If the target fee credit record exists, it should be locked using a lockFC transaction in the target partition.
- 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.

- 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 also unlocks the target record.

And for reclaiming fee credits:

- The target bill should be locked using a lock transaction in the money partition.
- 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.
- 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 also unlocks the target bill.

5.2.7.1 Lock Fee Credit Record

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

1. ℓ – new lock status, of type $\mathbb{N}_{64}$
2. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

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

That is,

- ι identifies an existing fee credit record
- the record is not locked
- the new status is a “locked” one
- the transaction follows the previous valid transaction with the record
- the transaction fee can’t exceed the record balance
- there’s no fee credit reference or separate fee authorization proof

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

1. UpdateData(ι, f), where $f(D) = (D.b - M_S.f_a, H(P), P.A.\ell, D.t)$

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

5.2.7.2 Unlock Fee Credit Record

Transaction order $P = \langle(\alpha, \text{unlockFC}, \iota, A, M_C), s, s_f\rangle$ with $A = (\lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

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

That is,

- ι identifies an existing fee credit record
- the record is locked
- the transaction follows the previous valid transaction with the record
- the transaction fee can't exceed the record balance
- there's no fee credit reference or separate fee authorization proof

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

1. UpdateData(ι, f), where $f(D) = (D.b - M_S.f_a, H(P), 0, D.t)$

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 Transfer to Fee Credit

Transaction order $P = \langle(\alpha, \text{transFC}, \iota, A, M_C), s, s_f\rangle$, with $A = (v, \alpha', \iota_f, t_b, t_e, \lambda', \lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. v – amount to transfer, of type $\mathbb{N}_{64}$
2. α' – target system identifier, of type $\mathbb{A}$
3. ι_f – target fee credit record identifier, of type $\mathbb{I}_{\alpha'}$ (the type of unit identifiers in the target system)
4. t_b – earliest round in which the corresponding "add fee credit" transaction can be executed in the target system, of type $\mathbb{N}_{64}$
5. t_e – latest round in which the corresponding "add fee credit" transaction can be executed in the target system, of type $\mathbb{N}_{64}$
6. λ' – target backlink, of type $\mathbb{H} \cup \{\bot\}$; for the proof of transfer to be usable in a following addFC operation, this must be set to the current state hash of the target credit record if the record exists, or to $\bot$ if the record does not exist yet

Table 12. Data fields of the transFC transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_{money}
2.	transaction type	τ	$\mathbb{T}$	transFC
3.	unit identifier	ι	$\mathbb{I}$	-
4.	amount	$A.v$	$\mathbb{N}_{64}$	-
5.	target system identifier	$A.\alpha'$	$\mathbb{A}$	-
6.	target record identifier	$A.\iota_f$	$\mathbb{I}_{\alpha'}$	-
7.	earliest addition time	$A.t_b$	$\mathbb{N}_{64}$	-
8.	latest addition time	$A.t_e$	$\mathbb{N}_{64}$	-
9.	target backlink	$A.\lambda'$	$\mathbb{H} \cup \{\bot\}$	-
10.	backlink	$A.\lambda$	$\mathbb{H}$	-
11.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
12.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
13.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	$\bot$
14.	owner proof	s	$\{0,1\}^*$	-
15.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	$\bot$
16.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

7. λ – backlink, of type $\mathbb{H}$

Transaction-specific validity condition:

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

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 transaction follows the previous valid transaction with the bill
- the the transaction fee can't exceed the transferred amount
- there's no fee credit reference or separate fee authorization proof

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

1. UpdateData(ι, f), where $f(D) = (D.v - P.A.v, S.n, H(P), D.\ell)$

Note: The transferred credits will be aggregated and added to the target system'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.

5.2.7.4 Add Fee Credit

Transaction order $P = \langle(\alpha, \text{addFC}, \iota, A, M_C), s, s_f\rangle$, with $A = (\varphi, P', \Pi')$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. $\varphi \in \mathbb{L}$ – target fee credit record owner condition
2. P' – bill transfer record of type transFC
3. Π' – transaction proof of P', of type $\mathbb{XP}$

Validity Condition

$\psi_{\text{addFC}}(P, S)$ is logical conjunction of the following conditions:

1. ExtrType$(P.\iota) =$ fcr – target unit is a fee credit record,
2. $S.N[P.\iota] = \bot \vee S.N[P.\iota].\varphi = P.A.\varphi$ – if the target exists, the owner condition matches,
3. VerifyTxProof$(\Pi', P', S.\mathcal{T}, S.\mathcal{SD})$ – proof of the bill transfer order verifies,
4. $P'.\alpha = \alpha_{\text{money}} \wedge P'.\tau =$ transFC – bill was transferred to fee credits,
5. $P'.A.\alpha = P.\alpha$ – bill was transferred to fee credits for this system,
6. $P'.A.\iota_f = P.\iota$ – bill was transferred to fee credits of the target record,
7. $(S.N[P.\iota] = \bot \wedge P'.A.\lambda' = \bot) \vee (S.N[P.\iota] \neq \bot \wedge P'.A.\lambda' = S.N[P.\iota].\lambda)$ – bill transfer order contains correct target backlink value,
8. $P'.A.t_b \leq t \leq P'.A.t_e$, where t is the number of the current block being composed – bill transfer is valid to be used in this block,
9. $P'.M_S.f_a + P.M_C.f_m \leq P'.A.v$ – the transaction fees can't exceed the transferred value,
10. $P.M_C.\iota_f = \bot \wedge s_f = \bot$ – there's no fee credit reference or separate fee authorization proof.

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

1. $v' \leftarrow P'.A.v - P'.M_S.f_a - M_S.f_a$ – the net value of credit
2. if $S.N[P.\iota] = \bot$:
   2.1 AddCredit$(P.\iota, P.A.\varphi, v', H(P), P.A.t_e + 1)$
3. else:
   3.1 IncrCredit$(P.\iota, v', H(P), P.A.t_e + 1)$

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

5.2.7.5 Close Fee Credit

Transaction order $P = \langle(\alpha, \text{closeFC}, \iota, A, M_C), s, s_f\rangle$, $A = (v, \iota', \lambda')$, $M_C = (T_0, f_m, \iota_f)$, where:

1. $v \in \mathbb{N}_{64}$ – amount
2. $\iota' \in \mathbb{I}_{\text{money}}$ – target bill
3. $\lambda' \in \mathbb{H}$ – target backlink; for the proof of closure to be usable in a following reclFC operation, this must be set to the current state hash of the target bill

Validity Condition

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

That is,

- ι identifies an existing fee credit record,
- the record is not locked,
- the amount is the current balance of the record,
- the transaction fee can't exceed the current balance of the record,
- there's no fee credit reference or separate fee authorization proof.

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

1. $\text{DecrCredit}(\iota, S.N[\iota].b)$
2. $\text{UpdateData}(\iota, f$, where $f(D) = (D.b, H(P), D.\ell, D.t)$

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

5.2.7.6 Reclaim Fee Credit

Transaction order $P = \langle(\alpha, \text{reclFC}, \iota, A, M_C), s, s_f\rangle$, with $A = (P', \Pi', \lambda)$ and $M_C = (T_0, f_m, \iota_f)$, where:

1. P' – fee credit closure order, of type closeFC
2. Π' – transaction proof of P', of type $\mathbb{XP}$
3. λ – backlink, of type $\mathbb{H}$

Table 13. Data fields of the reclFC transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_{money}
2.	transaction type	τ	$\mathbb{T}$	reclFC
3.	unit identifier	ι	$\mathbb{I}$	-
4.	credit closure order	$A.P'$	closeFC	-
5.	transaction proof	$A.\Pi'$	$\mathbb{XP}$	-
6.	backlink	$A.\lambda$	$\mathbb{H}$	-
7.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
8.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
9.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	$\bot$
10.	owner proof	s	$\{0,1\}^*$	-
11.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	$\bot$
12.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\text{reclFC}}(P, S) \equiv \\
&\quad S.N[P.\iota] \neq \bot \wedge \\
&\quad \text{VerifyTxProof}(\Pi', P', S.\mathcal{T}, S.\mathcal{SD}) \wedge \\
&\quad P'.\tau = \text{closeFC} \wedge \\
&\quad P'.A.\iota' = P.\iota \wedge \\
&\quad P'.A.\lambda' = S.N[P.\iota].D.\lambda \wedge \\
&\quad P'.M_S.f_a + P.M_C.f_m \leq P'.A.v \wedge \\
&\quad P.A.\lambda = S.N[P.\iota].D.\lambda \wedge \\
&\quad P.M_C.\iota_f = \bot \ \wedge \ s_f = \bot
\end{aligned}
$$

That is,

- ι identifies an existing bill
- the proof of the credit closure order verifies
- the order is a credit closure
- the order targets the current bill
- the order contains the correct target backlink value
- the transaction fees can't exceed the transferred value
- the transaction follows the previous valid transaction with the bill
- there's no fee credit reference or separate fee authorization proof

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

1. $v' \leftarrow P'.A.v - P'.M_S.f_a - M_S.f_a$ – net value reclaimed

2. UpdateData(ι, f), where $f(D) = (D.v + v', S.n, H(P), 0)$

Note: The reclaimed credits will be aggregated and removed from the target system'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.

5.2.8 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:
   $\mathcal{I} \leftarrow \{\iota : \text{ExtrType}(\iota) = \text{bill} \ \wedge \ N[\iota] \neq \perp \ \wedge \ N[\iota].D.v = 0 \ \wedge \ N[\iota].D.T_{\text{dust}} < S.n\}$
   1.2 For each known transaction system identifier α:
   $\mathcal{I} \leftarrow \mathcal{I} \backslash \{S.\mathcal{SD}[\alpha].\iota_{FC}\}$
   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 transaction system identifier α:
      1.1.1 Compute v_+ as the sum of $P.A.v - P.M_S.f_a$ over all transFC records P with $P.A.\alpha = \alpha$ in the current block
      1.1.2 Compute v_- as the sum of $P.A.P'.A.v - P.A.P'.M_S.f_a$ over all reclFC records P with $P.A.P'.\alpha = \alpha$ in the current block
      1.1.3 UpdateData($S.\mathcal{SD}[\alpha].\iota_{FC}, f$), where $f(D) = (D.v + v_+ - v_-, D.t, D.\lambda, D.\ell)$
   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 UpdateData($S.\mathcal{SD}[\alpha_{\text{money}}].\iota_{FC}, f$), where $f(D) = (D.v + v, D.t, D.\lambda, D.\ell)$

6 Atomicity Partition Type 2.0

6.1 Motivation and General Description

6.1.1 Motivation

Let $u_1, \ldots, u_m$ be units with identifiers $\iota_1, \ldots, \iota_m$ and owner conditions $\varphi_1, \ldots, \varphi_m$, respectively.

The units $u_1, \ldots, u_m$ may belong to different transaction systems (partitions) with identifiers $\alpha_1, \ldots, \alpha_m$, respectively. It is assumed that in all these partitions there are transaction types for changing the ownership conditions of units.

Goal: transfer the units atomically to new owner conditions $\varphi'_1, \ldots, \varphi'_m$ so that either:

- *all transfers happen* – all units $u_1, \ldots, u_m$ are transferred to the new owner conditions $\varphi'_1, \ldots, \varphi'_m$, or
- *none of the transfers happen* – all units will have owner conditions equivalent to the previous conditions $\varphi_1, \ldots, \varphi_m$

The units may potentially be controlled by different parties. We assume that these parties may communicate in order to agree on the atomic transfer, i.e. after communication, all parties know $\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m, \varphi_1, \ldots, \varphi_m$. The parties also agree on other transaction specific parameters.

If there is more than one party, this is an *atomic swap*. If there is a single party, this is an *atomic multi-unit transfer*.

6.1.2 General Description of the Atomicity Partition

There is a specific transaction system (partition) with identifier α_0 that provides necessary unique references for atomic multi-unit transactions. We call this *atomicity partition*.

Units of the atomicity partition are the atomic multi-unit transactions, i.e. every such transaction has a unique pseudo-random identifier ι (referred to as *contract identifier*) in the atomicity partition.

There is no ownership or value for the transactions themselves; $\varphi \equiv 1$ for every such unit.

Transactions of the atomicity partition are:

1. fin – final status transaction that is generated automatically by the atomicity partition and that indicates the final status of the contract ι
2. con – confirming a transaction with contract identifier ι

The data D of the units of the atomicity partition consists of the following components:

1. count – number of component transactions involved
2. nonce – nonce of type $\mathbb{N}_{128}$
3. begin – round number
4. end – round number
5. confirmed – set of tuples (i, P', Π), where i is the index of the target of P' among $u_1, \ldots, u_m$ and Π is a proof that the transaction in the record P' was executed in the appropriate partition.

6.2 Phases of Atomic Multi-Unit Transactions

6.2.1 Phase 1: Preparation

Parties $\mathcal{P}_1, \ldots, \mathcal{P}_m$ prepare transaction orders $P_1, \ldots, P_m$ that transfer the ownerships of the units $\iota_1, \ldots, \iota_m$ to special parameterised owner predicates $\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \varphi_1, \varphi'_1; \cdot; \cdot, \cdot, \cdot), \ldots, \varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \varphi_m, \varphi'_m; \cdot; \cdot, \cdot, \cdot)$

The contract identifier ι is computed with a deterministic collision-resistant function $f(r, \tilde{P}_1, \ldots, \tilde{P}_m)$ on:

1. A nonce r of type $\mathbb{N}_{128}$
2. The reduced payment orders $\tilde{P}_1, \ldots, \tilde{P}_m$ without signatures (owner proofs) and with the reduced owner predicates $\tilde{\varphi}_{\text{ato}}(\alpha_0, \bigsqcup, t_b, t_e, \varphi_i, \varphi'_i; \cdot; \cdot, \cdot, \cdot)$, where the identifier ι is replaced with empty string $\bigsqcup$.

The predicate $\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \varphi, \varphi'; \cdot; \cdot, \cdot, \cdot)$ is defined as follows:

$\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \varphi, \varphi'; P; \Pi, P_{\text{fs}}, s)$, where the triple (Π, P_{fs}, s) represents the owner proof, is true if and only if one of the following conditions holds:

1. $\varphi'(P, s) = 1$, and Π of type $\mathbb{XP}$ is a proof that the final status message $P_{\text{fs}} = \langle \text{fin}, \iota, 1 \rangle$ is included in round t $(t_b < t \leq t_e)$ of the partition α_0, i.e.

$$\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \varphi, \varphi'; \cdot; \Pi, P_{\text{fs}}, \cdot) \equiv \varphi'(\cdot, \cdot)$$

2. $\varphi(P, s) = 1$, and Π of type $\mathbb{XP}$ is a proof that the final status message $P_{\text{fs}} = \langle \text{fin}, \iota, 0 \rangle$ is included in round t $(t_b < t \leq t_e)$ of the partition α_0, i.e.

$$\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \varphi, \varphi'; \cdot; \Pi, P_{\text{fs}}, \cdot) \equiv \varphi(\cdot, \cdot)$$

3. $\varphi(P, s) = 1$, and Π of type $\mathbb{SP}$ is a proof that there is no unit with identifier ι in the state tree of round t_e of the partition α_0.

Therefore, the next transaction P with the unit can only be made if one of the final status messages $\langle \text{fin}, \iota, 1 \rangle$, or $\langle \text{fin}, \iota, 0 \rangle$ is stored in the blockchain of the atomicity partition, or, as a (rare) backup case, when it is proved that no units with identifier ι exist at the round t_e of the atomicity partition, which means that none of the con transactions reached the atomicity partition.

6.2.2 Phase 2: Confirmation

All parties sign their transaction orders $P_1, \ldots, P_m$ (by adding ownership proofs s_i to P_i) and send them to the corresponding partitions $\alpha_1, \ldots, \alpha_m$.

Parties obtain transaction proofs $\Pi_1, \ldots, \Pi_m$ for the records of transactions $P_1, \ldots, P_m$.

Every party i sends the confirmation message $\langle \alpha_0, \text{con}, \iota, A_i, M_C \rangle$ to the atomicity partition with $A_i = (m, r, i, P_i, \Pi_i)$.

6.2.2.1 Actions of the Confirmation Message

Let t be the current round number of the atomicity partition.

Having received a message $\langle \alpha_0, \text{con}, \iota, (m, r, i, P', \Pi), M_C \rangle$, the atomicity partition does the following.

1. If any of the following conditions fails, discard the message:
   1.1 $m \geq 2$ and $1 \leq i \leq m$
   1.2 P' contains a locking script of type $\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \ldots)$ with t_b and t_e such that $t_b < t \leq t_e$, where t is the current round number of the atomicity partition.
   1.3 Π is a proof that the transaction in the record P' was executed in the appropriate partition.
2. If unit ι does not exist, then add a new unit with:
   - count = m,
   - nonce = r
   - begin = t_b
   - end = t_e
   - confirmed = $\{(i, P, \Pi)\}$.
3. If unit ι exists:
   3.1 If m = count, r = nonce, t_b = begin, and t_e = end then:
      3.1.1 confirmed $\leftarrow$ confirmed $\cup \{(i, P, \Pi)\}$
      3.1.2 If confirmed contains confirmations for all m messages $P_1, \ldots, P_m$, then:
         3.1.2.1 If $\iota = f(r, \tilde{P}_1, \ldots, \tilde{P}_m)$ then add $P_{\text{fs}} = \langle \text{fin}, \iota, 1 \rangle$ to the current block.
         3.1.2.2 If $\iota \neq f(r, \tilde{P}_1, \ldots, \tilde{P}_m)$ then add $P_{\text{fs}} = \langle \text{fin}, \iota, 0 \rangle$ to the current block.

6.2.3 Other Actions of the Atomicity Partition

At the beginning of every block with round number t:

1. For all units ι with end $< t$, delete the unit ι from the state tree.
2. If any units were deleted, record this event by adding a deleteObsRec record to the current block.

At the end of every block with round number t:

1. For all units ι with end $= t$ for which confirmed does not contain all confirmations, add $P_{\text{fs}} = \langle \text{fin}, \iota, 0 \rangle$ to the current block.

6.2.4 Explanations

All the multi-unit transactions for which at least one of the con reaches the atomicity partition will have final status during the atomicity partition rounds $t_b + 1, \ldots, t_e$ and the corresponding fin transaction can be found in blocks (and the state tree) of those rounds.

All con transactions received after t_e will be discarded. Hence, if no con transactions were received during $t_b + 1, \ldots, t_e$, then there will be no units with identifier ι in the state tree of the atomicity partition.

Any attempt to trick the atomicity partition to issue the final status $\langle \text{fin}, \iota, 1 \rangle$ by sending confirmation messages with ι fails, as changing the timeouts t_b, t_e will also change ι as long as f is collision-resistant.

6.2.5 Memory Size of the Atomicity Partition

Let m be the average number of units involved in the multi-unit atomic transactions. Let ρ be the average number of multi-unit transactions per block. Let N be the total number of units in all partitions of the AlphaBill system. Let k be the bit-length of the hash values. Let t_{tr} be the average value of $t_e - t_b$.

The maximum size of the data part D of a state tree node is $\approx m \cdot k \cdot \log_2 N$ bits, which is dominated by the size of m block inclusion proofs each of size $\approx k \cdot \log_2 N$ bits. Together with the identifier and two pointers, a node has $3k$ additional bits.

There are about $t_{\text{tr}} \cdot \rho$ nodes in the tree and hence the total size of the state tree in bits is:

$$t_{\text{tr}} \cdot \rho \cdot k \cdot (m \cdot \log_2 N + 3)$$

Say $\rho = 10^6$, $m = 3$, $k = 256$, $t_{\text{tr}} = 10$, $N = 10^{12}$. Then $\log_2 N \approx 40$ and the total size of the state tree is about 40 Gbytes.

One machine may not be able to handle 10^6 transactions per block. Hence, sharding may be necessary. If one shard can handle $10\,000$ requests per block, then we need 100 shards, and hence, every shard needs only $40/100 = 0.4$ Gbytes = 400 Mbytes of memory.

6.3 Specification of the Atomicity Partition

6.3.1 Parameters, Types, Constants, Functions

System identifier: α_0

Type and unit identifier lengths: tidlen = 8, uidlen = 256

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

Summary trust base: $\mathcal{V} = 0$

Summary check: $\gamma \equiv 1$

Unit types: $\mathcal{U} = \{\text{ato}, \text{fcr}\}$ (multi-unit atomic transactions, fee credit records).

Data types:

- $\mathbb{D}_{\text{ato}}$: tuples $(\text{count}, \text{nonce}, \text{begin}, \text{end}, \text{confirmed})$ where:

1. count – number of component transactions involved of type $\mathbb{N}_{16}$
2. nonce – nonce of type $\mathbb{N}_{128}$
3. begin – block round number of type $\mathbb{N}_{64}$
4. end – block round number of type $\mathbb{N}_{64}$
5. confirmed – set of tuples (i, P', Π), where i is the index of the target of P' among $u_1, \ldots, u_m$ and Π is a proof that the transaction in the record P' was executed in the appropriate partition.

- $\mathbb{D}_{\text{fcr}} = (b, \lambda, \ell, t)$, where
  1. $b \in \mathbb{N}^{\pm}_{64}$ is the current balance of this record, in fixed point format with 8 fractional decimal digits
  2. $\lambda \in \mathbb{H}$ is the hash of the last addFC, closeFC, lockFC, or unlockFC transaction for this record
  3. $\ell \in \mathbb{N}_{64}$ is the lock status of the record; $\ell = 0$ means the record is not locked, any other value means it's locked
  4. $t \in \mathbb{N}_{64}$ is the minimum lifetime of this record

Summary functions:

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

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

Transaction types: $\mathbb{T}$ = {con, fin, lockFC, unlockFC, addFC, closeFC} (confirm a transaction, finalize a transaction, lock a fee credit record, unlock a fee credit record, add fee credit, close fee credit)

The fin transactions are never accepted from clients; these transactions can only be generated by validators as part of processing the con transactions.

6.3.2 Transactions

6.3.2.1 Confirm

Transaction order $T = \langle(\alpha_0, \text{con}, \iota, A, M_C), s, s_f\rangle$ to the atomicity partition with $A = (m, r, i, P', \Pi)$, where:

1. m – number of transactions in the multi-unit transaction of type $\mathbb{N}_{16}$
2. r – nonce of type $\mathbb{N}_{128}$
3. i – sequence number of the confirmed transaction of type $\mathbb{N}_{16}$
4. P' – transaction record of any type
5. Π – transaction proof

Transaction-specific validity condition $\psi_{\text{con}}(T, S)$: does the following checks:

1. ExtrType(ι) = ato – target unit is an atomic transaction

Table 14. Data fields of the con transaction record.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_0
2.	transaction type	τ	$\mathbb{T}$	con
3.	unit identifier	ι	$\mathbb{I}$	-
4.	transaction count	$A.m$	$\mathbb{N}_{16}$	-
5.	nonce	$A.r$	$\mathbb{N}_{128}$	-
6.	sequence number	$A.i$	$\mathbb{N}_{16}$	-
7.	transaction record	$A.P$	variable	-
8.	transaction proof	$A.\Pi$	Π^u_{prim}	-
9.	message timeout	$M_C.T_0$	$\mathbb{N}_{64}$	-
10.	maximum fee	$M_C.f_m$	$\mathbb{N}_{64}$	-
11.	fee credit record identifier	$M_C.\iota_f$	$\mathbb{I} \cup \{\bot\}$	-
10.	owner proof	s	$\{0,1\}^*$	-
11.	fee authorization proof	s_f	$\{0,1\}^* \cup \{\bot\}$	-
12.	actual fee	$M_S.f_a$	$\mathbb{N}_{64}$	-

2. $A.m \geq 2$ and $1 \leq A.i \leq A.m$
3. $A.P$ contains a locking script $\varphi_{\text{ato}}(\alpha_0, \iota, t_b, t_e, \ldots)$ with t_b and t_e such that $t_b < S.n \leq t_e$, where $S.n$ is the current round number of the atomicity partition.
4. $A.\Pi$ is a transaction proof of type Π^u_{prim} that $A.P$ is included in a block of the partition and shard of $A.P$.

Actions Action$_{\text{con}}$: for $\langle \alpha_0, \text{con}, \iota, (m, r, i, P', \Pi), M_C, s, s_f \rangle$:

1. Parse t_b and t_e from P'
2. If $N[\iota] = \bot$, then add a new unit ι with $D = (m, r, t_b, t_e, \{(i, P', \Pi)\})$
3. Else:

   3.1 If $m = N[\iota].D$.count, $r = N[\iota].D$.nonce, $t_b = N[\iota].D$.begin, and $t_e = N[\iota].D$.end then:

   3.1.1 $N[\iota].D$.confirmed $\leftarrow N[\iota].D$.confirmed $\cup \{(i, P', \Pi)\}$

   3.1.2 If confirmed contains confirmations for all m messages $P_1, \ldots, P_m$, then:

   3.1.2.1 If $\iota = f(r, \tilde{P}_1, \ldots, \tilde{P}_m)$ then add $P_{\text{fs}} = \langle \text{fin}, \iota, 1 \rangle$ to the current block.

   3.1.2.2 If $\iota \neq f(r, \tilde{P}_1, \ldots, \tilde{P}_m)$ then add $P_{\text{fs}} = \langle \text{fin}, \iota, 0 \rangle$ to the current block.

6.3.2.2 Fee Credit Handling

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

6.3.3 Round initialization and completion

6.3.3.1 Round Initialization: $\text{RInit}_{\text{ato}}$

1. Delete obsolete records:
   1.1 Find all such records:
   $\mathcal{I} \leftarrow \{\iota : \text{ExtrType}(\iota) = \text{ato} \ \wedge \ N[\iota] \neq \perp \ \wedge \ N[\iota].D.\text{end} < S.n\}$
   1.2 For each $\iota \in \mathcal{I}$: DelItem(ι)

6.3.3.2 Round Completion: $\text{RCompl}_{\text{ato}}$

1. Finalize expired transactions:
   1.1 Find all such transactions:
   $\mathcal{I} \leftarrow \{\iota : \text{ExtrType}(\iota) = \text{ato} \ \wedge \ N[\iota] \neq \perp \ \wedge \ N[\iota].D.\text{end} = S.n \ \wedge \ |N[\iota].D.\text{confirmed}| < N[\iota].D.\text{count}\}$

7 User-Defined Token Partition Type

7.1 Motivation and General Description

hierarchical type system

7.2 Specification

7.2.1 General Parameters

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

Unit data $\mathbb{D}_u$ depends on the unit type 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;
  - $\iota_p \in \mathbb{I}$ identifies the parent type that this type derives from; $\iota_p = 0_{\mathbb{I}}$ indicates there is no parent type;
  - $\varphi_{\text{s}} \in \mathbb{L}$ is the predicate (subtype clause) that controls defining new subtypes of this type;
  - $\varphi_{\text{t}} \in \mathbb{L}$ is the predicate (mint clause) that controls creating new tokens of this type;
  - $\varphi_{\text{i}} \in \mathbb{L}$ is the invariant predicate (inherit bearer clause) that all tokens of this type (and of subtypes of this type) inherit into their bearer predicates;
  - $\varphi_{\text{d}} \in \mathbb{L}$ is the (inherit data clause) that all tokens of this type (and of subtypes of this type) inherit into their data update predicates;
- $\mathbb{D}_{\text{ntoken}} = (\iota_t, nam, uri, dat, \varphi_{\text{d}}, t, \lambda, \ell)$, 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;
  - $\varphi_{\mathsf{d}} \in \mathbb{L}$ is the data update clause, which is the predicate that controls the update to the data field;
  - $t \in \mathbb{N}_{64}$ is the current partition round number of the last transaction with this token;
  - $\lambda \in \mathbb{H}$ is the hash of the last transaction order for 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;
- $\mathbb{D}_{\mathsf{ftype}} = (sym, nam, ico, \iota_p, dec, \varphi_{\mathsf{s}}, \varphi_{\mathsf{t}}, \varphi_{\mathsf{i}})$, 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;
  - $\iota_p \in \mathbb{I}$ identifies the parent type that this type derives from; $\iota_p = 0_{\mathbb{I}}$ indicates there is no parent type;
  - $dec \in 0 \ldots 8$ is the number of decimal places to display for values of tokens of this type;
  - $\varphi_{\mathsf{s}} \in \mathbb{L}$ is the predicate (subtype clause) that controls defining new subtypes of this type;
  - $\varphi_{\mathsf{t}} \in \mathbb{L}$ is the predicate (mint clause) that controls creating new tokens of this type;
  - $\varphi_{\mathsf{i}} \in \mathbb{L}$ is the invariant predicate (inherit bearer clause) that all tokens of this type (and of subtypes of this type) inherit into their bearer predicates;
- $\mathbb{D}_{\mathsf{ftoken}} = (\iota_t, v, t, \lambda, \ell)$, where
  - $\iota_t \in \mathbb{I}$ is the type of this token;
  - $v \in \mathbb{N}_{64}$ is the value of this token;
  - $t \in \mathbb{N}_{64}$ is the current partition round number of the last transaction with this token;

  - $\lambda \in \mathbb{H}$ is the hash of the last transaction order for 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;
- $\mathbb{D}_{\text{fcr}} = (b, \lambda, \ell, t)$, where
  - $b \in \mathbb{N}^{\pm}_{64}$ is the current balance of this record, in fixed point format with 8 fractional decimal digits;
  - $\lambda \in \mathbb{H}$ is the hash of the last addFC, closeFC, lockFC, or unlockFC transaction for 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;
  - $t \in \mathbb{N}_{64}$ is the minimum lifetime of this record.

Summary functions:

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

Transaction types: $\mathbb{T}$ = {createFType, createNType, createFToken, createNToken, transFToken, transNToken, lockToken, unlockToken, splitFToken, burnFToken, joinFToken, updateNToken, lockFC, addFC, closeFC} (create a fungible/non-fungible token type, create 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, lock a fee credit record, add fee credit, close fee credit)

7.2.1.1 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 $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ on the multipart input $s = (s_1, s_2, \ldots, s_m)$ as follows:

$$\pi(s) = (n = m) \wedge \pi_1(s_1) \wedge \pi_2(s_2) \wedge \ldots \wedge \pi_n(s_n).$$

7.2.2 Create a Fungible Token Type

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

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

- $\iota_p \in \mathbb{I}$ identifies the parent type that the new type derives from;
- $dec \in 0 \ldots 8$ is the number of decimal places to display for values of tokens of the new type;
- $\varphi_\mathsf{s} \in \mathbb{L}$ is the predicate (subtype clause) that controls defining subtypes of the new type;
- $\varphi_\mathsf{t} \in \mathbb{L}$ is the predicate (mint clause) that controls creating tokens of this type;
- $\varphi_\mathsf{i} \in \mathbb{L}$ is the invariant predicate (inherit bearer clause) that all tokens of the new type (and of subtypes of it) inherit into their bearer predicates;
- $s \in \mathbb{L}^*$ is the input to satisfy the subtype clause.

Validity Condition

$$\begin{aligned}
&\Psi_{\mathsf{createFType}}(P, S) \equiv \\
&\quad \mathsf{ExtrType}(P.\iota) = \mathsf{ftype} \ \wedge \ S.N[P.\iota] = \bot \ \wedge \\
&\quad (P.A.\iota_p = 0_\mathbb{I} \ \vee \ \mathsf{ExtrType}(P.A.\iota_p) = \mathsf{ftype} \ \wedge \ S.N[P.A.\iota_p] \neq \bot) \ \wedge \\
&\quad (P.A.\iota_p = 0_\mathbb{I} \ \vee \ S.N[P.A.\iota_p].D.dec = P.A.dec) \ \wedge \\
&\quad \Pi(P.A.\iota_p)(P.A.s) = 1,
\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_\mathbb{I} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_\mathsf{s} & \text{otherwise.} \end{cases}$$

That is,

- ι 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,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the subtype clauses along the type inheritance chain.

Actions

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

7.2.3 Create a Non-Fungible Token Type

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

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

- $\iota_p \in \mathbb{I}$ identifies the parent type that the new type derives from;
- $\varphi_\mathsf{s} \in \mathbb{L}$ is the predicate (subtype clause) that controls defining subtypes of the new type;
- $\varphi_\mathsf{t} \in \mathbb{L}$ is the predicate (mint clause) that controls creating tokens of the new type;
- $\varphi_\mathsf{i} \in \mathbb{L}$ is the invariant predicate (inherit bearer clause) that all tokens of the new type (and of subtypes of it) inherit into their bearer predicates;
- $\varphi_\mathsf{d} \in \mathbb{L}$ is the predicate (inherit data clause) that all tokens of the new type (and of subtypes of this type) inherit into their data update predicates;
- $s \in \mathbb{L}^*$ is the input to satisfy the subtype clause.

Validity Condition

$$\begin{aligned}&\Psi_{\mathsf{createNType}}(P, S) \equiv \\ &\quad \mathsf{ExtrType}(P.\iota) = \mathsf{ntype} \ \wedge \ S.N[P.\iota] = \bot \ \wedge \\ &\quad (P.A.\iota_p = 0_\mathbb{I} \ \vee \ \mathsf{ExtrType}(P.A.\iota_p) = \mathsf{ntype} \ \wedge \ S.N[P.A.\iota_p] \neq \bot) \ \wedge \\ &\quad \Pi(P.A.\iota_p)(P.A.s) = 1,\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_\mathbb{I} \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_\mathsf{s} & \text{otherwise.} \end{cases}$$

That is,

- ι 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,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the subtype clauses along the type inheritance chain.

Actions

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

7.2.4 Create a Fungible Token

Transaction order $P = \langle(\alpha, \mathsf{createFToken}, \iota, A, M_C), s, s_f\rangle$, $A = (\varphi, \iota_t, v, s)$, where

- $\varphi \in \mathbb{L}$ is the initial bearer predicate of the new token;
- $\iota_t \in \mathbb{I}$ identifies the type of the new token;
- $v \in \mathbb{N}_{64}$ is the value of the new token;
- $s \in \mathbb{L}^*$ is the input to satisfy the mint clause.

Validity Condition

$$\begin{aligned}&\Psi_{\text{createFToken}}(P, S) \equiv \\ &\quad \text{ExtrType}(P.\iota) = \text{ftoken} \ \wedge \ S.N[P.\iota] = \bot \ \wedge \\ &\quad \text{ExtrType}(P.A.\iota_t) = \text{ftype} \ \wedge \ S.N[P.A.\iota_t] \neq \bot \ \wedge \\ &\quad P.A.v > 0 \ \wedge \\ &\quad \Pi(P.A.\iota_t)(P.A.s) = 1,\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_{\text{t}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies a fungible token that does not yet exist,
- the type of the new token is an existing fungible token type,
- the new token has non-zero value,
- the input s given in the transaction order satisfies the multipart predicate obtained by joining all the mint clauses along the type inheritance chain.

Actions

1. AddItem$(P.\iota, P.A.\varphi, (P.A.\iota_t, P.A.v, S.n, H(P), 0))$

7.2.5 Create a Non-Fungible Token

Transaction order $P = \langle(\alpha, \text{createNToken}, \iota, A, M_C), s, s_f\rangle$, $A = (\varphi, \iota_t, nam, uri, dat, \varphi_{\text{d}}, s)$, where

- $\varphi \in \mathbb{L}$ is the initial bearer predicate of the new token;
- $\iota_t \in \mathbb{I}$ identifies the type of the new token;
- nam is the optional name of the new token;
- uri is the optional URI of an external resource associated with the new token;
- dat is the optional data associated with the new token;
- $\varphi_{\text{d}} \in \mathbb{L}$ is the data update clause of the new token;
- $s \in \mathbb{L}^*$ is the input to satisfy the mint clause.

Validity Condition

$$\begin{aligned}&\Psi_{\text{createNToken}}(P, S) \equiv \\ &\quad \text{ExtrType}(P.\iota) = \text{ntoken} \ \wedge \ S.N[P.\iota] = \bot \ \wedge \\ &\quad \text{ExtrType}(P.A.\iota_t) = \text{ntype} \ \wedge \ S.N[P.A.\iota_t] \neq \bot \ \wedge \\ &\quad \Pi(P.A.\iota_t)(P.A.s) = 1,\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_{\text{t}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies a non-fungible token that does not yet exist,
- the type of the new token is an existing non-fungible token type,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the mint clauses along the type inheritance chain.

Actions

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

7.2.6 Transfer a Fungible Token

Transaction order $P = \langle(\alpha, \text{transFToken}, \iota, A, M_C), s, s_f\rangle$, $A = (\varphi, v, \iota_t, \eta, \lambda, s)$, where

- $\varphi \in \mathbb{L}$ is the new bearer predicate of the token;
- $v \in \mathbb{N}_{64}$ is the value to transfer;
- $\iota_t \in \mathbb{I}$ identifies the type of the token;
- $\eta \in \{0,1\}^* \cup \{\bot\}$ is an optional nonce;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with this token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

$$\begin{aligned}
&\Psi_{\text{transFToken}}(P, S) \equiv \\
&\quad \text{ExtrType}(P.\iota) = \text{ftoken} \ \wedge S.N[P.\iota] \neq \bot \ \wedge \\
&\quad S.N[P.\iota].\ell = 0 \ \wedge \\
&\quad S.N[P.\iota].D.v = P.A.v \ \wedge \\
&\quad S.N[P.\iota].D.\iota_t = P.A.\iota_t \ \wedge \\
&\quad S.N[P.\iota].D.\lambda = P.A.\lambda \ \wedge \\
&\quad \Pi(S.N[P.\iota].D.\iota_t)(s) = 1,
\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\mathsf{i}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies an existing fungible token,
- the token is not locked,
- the value transferred is the value of the token,
- the token type in the transaction order matches the actual token type,
- the current transaction follows the previous valid transaction with the token,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. SetOwner($P.\iota, P.A.\varphi$)
2. UpdateData(ι, f), where $f(D) = (D.\iota_t, D.v, S.n, H(P), D.\ell)$

7.2.7 Transfer a Non-Fungible Token

Transaction order $P = \langle(\alpha, \text{transNToken}, \iota, A, M_C), s, s_f\langle, A = (\varphi, \iota_t, \eta, \lambda, s)$, where

- $\varphi \in \mathbb{L}$ is the new bearer predicate of the token;
- $\iota_t \in \mathbb{I}$ identifies the type of the token;
- $\eta \in \{0,1\}^* \cup \{\bot\}$ is an optional nonce;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with the token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

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

where

$$
\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\text{i}} & \text{otherwise.} \end{cases}
$$

That is,

- ι identifies an existing non-fungible token,
- the token is not locked,
- the token type in the transaction order matches the actual token type,
- the current transaction follows the previous valid transaction with the token,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. SetOwner($P.\iota, P.A.\varphi$)
2. UpdateData(ι, f), where $f(D) = (D.\iota_t, D.nam, D.uri, D.dat, D.\varphi_{\text{d}}, S.n, H(P), D.\ell)$

7.2.8 Lock a Token

Transaction order $P = \langle(\alpha, \text{lockToken}, \iota, A, M_C), s, s_f\rangle$, $A = (\ell, \lambda, s)$, where

- $\ell \in \mathbb{N}_{64}$ is the new lock status of the token;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with this token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

$$
\begin{aligned}
&\Psi_{\text{lockToken}}(P, S) \equiv \\
&\quad (\text{ExtrType}(P.\iota) = \text{ftoken} \ \vee \ \text{ExtrType}(P.\iota) = \text{ntoken}) \ \wedge \ S.N[P.\iota] \neq \bot \ \wedge \\
&\quad S.N[P.\iota].\ell = 0 \ \wedge \\
&\quad P.A.\ell > 0 \ \wedge \\
&\quad S.N[P.\iota].D.\lambda = P.A.\lambda \ \wedge \\
&\quad \Pi(S.N[P.\iota].D.\iota_t)(s) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\text{i}} & \text{otherwise.} \end{cases}
$$

That is,

- ι identifies an existing fungible or non-fungible token,
- the token is not locked,
- the new status is a “locked” one,
- the current transaction follows the previous valid transaction with the token,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. UpdateData(ι, f), where

$$
f(D) = \begin{cases} (D.\iota_t, D.v, S.n, H(P), P.A.\ell) & \text{if ExtrType}(\iota) = \text{ftoken} \\ (D.\iota_t, D.nam, D.uri, D.dat, D.\varphi_{\text{d}}, S.n, H(P), P.A.\ell) & \text{if ExtrType}(\iota) = \text{ntoken} \end{cases}
$$

7.2.9 Unlock a Token

Transaction order $P = \langle(\alpha, \text{unlockToken}, \iota, A, M_C), s, s_f\rangle$, $A = (\lambda, s)$, where

- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with this token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

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

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\mathsf{i}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies an existing fungible or non-fungible token,
- the token is locked,
- the current transaction follows the previous valid transaction with the token,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. UpdateData(ι, f), where

$$f(D) = \begin{cases} (D.\iota_t, D.v, S.n, H(P), 0) & \text{if } \mathsf{ExtrType}(\iota) = \mathsf{ftoken} \\ (D.\iota_t, D.nam, D.uri, D.dat, D.\varphi_{\mathsf{d}}, S.n, H(P), 0) & \text{if } \mathsf{ExtrType}(\iota) = \mathsf{ntoken} \end{cases}$$

7.2.10 Split a Fungible Token

Transaction order $P = \langle(\alpha, \mathsf{splitFToken}, \iota, A, M_C), s, s_f\rangle$, $A = (\varphi, v, v', \iota_t, \eta, \lambda, s)$, where

- $\varphi \in \mathbb{L}$ is the bearer predicate of the new token;
- $v \in \mathbb{N}_{64}$ is the amount to transfer;
- $v' \in \mathbb{N}_{64}$ is the remaining value of source token;
- $\iota_t \in \mathbb{I}$ identifies the type of the token;
- $\eta \in$ $bitstr \cup \{\bot\}$ is an optional nonce;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with this token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

$$\begin{aligned}&\Psi_{\text{splitFToken}}(P, S) \equiv \\ &\quad \text{ExtrType}(P.\iota) = \text{ftoken} \ \wedge S.N[P.\iota] \neq \bot \ \wedge \\ &\quad S.N[P.\iota].\ell = 0 \ \wedge \\ &\quad S.N[P.\iota].D.v = P.A.v + P.A.v' \ \wedge \\ &\quad P.A.v > 0 \ \wedge \ P.A.v' > 0 \ \wedge \\ &\quad S.N[P.\iota].D.\iota_t = P.A.\iota_t \ \wedge \\ &\quad S.N[P.\iota].D.\lambda = P.A.\lambda \ \wedge \\ &\quad \Pi(S.N[P.\iota].D.\iota_t)(s) = 1,\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_{\mathbb{i}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies an existing fungible token,
- the token is not locked,
- the initial value of the source token is equal to the transferred value plus the remaining value,
- the value to be transferred and the remaining value are both non-zero,
- the token type in the transaction order matches the actual token type,
- the current transaction follows the previous valid transaction with the source token,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. $\iota' \leftarrow \text{NodeID}(\text{ftoken}, \text{PrndSh}(\text{ExtrUnit}(P.\iota), P.\iota || P.A || P.M_C))$
2. $\text{AddItem}(\iota', P.A.\varphi, (P.A.D.\iota_t, P.A.v, S.n, H(P), 0))$
3. $\text{UpdateData}(P.\iota, f)$, where $f(D) = (D.\iota_t, D.v - P.A.v, S.n, H(P), D.\ell)$

Targets

For splitFToken transaction P, $\text{targets}(P) = \{P.\iota, \iota'\}$.

7.2.11 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:

- 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 lockToken transaction.

- The source tokens are "burned" (deleted) using burnFToken transactions. To prevent replay attacks, each of the burnFToken transactions must identify the selected target token and its current state.
- The value of the source tokens is added to the target token using a joinFToken transaction. As this transaction completes the joining process, it also unlocks the target token.

7.2.11.1 Burning Step

Transaction order $P = \langle(\alpha, \text{burnFToken}, \iota, A, M_C), s, s_f\rangle$, $A = (v, \iota_t, \iota', \lambda', \lambda, s)$, where

- $v \in \mathbb{N}_{64}$ is the value to burn; note that for the proof of burn to be usable in a following joinFToken operation, the resulting value of the target token must not overflow $\mathbb{N}_{64}$;
- $\iota_t \in \mathbb{I}$ identifies the type of the token to burn;
- $\iota' \in \mathbb{I}$ is the identifier of the target token;
- $\lambda' \in \mathbb{H}$ is the current state hash of the target token;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with this token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

$$
\begin{aligned}
&\Psi_{\text{burnFToken}}(P, S) \equiv \\
&\quad \text{ExtrType}(P.\iota) = \text{ftoken} \ \wedge S.N[P.\iota] \neq \bot \wedge \\
&\quad S.N[P.\iota].\ell = 0 \wedge \\
&\quad S.N[P.\iota].D.v = P.A.v \wedge \\
&\quad S.N[P.\iota].D.\iota_t = P.A.\iota_t \wedge \\
&\quad S.N[P.\iota].D.\lambda = P.A.\lambda \wedge \\
&\quad \Pi(S.N[P.\iota].D.\iota_t)(s) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\text{i}} & \text{otherwise.} \end{cases}
$$

That is,

- ι identifies an existing fungible token,
- the token is not locked,
- the value to be burned is the value of the token,
- the type of token to burn matches the actual type of the token,
- the current transaction follows the previous valid transaction with the token,
- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. UpdateData(ι, f), where $f(D) = (D.\iota_t, 0, S.n, H(P), D.\ell)$

7.2.11.2 Joining Step

Transaction order $P = \langle(\alpha, \text{joinFToken}, \iota, A, M_C), s, s_f\rangle$, $A = (P_1, \ldots, P_m, \Pi_1, \ldots, \Pi_m, \lambda, s)$, where

- $P_1, \ldots, P_m$ are the transactions that burned the source tokens;
- $\Pi_1, \ldots, \Pi_m \in \Pi^u_{\text{prim}}$ are the transaction proofs of $P_1, \ldots, P_m$;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with the target token;
- $s \in \mathbb{L}^*$ is the input to satisfy the inherited bearer clause.

Validity Condition

$$
\begin{aligned}
&\Psi_{\text{joinFToken}}(P, S) \equiv \\
&\quad \text{ExtrType}(P.\iota) = \text{ftoken} \ \wedge S.N[P.\iota] \neq \bot \wedge \\
&\quad P.A.P_1.\alpha = \ldots = P.A.P_m.\alpha = \alpha_{\text{user}} \wedge \\
&\quad P.A.P_1.\tau = \ldots = P.A.P_m.\tau = \text{burnFToken} \wedge \\
&\quad P.A.P_1.\iota < \ldots < P.A.P_m.\iota \wedge \\
&\quad P.A.P_1.A.\iota_t = \ldots = P.A.P_m.A.\iota_t = S.N[P.\iota].D.\iota_t \wedge \\
&\quad P.A.P_1.A.\iota' = \ldots = P.A.P_m.A.\iota' = P.\iota \wedge \\
&\quad P.A.P_1.A.\lambda' = \ldots = P.A.P_m.A.\lambda' = P.A.\lambda \wedge \\
&\quad \text{VerifyTxProof}(P.A.\Pi_1, P.A.P_1, S.\mathcal{T}, S.\mathcal{SD}) \wedge \\
&\quad \ldots \wedge \\
&\quad \text{VerifyTxProof}(P.A.\Pi_m, P.A.P_m, S.\mathcal{T}, S.\mathcal{SD}) \wedge \\
&\quad S.N[P.\iota].D.v + P.A.P_1.A.v + \ldots + P.A.P_m.A.v < 2^{64} \wedge \\
&\quad S.N[P.\iota].D.\lambda = P.A.\lambda \wedge \\
&\quad \Pi(S.N[P.\iota].D.\iota_t)(s) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} 1 & \text{if } 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\mathbb{i}} & \text{otherwise.} \end{cases}
$$

That is,

- ι identifies an existing fungible token,
- the transactions $P_1, \ldots, P_m$ were in this system,
- the transactions $P_1, \ldots, P_m$ were burning transactions,
- burning transactions orders are listed in strictly increasing order of token identifiers (in particular, this ensures that no source token can be included multiple times),
- 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 backlinks,
- the burning transactions were valid transactions,
- the value of the joined token would not overflow $\mathbb{N}_{64}$,
- the current transaction follows the previous valid transaction with the token,

- the input s given in the transaction request satisfies the multipart predicate obtained by joining all the inherited bearer clauses along the type inheritance chain.

Actions

1. UpdateData(ι, f), where
   $f(D) = (D.\iota_t, D.v + P.A.P_1.A.v + \ldots + P.A.P_m.A.v, S.n, H(P), 0)$

7.2.12 Update a Non-Fungible Token

Transaction order $P = \langle(\alpha, \text{updateNToken}, \iota, A, M_C), s, s_f\rangle$, $A = (dat, \lambda, s)$, where

- dat is the new data to replace the data currently associated with the token;
- $\lambda \in \mathbb{H}$ is the backlink to the previous transaction with the token;
- $s \in \mathbb{L}^*$ is the input to satisfy the token's data update clause.

Validity Condition

$$\begin{aligned}&\Psi_{\text{updateNToken}}(P, S) \equiv\\ &\quad \text{ExtrType}(P.\iota) = \text{ntoken} \ \wedge S.N[P.\iota] \neq \bot \wedge\\ &\quad S.N[P.\iota].\ell = 0 \wedge\\ &\quad S.N[P.\iota].D.\lambda = P.A.\lambda \wedge\\ &\quad \Pi(S.N[P.\iota])(S.N[P.\iota].D.dat, P.A.dat, s) = 1,\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} 1 & \text{if } \iota = 0_{\mathbb{I}} \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_{\mathsf{d}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies an existing non-fungible token,
- the token is not locked,
- the current transaction follows the previous valid transaction with the token,
- the data currently associated with the token, the new data to be associated with the token, and the input s given in the transaction request satisfy the multipart predicate obtained by joining all the token data update clauses along the type inheritance chain.

Actions

1. UpdateData(ι, f), where $f(D) = (D.\iota_t, D.nam, D.uri, P.A.dat, D.\varphi_{\mathsf{d}}, S.n, H(P), D.\ell)$

7.2.13 Fee Credit Handling

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

7.2.14 Round initialization and completion

7.2.14.1 Round Initialization: $\mathsf{RInit}_{\mathsf{user}}$

- **Zero Token Deletion**
  1. For each $\iota \in \{\iota : \mathsf{ExtrType}(\iota) = \mathsf{ftoken} \ \wedge \ N[\iota] \neq \perp \ \wedge \ N[\iota].D.v = 0\}$:
     1.1 $\mathsf{DelItem}(\iota)$

7.2.14.2 Round Completion: $\mathsf{RCompl}_{\mathsf{user}}$

No transaction system specific completion steps.

8 Alphabill Distributed Machine

8.1 Background

8.1.1 Definitions

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

UC is *Unicity Certificate*.

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

All partition validators and Root Chain 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.

A partition's validators are synchronized based on input from the Root Chain. There are some fixed time-outs.

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

Within a partition/shard there are k validators with identifier v, of which f might be faulty. For the Root Chain $k > 3f$. For partition α, $k_\alpha > 2f$. We assume that all faulty validators are 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 partition and 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). *Payment* is a special case of transaction.

8.1.2 Vocabulary

Transaction System is a set of rules and logic for defining *units* and performing *transactions* with them.

Partition implements an instance of a specific *Transaction System*. Partition is a distributed system, it consists of either a set of shards, or one or more partition validators.

Shard is a decomposition of a partition; all shards within a partition implement the same transaction system, thus these shards are compatible. Every shard is a distributed system: it consists of one or more shard validators.

Root Chain (also known as the *Root Partition*) implements a function within the Alphabill Framework which guarantees uniqueness of the states of its partitions: assigns specific transaction system logic to these partitions, performs validation and correctness checks and ensures consensus of individual validators. Root Chain generates Unicity Certificates for identified partitions.

8.1.3 Scope

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

8.1.4 Repeating Notation

n_r – round number of the Root Chain
n_α – round number of transaction system α
k_α – number of validators in partition α
v – validator identifier, unique within a partition; set of partition's validators is $\{v_i\}_{i\in\{1,\ldots,k_\alpha\}}$
$v_l \leftarrow$ LEADERFUNC$(\cdot)$ – leader for this block production attempt
h – state tree root hash
h' – previous state tree root hash

$ensure(\ldots)$ – function modelled 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.

8.2 Partitions and Shards

8.2.1 Timing

A Partition is synchronized using Input Records in returned UCs. For a partition, a UC can have the following options:

1. IR has not changed. Our partition can ignore this UC.
2. UC certifies an input from our partition, 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 partition, then this 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 UC request to the Root Chain. See Figure 12.

Root Chain has a timer `t2` for every partition; it is reset when a valid UC for this partition 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 a partition. New round is executed with a different leader. Nodes can determine which UC is latest based on round number. A block proposal generated by the leader is accomplished with a UC and this UC must point to this leader. Block is finalized when UC is embedded. The retry mechanism is illustrated by Fig. 13.

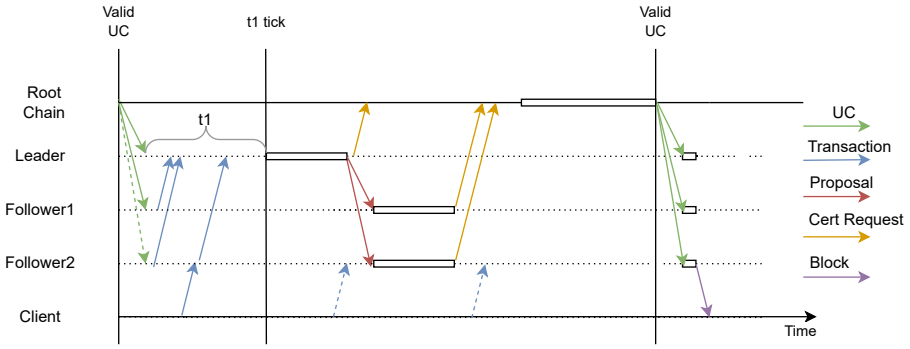

Figure 12. Successful Partition Round

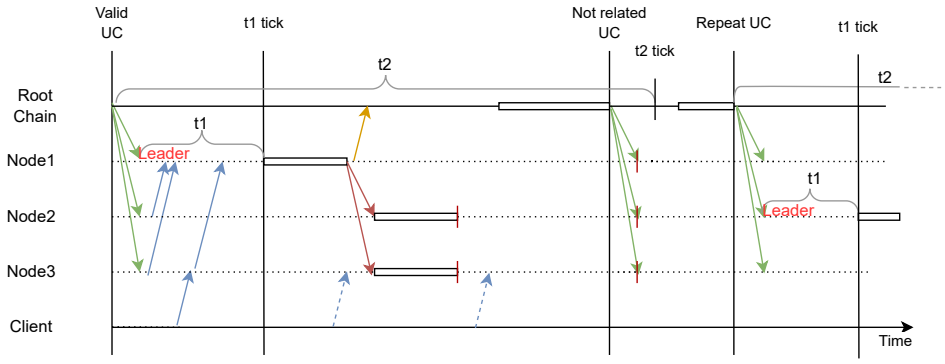

Figure 13. Partition Round attempt which did not produce a valid UC for this partition

8.2.2 Configuration and State

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

- System Identifier (α), synonymous with *partition id*.

- Shard Identifier (σ); present for multi-shard partitions.
- Validator Identifier (v), unique within a partition/shard. There are k_α validators in partition α.
- 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 Chain
- public keys of other validators within the shard/partition
- communication addresses of other validators within the shard/partition
- communication addresses of the Root Chain validators

Data layer:

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

Variables:

- v_l: Current round leader's indentifier; 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 *round number* and the *state hash* of the last certified state, in a conveniently signed and versioned data structure.
- 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 UC 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.

Algorithm 1 State and Initialization

Constants:
α: Transaction System (Partition) Identifier
σ: Shard Identifier
v: Node Identifier
k_α: number of validators in the partition α
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
▷ Implicitly, <*round number to be certified*> = $luc.IR.n + 1$
▷ Implicitly, <*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 BlockCertification request waiting for UC
$\mathcal{B}$: blockchain

function START_NEW_ROUND(uc)
  ensure($uc.IR.n > luc.IR.n$)
  ensure($uc.IR.h' = luc.IR.h$) ▷ Double checking
  cp $\leftarrow$ CHECKPOINT(N)
  RINIT()
  $log \leftarrow \{\}$
  $pr \leftarrow \{\}$
  $v_l \leftarrow$ LEADERFUNC(uc)
  $luc \leftarrow uc$
  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 $\{v_i \mid i \leftarrow \text{integer}(H(uc)) \bmod k_\alpha + 1\}$ ▷ Simplest example
end function

2. *After applying any transaction(s)*: There are changes in the state tree; executed transactions are recorded in $propsoal$ (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 BlockProposal message (if leader) and sending UCReq 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 UCMsg*:
   - *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.
   - *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.

8.2.3 Subcomponents

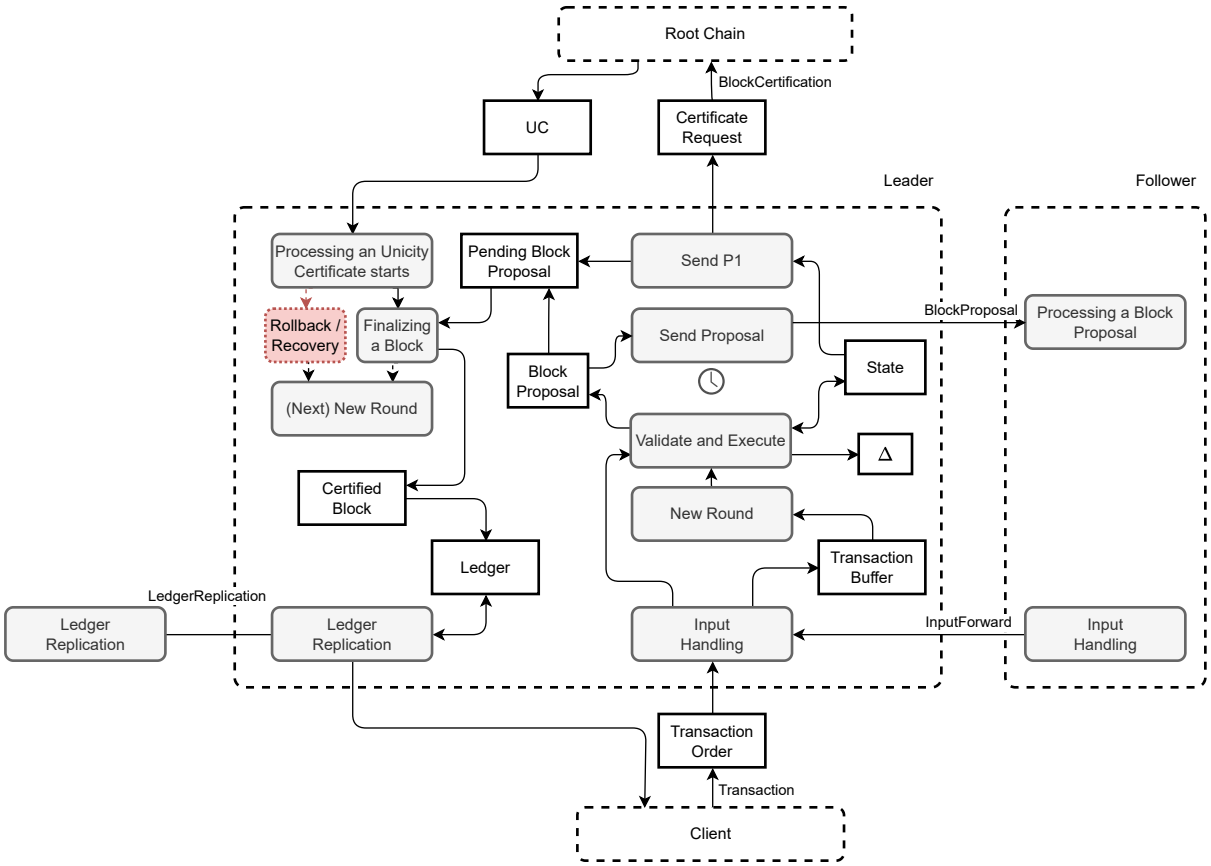

Figure 14. Data Flow of Partition Leader Node

8.2.3.1 Input Handling

Input Handling prioritizes latency (fast finality). It is optimized for the case where there is enough processing capacity and blockspace 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 partition/shard; transaction orders sent to wrong partition will be discarded. Optionally, implement QoS / overload protection. There is no quarantee 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.

Partition validators forward transaction orders, as they arrive, to respective shard/partition leaders (next block proposal producers); while observing time-outs and discard 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.

8.2.3.2 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 UCReq 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.

Algorithm 2 Input Handling

```
upon message <TransactionMsg | P> do
    if SANITY_CHECK(P) then
        if ν_l = ν then                                   ▷ this process is the leader
            PROCESS({P})                                  ▷ Beware of parallel execution
        else if ν_l ≠ NULL then                           ▷ someone else is the leader
            if ¬SEND_INPUTFORWARDMSG(ν_l, P) then         ▷ ν_l is the recipient
                buf ← buf ∪ P                             ▷ Store on failure
            end if
        else                                              ▷ Buffer transactions until leader is known
            buf ← buf ∪ P
        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
        send_BlockProposalMsg(α, ν, luc, log)                          ▷ Sign and Broadcast
        do_cert_req(log, ν)
    end if
    ν_l ← Null                          ▷ leader does not accept new transaction orders
end on
function do_cert_req(txs, l)
    h' ← luc.IR.h, n ← luc.IR.n + 1, e ← luc.IR.e
    h ← StateRoot(N), V ← DataSummary(N)
    pr ← (n, e, h', h, ν_l, log)                                   ▷ Pending Block Proposal
    b ← (α, σ, luc.IR.h_B, ν_l, txs, Null)                                    ▷ temporary
    h_B ← block_hash(b)
    if store_in_durable_way(pr) then
        send_UCReq(α, ν, (n, e, h', h, V, h_B))                            ▷ Sign and send
    else
                    ▷ Another validator may submit UCReq and this round gets finalized
    end if
end function
```

8.2.3.3 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.

The expiration of Transaction Orders is checked relative to partition/shard round number of the UC the block proposal is trying to extend, incremented by one ($luc.IR.n + 1$).

Refer to Algorithm 4 for details. Note that fee processing is omitted for brevity.

8.2.3.4 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. System Identifier is checked.
2. The time-stamp in UC is checked for sanity: it must not “jump around” and it must

Algorithm 4 Validate and Execute Transaction Orders

```
function SANITY_CHECK(P; t ← luc.IR.n + 1)        ▷ Transactions will be fully validated later
    return P.α = α ∧ P.σ = σ ∧ P.T_0 > t
end function

function VALIDATE(P; t ← luc.IR.n + 1)                              ▷ default value of t
                                          ▷ see Platform Specification, Valid Transaction Orders
end function

function PROCESS(txs)                          ▷ Should be implemented as processing queue
    for all P ∈ txs do
        if VALIDATE(P) then
            EXECUTE(N, P)                  ▷ Can be executed in parallel if non-related units
            log ← log ∪ P                                  ▷ Retaining the ordering of input
        end if
    end for
end function
```

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 partition/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:

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

On failing any of these checks, an equivocation proof must be logged with all necessary evidence.

5. UC must have newer round number than the last one seen.
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.

Algorithm 5 Processing a received Unicity Certificate

```
upon message <UCMsg | (uc)> do
    if GOT_NEW_UC(uc) then
        START_NEW_ROUND(uc)
    end if
end upon
function GOT_NEW_UC(uc)
    ensure(uc ≠ luc)                                          ▷ ignore the same UC
    ensure(VALID(uc))                                         ▷ 'ensure()' returns 0 on failure
    ensure(uc.C^uni.α = α)
    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 UC request
        if uc.IR.h ≠ STATEROOT(N) then
            RECOVERY(uc)
        end if
    else
        if uc.IR.h = pr.h ∧ uc.IR.h' = pr.h' ∧ uc.IR.e = pr.e ∧ 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. 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 5 for details. Block Finalization is presented in Algorithm 7.

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

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 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($uc.C^{\mathsf{r}}.n_r > luc.C^{\mathsf{r}}.n_r$)
  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**
      ▷ non-empty block hash can only repeat in repeat UC
      ensure($uc.IR.h = uc'.IR.h \wedge uc.IR.h' = uc'.IR.h' \wedge uc.IR.V = uc'.IR.V$)
    end if
  end if
  if $uc.IR.n = uc'.IR.n + 1$ **then**
    if $\neg(uc.IR.h = uc'.IR.h \wedge uc.IR.h' = uc'.IR.h' \wedge uc.IR.V = uc'.IR.V)$ **then**
      ensure($uc.IR.h' = uc'.IR.h$) ▷ if not a repeat UC
    end if
  end if
  return 1
end function

Algorithm 7 Finalizing a Block

function FINALIZE_BLOCK(pr, uc)
  $B \leftarrow (\alpha, \sigma, luc.IR.h_B, pr.l, pr.txs, uc)$
  ensure(BLOCK_HASH(B) $= uc.IR.h_B$)
  $\mathcal{B} \leftarrow \mathcal{B} \cup B$ ▷ Adding a new block to shard's blockchain
    ▷ If a block can not be retained and made available then the round must be not closed
end function

8.2.3.5 Processing a Block Proposal

Summary: Upon receiving a BlockProposalMsg message, validate the signature and header fields, execute transaction orders from the proposal, updating State Tree (N) and Rollback Buffer (cp). Executed transactions go into Block Proposal. Create and send Uniqueness Certificate message, producing a Pending Block Proposal data structure.

This procedure is performed by the non-leader validators. There are following steps (See Fig. 15):

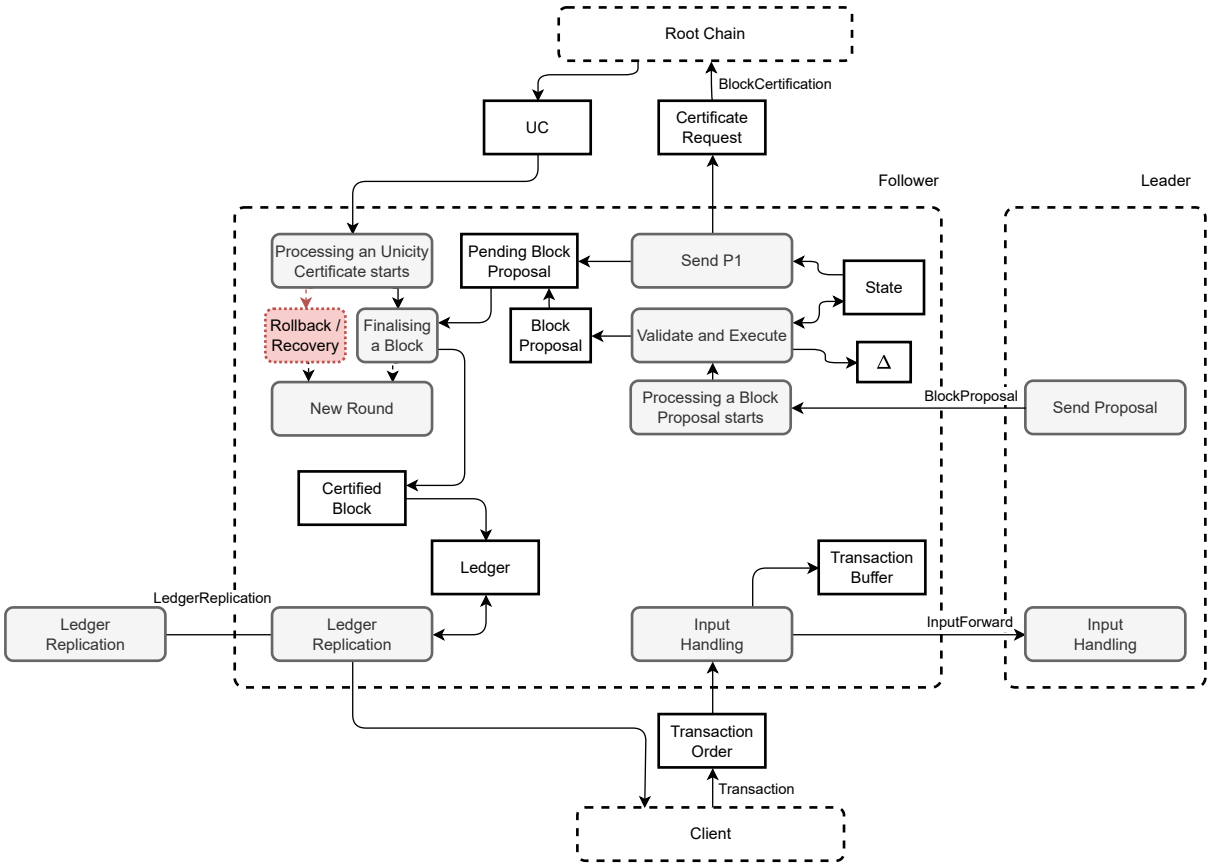

Figure 15. Data Flow of a non-leader partition validator

1. Block proposal as a whole is validated. It must have valid signature, correct transaction system ID, 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.
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.
4. All transaction orders in proposal are validated; on encountering an invalid transaction order the processing is aborted.
5. Transaction orders are executed by applying them to the state tree.
6. Pending certificate request data structure is created and persisted.
7. Certificate Request query (UCReq) is assembled and sent to the Root Chain.

Please refer to Algorithm 8 for details.

8.2.3.6 Ledger Replication

Relatively independent subsystem for serving and replicating ledger data. Pseudocode of the service is provided in Algorithm 9.

Algorithm 8 Processing a received Block Proposal

upon message <BlockProposalMsg | $m = (\alpha, \nu, uc, txs; s)$> **do**
  ensure(VALID(m))
  ensure($m.\nu \neq \nu \wedge m.\alpha = \alpha$)
  ensure(VALID(uc))
  ensure($m.uc.IR.n \geq luc.IR.n$)
  ensure($m.\nu =$ LEADERFUNC(uc))
  if $m.uc.IR.n > luc.IR.n$ **then**
    ensure(GOT_NEW_UC($m.uc$)) ▷ newer UC must be validated and processed
    if *processing of new UC took too much time (recovering?)* **then**
      return START_NEW_ROUND($m.uc$)
    else
      $luc \leftarrow m.uc$
    end if
  end if
  $h' \leftarrow m.uc.IR.h$
  ensure(STATEROOT(N) $= h'$)
  ensure({ $\forall\ P \in m.txs$ | VALIDATE(P)})
  cp $\leftarrow$ CHECKPOINT(N)
  RINIT()
  for all $P \in m.txs$ **do**
    EXECUTE(N, P)
  end for
  RCOMPL()
  DO_CERT_REQ($m.txs, m.\nu$)
end upon

Algorithm 9 Ledger Replication

upon message <LedgerReplication | $n_1, n_2 \leftarrow luc.IR.n$> **do**
  return $\{\mathcal{B}_i \mid \mathcal{B}_i.UC.IR.n \in [n_1 \,..\, n_2]\}$
end upon

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

8.2.4 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 partition, whose authoritative source is the Root Chain.

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.

In more details:

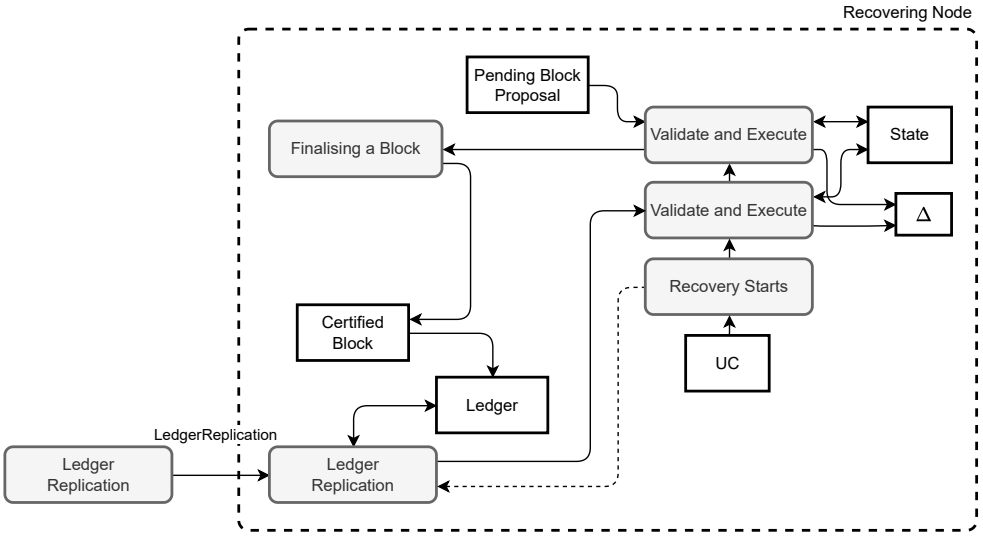

Figure 16. Data Flow of an out-of-sync (recovering) Partition Node

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 transaction system ID;
4. Each transaction within block is verified,
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 10 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.

8.2.5 Epoch Change

Partition Epoch Change is triggered by the Root Chain, by incrementing the Epoch number field in Input Record.

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

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

Algorithm 10 Partition Node Recovery

```
function RECOVERY(uc)                                   ▷ Assuming that Revert() is done by caller
    ensure(VALID(uc))
    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)
            ensure(b.UC.IR.n > luc.IR.n)
            ensure(STATEROOT(N) = luc.IR.h = b.IR.h′)
            ensure({ ∀ P ∈ b.txs | VALIDATE(P, luc.IR.n + 1)})
            cp ← CHECKPOINT(N)
            RINIT()
            for all P ∈ b.txs do
                EXECUTE(N, P)
            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
        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({ ∀ P ∈ pr.txs | VALIDATE(P, pr.n)})
            cp ← CHECKPOINT(N)
            RINIT()
            for all P ∈ pr.txs do
                EXECUTE(N, P)
            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
```

8.2.6 Protocols – Partition Nodes

8.2.6.1 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: $\langle \texttt{TransactionMsg} \mid P \rangle$

8.2.6.2 Protocol UCReq – Block Certification

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

If h' is unknown to Root Chain then the latest UC is returned immediately via UCMsg.

Returned UC can be 'repeat' UC for h' which triggers next attempt using a different leader. A partition validator can have multiple pending requests extending the same hash; latest one is identified using greater n_r, which must come from a valid UC. Partition validators must not create multiple requests with the same n_r.

This message "subscribes" the validator to receive UC messages for a certain period, either a fixed number (2) or more precisely until the partition have successfully proposed a following block, that is, there is another set of Root Chain validators which have received a quorum of UCreq messages and therefore "taken over" the subscription.

Message: $\langle \texttt{UCReq} \mid \alpha, v, IR; s \rangle$

If a partition/shard validator has reason to suspect that Root Chain have generated a new UC then he must try to fetch it by retrying.

8.2.6.3 Protocol UCMsg – returning of UC

Asynchronous response (in the sense of data flow) to UCReq; there may be many UCMsg responses to one client-sent message, either UCReq or Ping.

Message: $\langle UC \rangle$

8.2.6.4 Protocol Subscription – subscribing to UCMsg messages

This message "subscribes" the validator to future UCMsg messages, without presenting a UC Request in UCReq message. Synchronous response is the latest UC for requestor.

Subscription ends when the partition have successfully proposed a following block, that is, there is another set of Root Chain validators which have received a quorum of UCreq messages and therefore serving a subscription.

Query: $\langle \texttt{SubscriptionMsg} \mid \alpha, v, s \rangle$

Response: $\langle UC \rangle$

8.2.6.5 Protocol InputForwardMsg – Input Forwarding

Forward a set of transaction orders.

Message: $\langle$`InputForwardMsg` | $\{P\}\rangle$

8.2.6.6 Protocol BlockProposalMsg – Block Proposal

Leader broadcasts its block proposal to other partition validators.

Message: $\langle$`BlockProposal` | $\alpha, \nu_l, uc, txs; s\rangle$
where $txs = \{P\}$

8.2.6.7 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 partition 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: $\langle$`LedgerReplication` | $(\alpha, n_1, [n_2]\rangle$

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.

8.3 Root Chain

8.3.1 Summary

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

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

8.3.2 Timing

Root Chain serves many partitions/shards, each implemented by a cluster of parallel validator machines. Operation cycles work like this:

1. Partitions operate in parallel:
   - Partition validators send BlockCertification requests.
   - Once there is a quorum of requests for transaction system α, the Root Chain updates an entry in the array of input records.
2. *Eventually* Root Chain 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 partition wait `t1`.
System parameters are: average root Certificate generation time t_r; average partition round processing time from receiving UC to sending UC request, including `t1` wait: t_p; with standard deviation σ_p. There are k partitions.

Let m – how many Root Chain rounds fit into one partition 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. 17):
Start Root Chain 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.

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

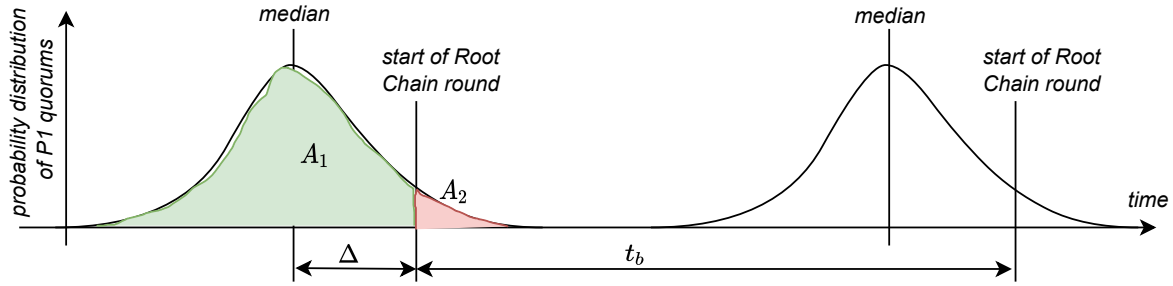

Figure 17. Optimal average latency is achieved when $\frac{A_1}{A_2} \approx \frac{t_b}{\Delta}$, where A represents the respective area (number of messages). $t_b - \Delta \approx t_r + t_p$.

8.3.3 State

System configuration of every Root Chain validator:

- $\mathcal{T}$ – Unicity Trust Base
- $\mathcal{A}[]$ – System Identifiers
- $\mathcal{SD}[]$ – System Description Records
- $k[]$ – Partition/shard cluster sizes for every system identifier

Communication layer:

- Node secret key used to sign messages
- related public key; known to other Root Chain validators via configuration management/governance and to other parties via $\mathcal{T}$
- public keys of other Root Chain validators (used by the underlying communication layer)
- public keys of Partition Validators (usage captured by the opaque function VALID())
- communication addresses of other Root Chain validators
- $conn[][]$ – Connection contexts to return responses to validators which sent a BlockCertification request, variable

State, must be persisted:

- n – Round Number (referred as n_r outside the Root Chain context)
- e – Epoch Number (or e_r if the context is not clear)
- r_- – Previous round's Unicity Tree root hash
- $\mathcal{IR}[]$ – Certified Input Records for every partition

Variables

- $req[][]$ – BlockCertification requests, indexed by system id and validator id
- $changes[]$ – Validated Changes to IR, applied before UC generation

Time-outs and timers:

- `t2` – Root Chain waits for `t2` time units before re-issuing a UC for a partition/shard. This triggers a retry of partition/shard block creation, with another leader. There is one timer instance per partition, referred as $\texttt{t2}_\alpha$ for partition α.
- `t3` – Triggers Unicity Tree re-calculation and UC response generation (monolithic Root Chain only); Target Block Rate (t_b) – Distributed Root Chain specific parameter

8.3.4 Analysis

8.3.4.1 Safety

Root Chain enforces that each block is 'extended' by one block only. This excludes conflicting blocks (forks). Root Chain must not generate "equivocating" UCs (see Section 8.2.3.4 and Algorithm 6).

8.3.4.2 Liveness

Usual properties of partially synchronous communication model apply.

8.3.4.3 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 Chain have issued a repeat UC, then all BlockCertification requests must refer to this UC. Arriving and pending BlockCertification requests, referring to older UC, 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, initiated by later repeat UC.
3. All BlockCertification requests extending the current state are considered 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/partition 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.

8.4 Monolithic Implementation

This section (Algorithms 11, 12, 13, 14) defines a monolithic, non-distributed Root Chain implementation. It serves distributed partitions/shards. Functionally it is equivalent to distributed Root Chain (see Section 8.5).

Algorithm 11 Global Parameters and Variables

Configuration:
$\mathcal{T}$: Unicity Trust Base
$\mathcal{A}[]$: System Identifiers
$\mathcal{SD}[]$: System Descriptions
$k[]$: Cluster sizes of partitions
State:
$n \leftarrow 0$: Round number
$e \leftarrow 0$: Epoch number
$r_{-} \leftarrow$ NULL: Previous round's Unicity Tree root hash
$\mathcal{IR}[] \leftarrow \{\perp\}$: Input Records
Variables:
$changes[] \leftarrow \{\}$ Changes to IR, applied at the end of round, indexed by system id
$req[][] \leftarrow \{\}$: BlockCertification requests, indexed by system id and validator id
$conn[][] \leftarrow \{\}$: Partition/shard validator connections
Start Timers:
RESET_TIMER(t3)
for $a \in \mathcal{A}$ **do**
  RESET_TIMER($t2_a$)
end for

8.4.1 UCReq Request Processing

UCReq Request Processing starts with checking if a newer UC is available; if yes then it is returned immediately. This would initiate partition/shard recovery if necessary, and start a new partition 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 partition/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 partition/shard ID. If it is clear that a partition/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/partition consensus attempt.

See Algorithm 12.

8.4.2 Unicity Certificate Generation

Periodically, do the following:

1. In case a partition/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'.

Algorithm 12 UCReq Message Handling.

upon message <UCReq | $(\alpha, \nu, IR; s); context$> **do**
  ensure(VALID$((\alpha, \nu, IR; s))$)
  $conn[\alpha][\nu] \leftarrow context$ ▷ Network connection for returning messages
  if $IR.n \neq \mathcal{IR}[\alpha].n + 1$ **then** ▷ Round is behind/ahead
    SEND_UCMSG$(context,$ GET_UC$(\alpha))$
    return ▷ Stop processing
  else if $IR.h' \neq \mathcal{IR}[\alpha].h$ **then**
    SEND_UCMSG$(context,$ GET_UC$(\alpha))$ ▷ Extending of unknown state
    return ▷ Stop processing
  end if
  if $req[\alpha][\nu]$ **then** ▷ Reject duplicate request
    return
  end if
  ensure(TX-SYSTEM-SPECIFIC-CHECKS$(\mathcal{SD}[\alpha], IR))$
  $req[\alpha][\nu] \leftarrow IR$ ▷ Add new message
  $c \leftarrow |\{\ r \in req[\alpha] \mid r = IR\ \}|$ ▷ count of matching votes
  if $c > k[\alpha]/2$ **then** ▷ Consensus
    $changes[\alpha] \leftarrow IR$
  else if $k[\alpha] - |req[\alpha]| + c < k[\alpha]/2$ **then** ▷ Consensus not possible
    $(n, e, h', h, V, h_B) \leftarrow \mathcal{IR}[\alpha]$
    $changes[\alpha] \leftarrow (n + 1, e, h', h, V, h_B)$ ▷ Produce 'repeat UC'
  end if
end upon

Algorithm 13 Obtaining an individual UC for a Transaction System

function GET_UC(a) ▷ Assuming Unicity Tree and Unicity Seal are generated
  return $(\mathcal{IR}[a],$ CREATEUNICITYTREECERT$(a, \chi, \mathcal{SD}), C^r)$
end function

2. The pending changes in $changes$ array are applied to IR.
3. Based on the array of Input Records, build the Unicity Tree. Extract root hash value. Obtain wall clock time. Create the Unicity Certificate.
4. For every record in IR changes array, respond to shard/partition based on request context from the request buffer. Then, clean up req buffer, and reset respective `t2` timer.
5. Finally, reset $changes$, update the previous root hash used for linking[1], increment Root Chain round number and reset the timer `t3` which triggers the Unicity Certificate Generation (Algorithm 14).

8.5 Distributed Implementation

Distributed Root Chain is bisimilar to monolithic Root Chain in the sense that they provide the same service implementing the same business logic. Distributed Root Chain is

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

Algorithm 14 Unicity Certificate Generation

on event t3 **do** ▷ Simplified, see section "Timing"
  for $a \in \mathcal{A}$ **do** ▷ Process transaction system timeouts
    if $\neg changes[a] \wedge$ EXPIRED_TIMER($\mathtt{t2}_a$) **then**
      $(n, e, h', h, V, h_B) \leftarrow \mathcal{IR}[a]$
      $changes[a] \leftarrow (n+1, e, h', h, V, h_B)$ ▷ Produce 'repeat UC'
    end if
  end for
  for $(a, ir) \in changes$ **do** ▷ Apply changes
    ensure($\mathcal{IR}[a].n + 1 = ir.n$) ▷ Double-checking
    ensure($\mathcal{IR}[a].h = ir.h'$) ▷ Double-checking
    ensure($\mathcal{IR}[a].e \leq ir.e$)
    $\mathcal{IR}[a] \leftarrow ir$
  end for
  $\chi \leftarrow$ CreateUnicityTree($\mathcal{A}, \mathcal{SD}, \mathcal{IR}$)
  $r \leftarrow \chi(\lfloor\rfloor)$
  $t \leftarrow$ GetUTCDateTime()
  $C^r \leftarrow$ CreateUnicitySeal(n, t, r_-, r; sk_r)
  for all $a \in changes$ **do**
    $req[a] \leftarrow []$
    $uc \leftarrow$ GET_UC(a)
    for all $connection \in conn[a]$ **do** ▷ respond to all nodes in a cluster, in parallel
      SEND_UCMsg($connection, uc$) ▷ subscriptions expire - see protocol desc.
    end for
    RESET_TIMER($\mathtt{t2}_a$)
  end for
  $changes \leftarrow \{\}$
  $r_- \leftarrow r$
  $n \leftarrow n + 1$
  RESET_TIMER(t3)
end on

Algorithm 15 "SubscriptionMsg" message handling

upon message <SubscriptionMsg | $(\alpha, \nu; s)$; $context$> **do**
  ensure(VALID($(\alpha, \nu; s)$))
  $conn[\alpha][\nu] \leftarrow context$ ▷ Network connection for returning messages
  return GET_UC(α)
end upon

Byzantine fault tolerant. It uses the SMR (State Machine Replication) concept.

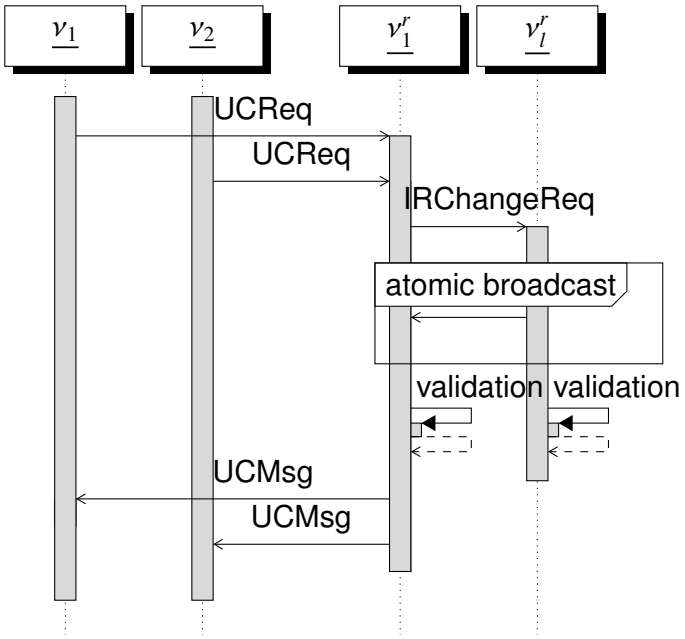

Figure 18. Message flow (simplified)

The messaging between a partition and the Root Chain is illustrated by Figure 18, where a partition with two validators v_1 and v_2 requests a UC. Root chain is also depicted with two validators, the next leader after reaching a quorum of partition requests is v_l^r.

8.5.1 Summary of Execution

The summary follows the processing flow of a UC request by a Root Chain validator. The flow is illustrated in Fig. 19; loosely moving counter-clockwise.

8.5.1.1 Node Selection

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

If a partition validator haven’t received a UC-s from chosen Root Chain validators within $2 \times \mathtt{t2}$ then it sends SubscriptionMsg requests to random other Root Chain validators. This ensures eventual synchronization if the firstly chosen validators happen to be faulty.

8.5.1.2 UCReq Validation

No difference from Monolithic Implementation, please refer to Algorithm 12 for details. If available, or on invalid request, UC is returned immediately by the same Root Chain validator.

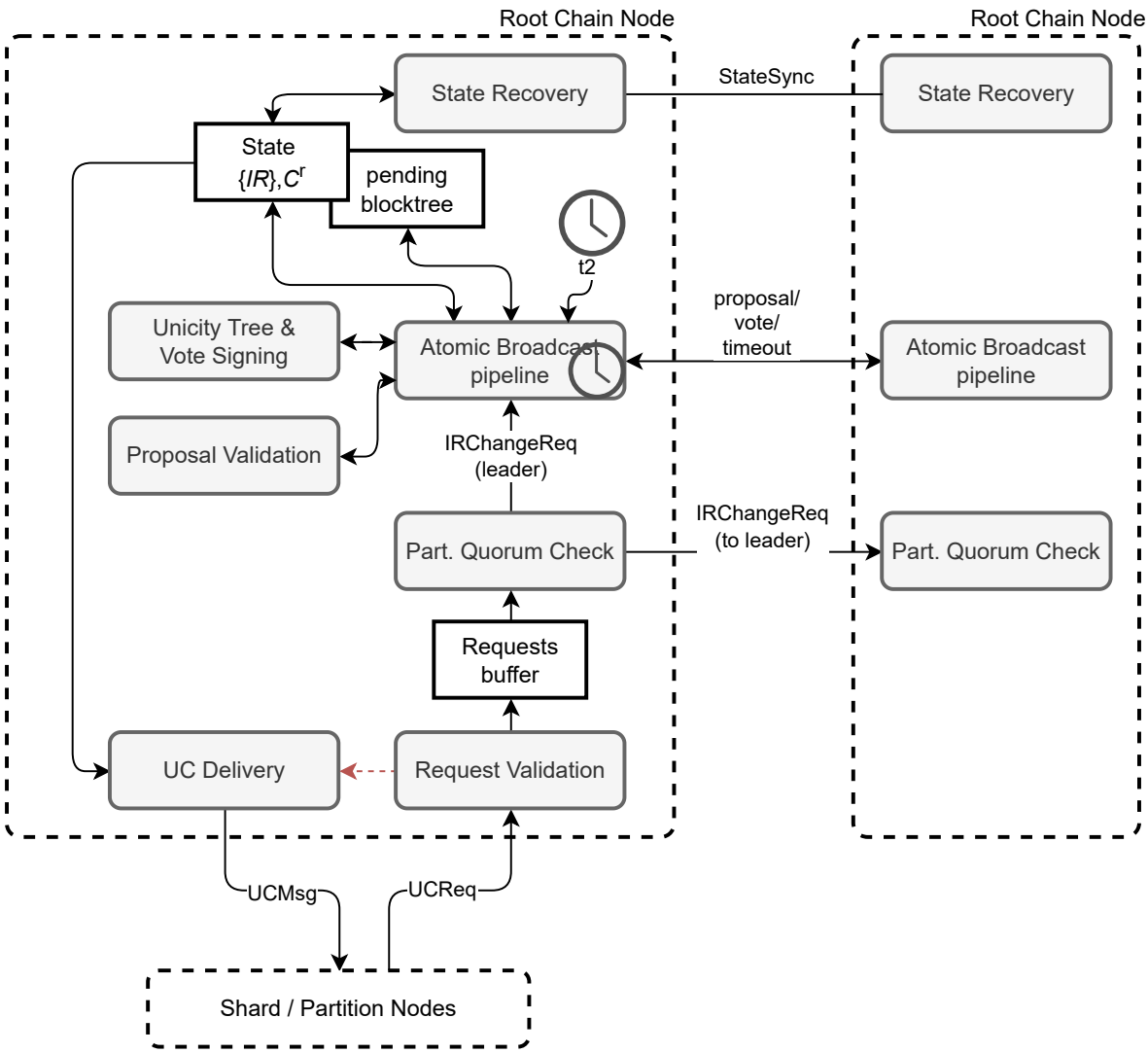

Figure 19. Data-flow of a Distributed Root Chain Node

8.5.1.3 Partition Quorum Check

Partition/shard Quorum Check is the same as on Monolithic case (Alg. 12). If a quorum is achieved or considered impossible, then a message is assembled (IRChangeReq), which includes all UCReq messages, and forwarded to the next Root Chain leader, using the Atomic Broadcast submodule.

8.5.1.4 IR Change Request Validation

The request must be validated analogously to the Algorithm 8.

8.5.1.5 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 partition 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 partition.

Due to the pipelined 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 Chain validators.

8.5.1.6 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 8. Summary of the cases:

Quorum achieved The justification must prove the achievement of consensus by a partition. More than half of the partition 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 8.5.2.1).

8.5.1.7 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.

8.5.1.8 UC Generation

If a leader updates the last UC array element then it returns UCMsg responses to all pending shard/partition 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).

8.5.2 Proposal

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

- Change requests with justifications.
- Repeat UC request where consensus is considered impossible.
- Repeat UC request on `t2` timeout of a partition.

8.5.2.1 State Synchronization

A Root Chain validator must have up-to-date vector of Input Records of all partitions/shards. There is no persistent block storage. Returned Unicity Certificates, certifying IRs, are possibly persisted within partition 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.

8.5.3 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 partitions where they could be saved as part of partition or shard blocks. Node implementation is encouraged to produce an audit log with all the block proposal payloads.

Blockchain: There is no such thing as the Root Chain Blockchain. However, Unicity Trust Base is somewhat blockchain-like: it gets a new entry added once per Root Chain epoch, and similarly to block headers it can be interpreted as the Root of Trust.

8.5.3.1 Round Pipeline

Committing the payload happens across many rounds due to the pipelined nature of Hot-Stuff. 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.

8.5.3.2 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 Chain 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.

8.5.3.3 Leader Election

Initially, a round-robin selection algorithm is used. In the roadmap there is DPoS 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.

8.5.3.4 Root Chain Epoch Change

In order to facilitate a dynamic, responsive Root Chain, 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 potentially dynamic requirements and the operating environment.

The configuration can be changed once per epoch. The source information comes from a Governance Process whose output is Validator Assignment Records (Table 17).

Any Root Chain protocol leader can initiate an epoch change, given it has received the Governance Decision. The procedure works as follows:

1. The leader produces a proposal where the usual payload is replaced by Epoch Change Request (Table 15);
2. Validators who approve the epoch change continue with usual flow and do not include usual payload until the Epoch Change Request is committed.
3. Next round after committing an Epoch Change Request is the first round of this epoch: Epoch in BlockData is incremented; and execution continues by the updated set of validators.

Table 15. Root Chain Epoch Change Request.

No	Field	Notation	Type
1.	Epoch number	e	$\mathbb{N}_{64}$
2.	Validator identifiers and stakes	$\{v, b_v\}_e$	$\{(\{0,1\}^*, \mathbb{N}_{64})\}$
3.	Quorum size (Voting power)	k_e	$\mathbb{N}_{16}$
4.	Hash of state summary	r	$\mathbb{H}$
5.	Hash of governance decision	h_{gov}	$\mathbb{H}$
6.	Hash of previous record	h_{e-1}	$\mathbb{H}$
7.	(attached) Governance Decision		(Table 18)

Fields 1, 2, 3 are copied from the Governance Decision. Governance Decision 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 chnnels for distributing Governance Decisions.

Root Chain'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 8.5.6.7); and the new record gets propagated to Partition validators together with the next Unicity Certificate, as an extra field of the UCResp message.

Rule. Root Chain 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 Governance Decisions. Instead, the latest valid one must be used, whenever available.

If there are multiple Governance Decisions, then the latest valid one must be executed.

8.5.4 Controlling Partition Epochs

Root Chain triggers partition epoch changes. This happens in following steps:

1. RC validators obtain Governance Decisions for the next epoch of a partition.
2. If an RC Leader includes an Input Record Change Request of a partition into a proposal and there is a pending epoch change of this partition, then the governance decision is included to the Request.
3. Presence of Governance Decision is the signal that Epoch should be incremented. Included governance decision is validted, and if valid then epoch number in IR is incremented. All other IR changes are validated and applied as well.
4. IR-s get certified.
5. New UC is returned to the partition.
6. Partition's configuration is updated: quorum size (voting power needed for consensus), list of validators. This changes RC's validation rules: the next UC Request must be presented by a valid quorum of next epoch's partition validators.

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

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

8.5.5 Data Structures

Below is an informal illustration on how Alphabill data structures integrate to HotStuff consensus primitive; in ABNF format[2]. The structures document the distributed, dynamic Root Chain.

```
; defined above
  UC =  IR UnicityTreeCertificate UnicitySeal
  IR = PreviousHash Hash BlockHash SummaryValue RoundNumber SumOfEarnedFees
  UnicityTreeCertificate = SystemIdentifier *SiblingHash SystemDescriptionHash

; interface:
  UnicitySeal = RootChainRoundNumber Epoch Timestamp PreviousHash Hash *Signatures ;
      C^r = (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:
  VoteMsg = VoteInfo LedgerCommitInfo HighQC Author Signature
```

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

```
  ProposalMsg = BlockData [LastRoundTc] Signature
  BlockData   = Author Round Epoch Timestamp Payload AncestorQC
  Payload     = *IRChangeReq | RCEpochChangeReq
  IRChangeReq = SystemIdentifier CertReason *BlockCertificationRequest
      [EpochChangeJustification] SenderSignature  ; presence of justification ==>
      epoch++
  RCEpochChangeReq = Epoch *(NodeID Pubkey Stake) QuorumThreshold StateHash GovDecisionHash
      PreviousEntryHash SenderSignature [GovDecision]

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

  TimeoutMsg = Timeout Author Signature
  Timeout = Epoch Round HighQC LastTC
              ; HighQC - highest known QC of the validator
              ; LastTC - if HighQC is not from prev.  round then there must be TC of prev.  round
  TC = Timeout *Signatures  ; 2f+1 Signatures

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

; partition-rc messages
  BlockCertificationRequest = SystemIdentifier NodeIdentifier IR  RootRoundNumber Signature
  ; 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

  BlockCertificationResponse  = UC [RootTrustBaseEntry] ; may be sent without corresponding request

; 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 = SystemIdentifier IR Sdrh

  ; Subscription - Subscribe to BlockCertificationResponse messages, while obtaining the latest UC for
      synchronization
  SubscriptionMsg = systemIdentifier nodeIdentifier signature ;
        ; this request provides or updates validator connection parameters at
        ; transport layer so that Root can return UCMsg messages
  SubscriptionResp = UC    ; response:  latest UC for the system, returned synchronously
```

8.5.6 Root Chain Implementation Specific Shared Data Structures

8.5.6.1 Versioning

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

8.5.6.2 Evolving

Unicity Trust Base itself evolves (e.g., new records are added, while format/version stays the same) when the Root Chain 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.

8.5.6.3 Monolithic, Static Root Chain

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

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

function VerifyUnicitySeal($r, C^{\text{r}}, \mathcal{T}$)
  return ($r = C^{\text{r}}.r \wedge \text{Ver}_{\mathcal{T}.\text{pk}}(C^{\text{r}}, C^{\text{r}}.s)$) ▹ 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

8.5.6.4 Monolithic, Dynamic Root Chain

In the case of dynamic Root Chain 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.

- $T_e = (e, \text{pk}_e; s)$
  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 Chain

[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*.

- $\mathcal{T} = (T_j, T_{j+1}, \ldots, T_k)$,
  where j is the first epoch and k is the latest epoch supported
- $C^{\mathsf{r}} = (n_r, t_r, r_-, r; (s, e))$
  as defined in the Platform Specification; s is a cryptographic signature created using Root Chain's secret key of the epoch e.

function VerifyUnicitySeal($r, C^{\mathsf{r}}, \mathcal{T}$)
  if $r \neq C^{\mathsf{r}}.r$ **then**
    return 0
  end if
  $T = \mathcal{T}.T_{C^{\mathsf{r}}.e}$
  if $T = \perp$ **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, $\exists\, i\colon\ \mathcal{T}.T_i.e = C^{\mathsf{r}}.e$. Obtaining a fresh Unicity Trust Base is covered by the Chapter Alphabill Anterior.

function VerifyUnicityTrustBase($\mathcal{T}_{\mathsf{base}}, \mathcal{T}$)
  $T = \mathcal{T}_{\mathsf{base}}.T_{|\mathcal{T}_{\mathsf{base}}|}$ ▷ last trusted record
  if $T.e < \mathcal{T}.T_1.e - 1$ **then**
    return error ▷ not a continuous chain
  end if
  if $\mathsf{Ver}_{T.\mathsf{pk}}(\mathcal{T}.T_{T.e+1}, \mathcal{T}.T_{T.e+1}.s) = 0$ **then**
    return 0
  end if
  for $e \in \{T.e + 2\,..\,\mathcal{T}.T_{|\mathcal{T}|}.e\}$ **do**
    if $\mathsf{Ver}_{\mathcal{T}.T_{e-1}.\mathsf{pk}}(\mathcal{T}.T_e, \mathcal{T}.T_e.s) = 0$ **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.

8.5.6.5 Distributed, static Root Chain

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

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

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

function VerifyUnicitySeal($r, C^{\mathsf{r}}, \mathcal{T}$)
  if $r \neq C^{\mathsf{r}}.r$ **then**
    return 0
  end if
  if $|C^{\mathsf{r}}.s| \neq \mathcal{T}.q$ **then** ▷ All signatures of distinct validators
    return 0 ▷ Wrong number of signatures
  end if
  for $(i, s) \in C^{\mathsf{r}}.s$ **do**
    if $(\mathsf{Ver}_{\mathcal{T}.\mathsf{pk}_i}(C^{\mathsf{r}}, s) = 0)$ **then** ▷ calculated over all fields except signature
      return 0 ▷ Invalid signature
    end if
  end for
  return 1 ▷ Success
end function

8.5.6.6 Distributed Root Chain, aggregated signatures

This is an optimization, reducing the sizes of produced proofs and the trust base. The underlying primitive implements 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 Chain 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.

8.5.6.7 Distributed, Dynamic Root Chain

The Unicity Trust Base Record is defined by Table 16.

Fields 1, 3 and 4 are copied from the referenced Governance Decision. Initially, the stakes are fixed to 1. When appropriate (delegated) Proof of Stake governance processes are implemented, the stake reflects the epoch's locked stake amounts of a particular validator.

Unicity Trust Base is a chain of records defined by Table 16.

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

Table 16. Unicity Trust Base Record of Dynamic Distributed Root Chain.

No	Field	Notation	Type
1.	Epoch number	e	$\mathbb{N}_{64}$
2.	Epoch starting round	r_e	$\mathbb{N}_{64}$
3.	Validator identifiers and stakes	$\{v, b_v\}_e$	$\{(\{0,1\}^*, \mathbb{N}_{64})\}$
4.	Quorum size (voting power)	k_e	$\mathbb{N}_{16}$
5.	Hash of state summary	r	$\mathbb{H}$
6.	Hash of related governance decision	h_{gov}	$\mathbb{H}$
7.	Hash of previous record	h_{e-1}	$\mathbb{H}$
8.	Signature of previous epoch validators	s_{e-1}	*version-dependent*

Unicity Seal is a record signed by the validator set of the respective epoch as defined in 8.5.6.4, with an additional requirement of using the required multi-party signature scheme.

The verification functions are as defined in 8.5.6.4, where the signatures are interpreted in broader sense as multi-party signatures, created by respective quorums of validators.

8.5.7 Alphabill as a Decentralized and Permission-less Blockchain

The sections so far document a blockchain system which must be bootstrapped and managed by a trusted entity. This can be set up as a trusted Foundation, which employs a team of system administrations, who change the configuration of the system based on Foundation's requests. For example, if a validator wants to join the Alphabill platform, the Foundation 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 section is about implementing Alphabill as a truly decentralized and permission-less system. It relies on delegated proof of stake mechanism and on-chain governance.

The launch as a permission-less, PoS controlled, decentralized blockchain is a fragile affair, due to initial instability (number of validators, locked stake, 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, on-chain governance processes, briefly described in the following sections, are replacing the inintially more manual ("permissioned") processes.

8.5.7.1 Proof of Stake Mechanisms

Validator stakes are not fixed to 1, but correspond to the actual staked amount. The stake amount is calculated by a governance process.

Votes in consensus algorithms have weights; and required quorum size is defined as the sum of stakes of agreeing validators necessary to reach consensus (required voting power).

The leader election algorithm may use stake amounts as an input (Section 8.5.7.2).

8.5.7.2 On-chain Governance

On-chain Governance is executed by the Governance Partition. There are following, independent processes. Subsections describe the interface between Alphabill Platform and the Governance (regardless on-chain or by the Foundation): expectations on processes and their output data structures.

Validator Assignment The process produces records (called Governance Decisions) assigning validators to specific partitions and shards, as shown in Table 17.

Table 17. Governance Decision: Validator Assignment Record

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

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

The exact content of Epoch Switching Condition is left open: it can be 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 may be a Root Chain round number range with soft enforcement – the responsible validators will not receive any fees for their work outside the expected epoch.

Quorum Size is measured in total amount of stake behind validator votes to reach consensus. For the Root Chain, $k > 2/3 \sum b_v$. For partitions / shards, $k > 1/2 \sum b_v$.

Partition Management This process creates, modifies and deletes partitions. The output is System Description Records and consensus-layer configuration, illustrated by Table 18.

Table 18. Governance Decision: Partition Change Record

No	Field	Notation	Type
1.	System Identifier	α	$\mathbb{A}$
2.	System Description	$\mathcal{SD}$	
3.	Cluster Size *)	k	$\mathbb{N}_{32}$
4.	Target Block Rate	`t`	$\mathbb{N}_{32}$(ms)
5.	Time-out *)	`t2` / `t3`	$\mathbb{N}_{32}$(ms)
6.	Hash of previous record	h_{e-1}	$\mathbb{H}$

The values marked by *) may be represented as a *suggested range* and/or a *required range*, where the precise value is chosen by per-epoch management process, based on operational requirements.

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 19.

Table 19. Governance Decision: Sharding Scheme Update

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

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 behaviour: instability, not following the ledger rules, not following the 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 expulse 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.

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.

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 Chain round number when the switch happens.

On-chain Democrary In order to approve arbitrary changes expressable 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 ecosystem.

9 Client Integration Patterns, Interfaces and Tools

9.1 Background

9.1.1 Definitions

Wallet is a component which produces Transaction Orders. In order to do so, it must have access to signing capability: the access to a private key or keys. Usually a "wallet" knows the list of units it "owns" and can spend. There might be other functionality packaged with a wallet—for example the capability to verify transactions. Wallet may be the central component executing and orchestrating Composite Transactions, involving possibly many shards and transaction systems.

If this definition feels controversial, then please mentally replace every occurrence of "wallet" with "keychain".

Full Node is a component which receives and validates new transactions of a partition/shard, either as new blocks or block proposals, and builds a state tree keeping track of all units managed by the partition/shard.

State Tree is a data structure for maintaining the state of units managed by a partition/shard. State tree is authenticated by Unicity Certificates.

RPC Node is a middleware service which provides services to blockchain clients, like generating Ledger Proofs. RPC Node must maintain an up-to-date state tree for every partition/shard it can generate these proofs for.

Ledger Proof is a compact proof certifying the state of a unit. It is extracted from State Tree.

Block Proof is a compact proof certifying the execution of a transaction order. It is extracted from the block where this transaction is recorded.

Composite Transaction is a related series of simple transactions, where a proof certifying the execution of previous transaction can be a pre-condition for executing the following transaction. Alphabill supports serial composition of transactions.

9.1.2 Creating Transactions

Transaction Order is a data structure used to instruct Alphabill to execute a transaction with a Unit; in some contexts it is also a message. Transaction Orders are recorded in ledger blocks as proofs of transaction execution. Transaction Order is accomplished with "owner proof", a data field which "unlocks" specified unit. Unit "owner" is anybody who can produce a valid owner proof. The most common example of ownership proof is a digital signature

signing the transaction order. Necessary private key is handled by a software or hardware component called “wallet”.

Transaction order messages are delivered from user to designated shard/partition nodes. The message may traverse intermediaries like RPC Nodes; but because the messages are signed and not further processed by middleware, these layers can be ignored (while not considering availability and censorship issues). Actual delivery mechanism is application specific and out of scope in this document.

Wallet can be managed by client device so that secret keys are under sole and full control of the client. Alternatively, keys can be managed by a third party; we call this architecture as *custodial wallet* (Fig. 25).

9.1.3 Verifying Transactions

The state of a unit is defined as the last certified state, as managed by designated shard/-partition. After certifying a state the transaction is included into a block, which is immediately released. Thus, block proves the state at certification time.

Successfully executed transaction order changes the state of a unit. In usual transaction, there are the sender and the recipient of transaction; considering effects the recipient is “relying party”, a party which first verifies a transaction and then makes a decision based on result, i.e., it relies on the correct validation of transaction execution. If the recipient is the sole owner of the transaction, then he can be sure that no-one else can produce valid transaction orders; and he can trust that he is still owner of the unit based on last execution proof. Sender of the transaction is usually interested in timeliness of execution and recipient’s verification, in order to e.g. receive something in exchange.

Following subsections list possible ways to verify execution of transactions.

9.1.3.1 Ledger Proof Based

Ledger Proof (denoted as Π) certifies unit’s state and last executed transaction order $(P; s)$. Ledger proof is verified by function call

$$\text{VerifyLedgerProof}(\Pi, (P; s), \mathcal{T}, \text{SD}) \ ;$$

where $\mathcal{T}$ is the root of trust, *unicity trust base*, and SD is system description defining the partition.

Unit’s state D is valid if $\Pi.y = H(\iota, \varphi, D)$. Unit state can be verified only based on ledger proof or up-to-date state tree. On special cases where units are “stateless”, i.e. its state depends only on the last executed transaction, all other verification methods apply as well.

Ledger Proof is created by a node with up-to-date state tree of specific shard. See Fig. 20, Fig. 21, Fig. 22 how this can be orchestrated as a service.

9.1.3.2 Block Proof Based

Block Proof (denoted as Ξ) certifies execution of transaction $(P; s)$ during a partition/shard round; certified by an Unicity Certificate generated at a Root Chain round. Block Proof can be verified by the function call

$$\text{VerifyBlockProof}(\Xi, (P; s), \mathcal{T}) \ .$$

Block Proofs are special because they can be generated efficiently from ledger content, based on the last block with transaction involving a bill. It is not necessary to have access to a full node with up-to-date state tree. See Fig. 24 how this function can be set up.

9.1.3.3 Unit Ledger Based

Unit Ledger is a list of unit records with all transactions executed on a unit. Based on Unit Ledger, it is possible to validate all transaction orders and compute unit's state as it evolves.

Unit Ledger based verification is special, because it validates all operations performed by shard/partition validators, thus providing stronger security by not relying on honest majority of validators.

9.1.3.4 Ledger Based

Given access to a ledger, the verification of a transaction can take many forms.

1. Find the latest block with transaction affecting a unit; if block is valid then transaction can be considered as valid.
2. Replay and validate all transactions in blocks, build state tree, the last state and executed transaction are validated results. This option has similar properties with Unit Ledger based verification, but with extra overhead of verifying all units.
3. Replay and validate all transactions in blocks, build state tree, extract ledger proof, validate ledger proof.
4. Find the latest block with transaction affecting a unit; extract block proof, validate block proof.

9.1.3.5 Full Node Based

A full node validates transactions, either by participating in consensus and receiving block proposals and UC-s, or by downloading all blocks. Unit state is defined as the state maintained by a correct full node.

9.1.4 Obtaining Proofs

9.1.4.1 From Partition/Shard Nodes

Ledger Proofs can be obtained from a partition/shard validator node, offering such service. Ledger Proof generation is rather expensive for validators—ledger proofs can be generated only during a short window in round cycle, while the UC closing a block is known and the state tree is not yet dirty due to continuing application of new validated transactions.

9.1.4.2 From RPC Nodes

RPC node is dedicated for offering services to end users applications. Based on input of new blocks from one (Fig. 22) or more partitions/shards (Fig. 21), a RPC Node maintains

its state tree and serves queries about states of bills. In a sense, RPC node provides a proxy service.

Like partition/shard nodes, the RPC nodes have only up-to-date state tree and can not serve historical queries about previous states of units.

9.1.4.3 Locally from Ledgers

On this case, the ledgers are processed client-side (by so-called “full node client”) and proofs are extracted locally (Fig. 23). Necessary tools are packaged as “full node SDK”.

9.2 Architecture

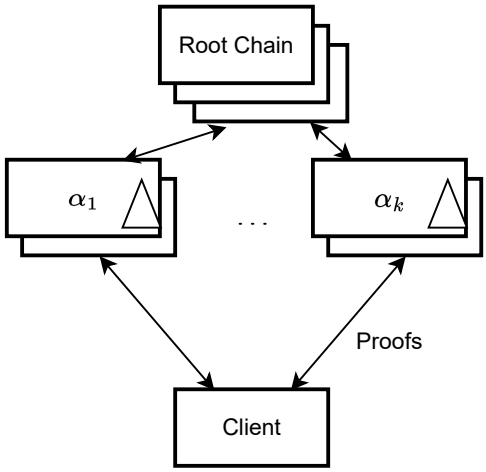

Figure 20. Client interactions with Alphabill; internal state tree is depicted as triangle. Shard/partition nodes provide ledger proof service.

9.2.1 Wallet

Wallet is a component which produces Transaction Orders. In order to do so, it must have access to signing capability: the access to a private key or keys. Usually a “wallet” knows the list of units it “owns” and can spend. There might be other functionality packaged with a wallet—for example the capability to verify transactions. Wallet may be the central component executing and orchestrating Composite Transactions, involving possibly many shards and transaction systems.

9.2.1.1 Thin Wallet

9.2.1.2 Custodial Wallet

9.2.2 RPC Node

RPC Node is a component which is offering services to Alphabill users. Its input is new blocks delivered over the LedgerReplication protocol. Unlike a partition/shard node, an RPC node does not participate in consensus.

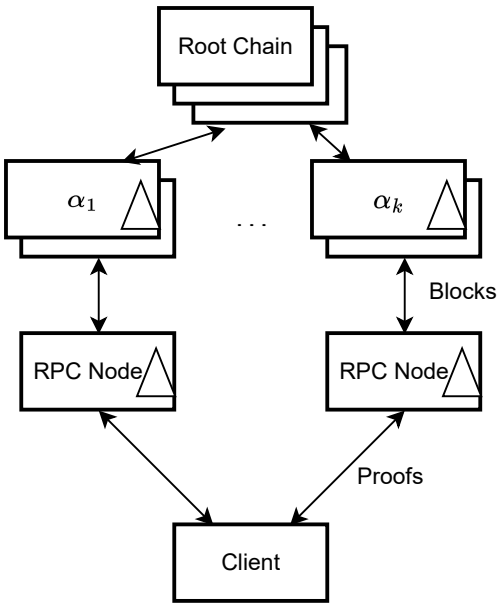

Figure 21. Client interactions with Alphabill; internal state tree is depicted as triangle. RPC nodes ingest blocks from shard/partition nodes and provide ledger proof service. There is one RPC node per shard/partition.

9.3 Transacting

9.3.1 Flows

Sequence diagrams on Fig. 26, Fig. 27, Fig. 28, Fig. 29 depict typical transaction flows in Alphabill.

Only Alphabill specific interaction sequences are depicted. It is assumed that the flows are executed by real-world applications within specific contexts. For example, an application connects recipients and senders, informs sender about expected transaction attributes, and provides messaging within the frames of its use-case.

Sequence diagrams on Fig. 30 depict sequential composition of transactions, e.g. for composed smart contract execution.

Please refer to *Alphabill Atomicity Partition* speification for the flow of multi-phase atomic multi-unit transactions.

9.3.2 Algorithms

9.3.3 Protocols

9.3.3.1 Protocol LedgerProof

The protocol returns ledger proofs, which provide a compact cryptographic proof of execution of a transaction. Ledger proofs can be delivered to third parties, including the recipient of a transaction, for independent verification, based on pre-agreed unicity trust base. The protocol has two varieties:

1. Provides proof about current state of a unit;

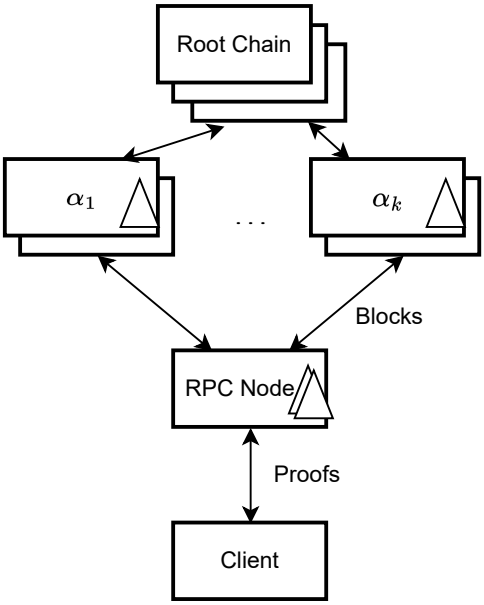

Figure 22. Client interactions with Alphabill; internal state tree is depicted as triangle. RPC node ingests blocks from shard/partition nodes and provide ledger proof service. One RPC node has the state of many shards/partitions.

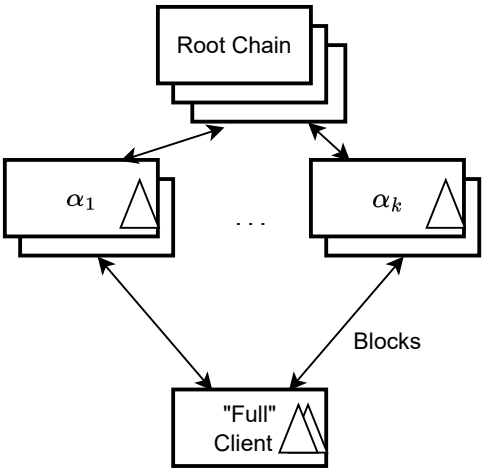

Figure 23. Client interactions with Alphabill; internal state tree is depicted as triangle. “Full node client” ingests blocks and reconstructs the state tree for all partitions/shards it cares about.

2. Provides the proof of execution of a specific transaction, identified in request by its hash. The request can block briefly, until transaction execution and availability of the proof, or until time-out.

In a sense, the second variety notifies about the execution of a transaction. This enables optimization for sequential execution of (composed) operations for least possible latency. The service does not provide information about past states of units and about content of transaction orders.

Query: $\langle \texttt{LedgerProof} \mid \alpha, \iota \rangle$

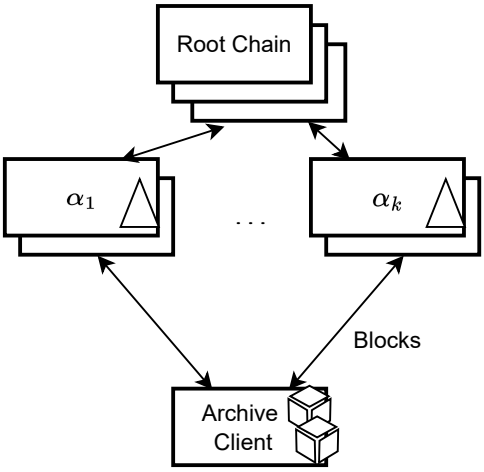

Figure 24. Client interactions with Alphabill; a blockchain copy is depicted as isometric cube. One archiving client replicates the copies of all shard/partition ledgers it cares about.

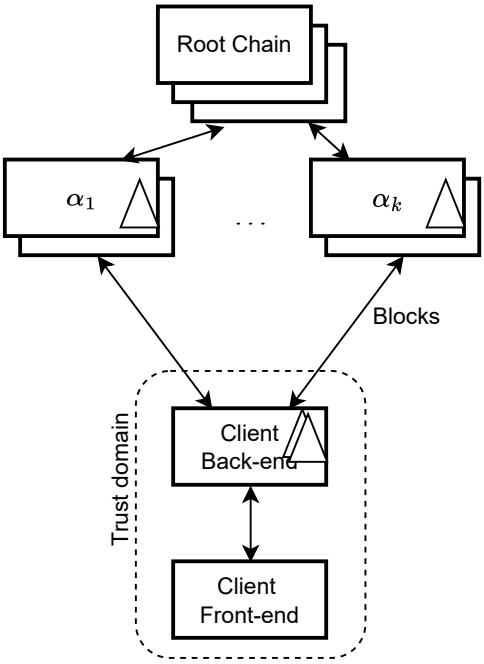

Figure 25. Client interactions with Alphabill; internal state tree is depicted as a triangle. Client application is decomposed into layers; only back-end interacts with Alphabill using designated SDK-s; and front-end fully trusts the back-end.

Reply: $(\Pi, D, \varphi, n_\alpha)$

where Π certifies the "current state" of unit ι at partition/shard round n_α. The query for specific transaction includes transaction hash $\lambda = H_d(P'; s')$:

Query: $\langle$`LedgerProof` $\mid \alpha, \iota, \lambda\rangle$,

Reply: $(\Pi, D, \varphi, n_\alpha)$

The query may block for up to T_0^{max} + `t2` time units (preconfigured).

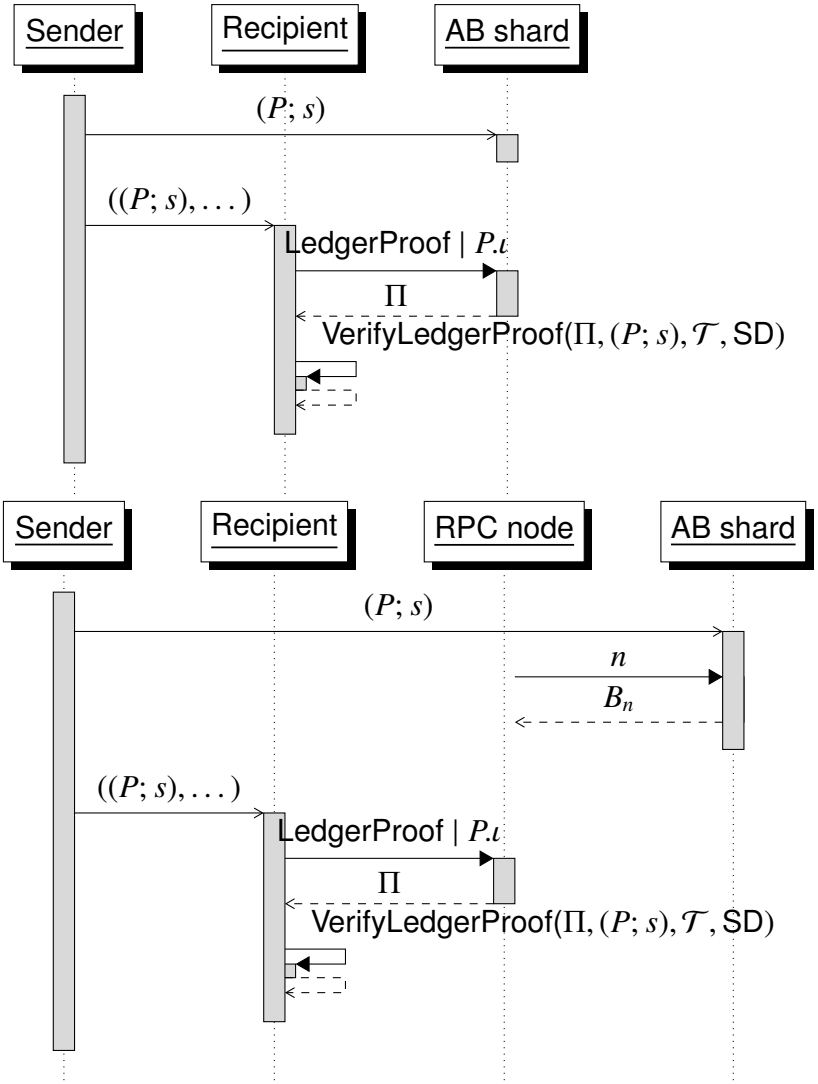

Figure 26. Sender notifies the recipient; recipient obtains ledger proof. LedgerProof service may be provided by AB shard node (1st diagram) or an RPC Node (2nd diagram).

There are following defined outcomes:

- Success: returned when the ledger proof about successfully executed transaction becomes (or is) available. $\lambda = H_d(P; s)$ and VerifyLedgerProof$(\Pi, (P; s), \mathcal{T}, \mathrm{SD}) = 1$
- Unknown transaction. Wait until max. transaction order execution timeout $T_0^{\max}$, then Reply is returned with last partition/shard block number and error code.
- Not executed transaction (e.g. invalid). Error code is returned. Returned only when validation happens while the query is blocked. Otherwise, a general error is returned (unknown transaction).
- Transaction is already executed, but there is another following transaction executed with the unit changing its state again, thus it is not possible to produce the ledger proof. Return an error and partition/shard block number n_α when the interesting transaction was executed. This information is available at least until $T_0^{\max} + \mathtt{t2}$ have passed, afterwards a generic error (unknown transaction) is returned.

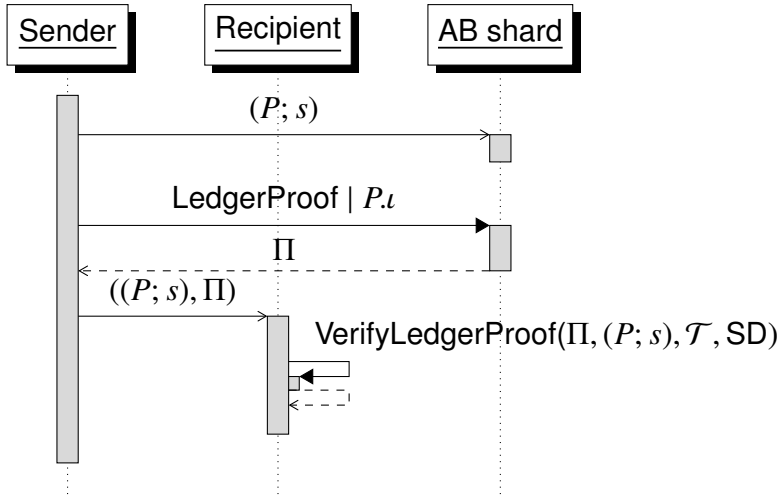

Figure 27. Sender notifies the recipient and provides ledger proof. As on previous figure, the LedgerProof service may be provided by an RPC Node (not depicted here).

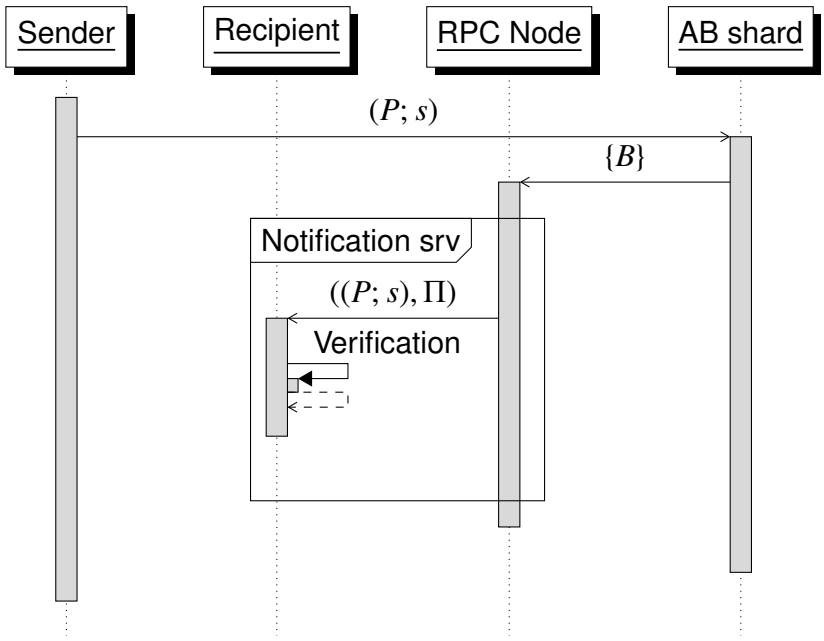

Figure 28. Transaction with middleware providing notification service

In order to identify executed payment orders, a node maintains a table of executed transaction order hashes and partition/shard block numbers identifying the block recording the transaction.

It is expected that client may fail over to another partition/shard node on error. Note, that in the byzantine setting, the no-execution results are just “opinions” of respective nodes.

9.3.3.2 Protocol BlockProof

BlockProof protocol returns either the known last transaction with a unit, or the block proof of specified transaction order execution.

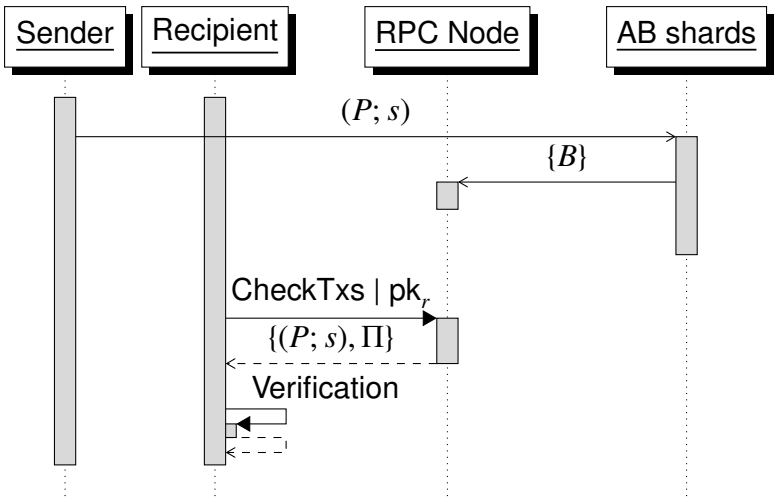

Figure 29. Transaction with middleware where the recipient checks for recent incoming transactions, or for listing of all his bills (known to RPC node which in turn may sync with one or more partitions)

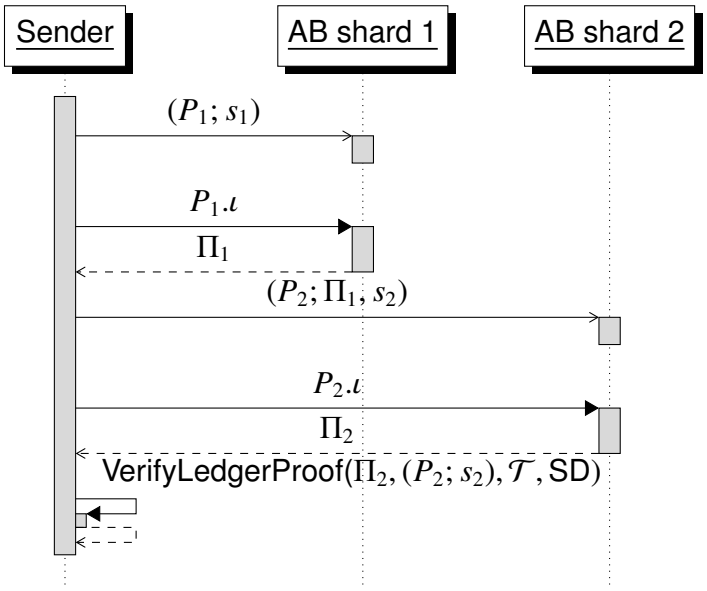

Figure 30. Sequential composition of transactions. Optional middleware is not depicted.

Query: $\langle$`BlockProof` | $\alpha, \iota)\rangle$,

Reply: $(\Xi, (P; s), \alpha, \sigma, n, h_-)$

Query: $\langle$`BlockProof` | $\alpha, \iota, \lambda)\rangle$,

Reply: $(\Xi, (P; s), \alpha, \sigma, n, h_-)$

If transaction is not found then the block number range where the search was performed is returned. Note that this is not a non-existence proof.

The block number n can be verified by checking that VerifyBlockProof($\Xi, P, \mathcal{T} = 1$) and $\Xi.h_h = H(\alpha, \sigma, n, h_-)$.

9.3.3.3 Protocol CheckTxs

This protocol has two varieties: a) return information about recent transactions, where recent is defined as everything executed after a round/block number specified in the query; and b) return all units owned by a party; where ownership is defined as a specific predicate content. A party might have multiple wallets, a wallet may contain multiple keys, and there are many ways how to encode an ownership predicate. The search works using literal comparison of predicate content—thus it is up to interested party to reach out-of-band agreement on a fixed set of keys and predicate encoding.

It is not feasible to evaluate predicates during a search.

The variety where all units in specific shard/partition locked by a predicate are returned can be achieved by setting the earliest block number to 0.

Query: $\langle \texttt{CheckTxs} \mid \alpha, \{\sigma\}, \varphi \rangle$,

Reply: $(\{\sigma, n\}, \{(P; s), n_P, \Pi\})$

The query includes the following parameters:

- α – system identifier,
- σ – optional list of shard identifiers to limit the search to specific shards;
- φ – literal predicate as a byte array.

Response:

- $\{\sigma, n\}$ – limit the scope of response to a set of shards, where for each specified shard the latest known block number is n,
- a set of
  - $(P; s)$ – signed transaction order,
  - n_P – the block number recording the transaction,
  - Π – ledger proof certifying the transaction and unit state at specified block number.

9.3.3.4 Protocol StateReplication

This protocol returns a recent *state file*, a snapshot of shard/partition state. A state file is used to quickly recover from scratch or instantiate a shard/partition node, an RPC node with state tree, or a full node client.

Applying a state file is equivalent to sequentially applying blocks from the same ledger, until the block number provided in the state file.

A state file has the same data type as *genesis block*: $B_0 = \langle \alpha, \sigma, \mathcal{SH}, \underline{\iota}_L, \underline{\iota}_R, n, \iota_r, N, \mathcal{T}, \mathcal{SD} \rangle$

Query: $\langle \texttt{StateReplication} \mid \alpha, \sigma, n' \rangle$,

Reply: (B_0^n)

Here, client specifies the transaction system identifier and optional shard identifier and block number which defines the earliest snapshot block number the client is interested, i.e. $B_0^n.n >= n'$.

9.4 Libraries

9.4.1 Wallet

9.4.2 Thin Client

9.4.3 Full Node Client

9.4.4 Alphabill Platform SDK

This toolkit allows building new partitions, bridges and oracles.

9.5 Guarantees

9.5.1 Reliability of Transactions

9.5.2 Atomic Composite Transactions

9.5.3 Serial Composition of Transactions

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

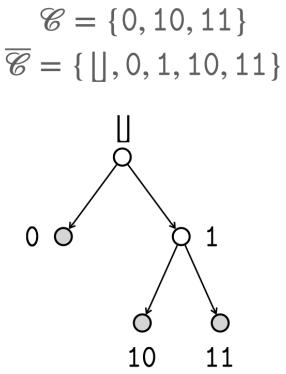

Figure 31. Irreducible prefix free code C and its closure $\overline{C}$.

B Encodings

B.1 Primitive Types

All **integers** are unsigned and represented with the bits ordered from the most significant to the least significant (also known as big endian or network byte order).

Time is expressed as the number of seconds since 1970-01-01 00:00:00 UTC, encoded as an unsigned integer. The count is zero-based: the time value for 1970-01-01 00:00:00 UTC is 0, the value for 1970-01-01 00:00:01 UTC is 1, etc. All days are considered to be exactly 86,400 seconds long. Leap seconds are handled by extending the duration of round immediately preceding the leap second.

When higher precision is needed, time is expressed as number of microseconds since 1970-01-01 00:00:00 UTC, encoded as an unsigned integer. During a leap second, the part corresponding to full seconds (up from the seventh digit in decimal notation) is kept the same as during the previous second, but the part corresponding to fractions of a second (the six 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.

All **text strings** are represented in the UTF-8 encoding of the UNICODE character set.

B.2 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.3 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: $r \in \mathbb{H} \cup \{\perp\}$, the value in the root of the tree

Computation:

function PLAIN_TREE_ROOT(L)
  if $n = 0$ **then** ▷ $L = \langle\rangle$
    return $\perp$
  else if $n = 1$ **then** ▷ $L = \langle x_1 \rangle$
    return x_1
  else
    $m \leftarrow 2^{\lfloor \log_2(n-1) \rfloor}$
    $L_{\text{left}} \leftarrow \langle x_1, \ldots, x_m \rangle$
    $L_{\text{right}} \leftarrow \langle x_{m+1}, \ldots, x_n \rangle$
    return $H($PLAIN_TREE_ROOT(L_{left}), 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 ot the chain

Output: $C = \langle (b_1, y_1), \ldots, (b_\ell, y_\ell) \rangle \in (\{0, 1\} \times \mathbb{H})^\ell$, 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. $C = \langle (b_1, y_1), \ldots, (b_\ell, y_\ell) \rangle \in (\{0,1\} \times \mathbb{H})^\ell$, 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 $b_\ell \in \{0,1\}$
    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_ℓ)
    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.

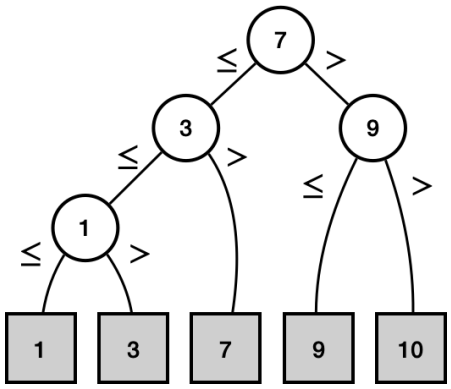

Figure 32. Keys of the nodes of an indexed hash tree.

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 – system identifier of type $\mathbb{A}$
- $\mathcal{SH}$ – sharding scheme of type $\mathbb{SH}$
- σ – shard identifier of type $\{0,1\}^{\mathcal{SH}.k}$
- $\underline{\iota}_L$ – left separator of type $\underline{\mathbb{I}} \cup \{\perp\}$
- $\underline{\iota}_R$ – right separator of type $\underline{\mathbb{I}} \cup \{\perp\}$
- $\mathcal{T}$ – unicity trust base of type $\mathbb{B}$
- $\mathcal{SD}$ – system descriptors of type $\mathbb{SD}[\mathbb{A}]$ for all registered partitions (including $\mathcal{SD}[\alpha]$)
- 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 $\mathbb{I}$
- φ – bearer condition, of type $\mathbb{L}$
- D – unit data, of type $\mathcal{SD}[\alpha].\mathbb{D}$
- x – state hash, of type $\mathbb{H}$
- $\langle (b_1, y_1), \ldots, (b_m, y_m) \rangle$ – the hash chain linking φ, D and x to the root of the unit tree, where b_i are of type $\{0,1\}$ and y_i of type $\mathbb{H}$
- `hasLeft` – existence of left child, of type $\{0,1\}$ (1-exists, 0-does not exist)
- `hasRight` – existence of right child, of type $\{0,1\}$ (1-exists, 0-does not exist)

Note that φ, 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. 4.10.2.

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 ckecksum

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. 4.10.

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

$N[\iota].d \leftarrow 1 + \max\{N[N[\iota].\iota_L].d, N[N[\iota].\iota_R].d\}$
$N[\iota].b \leftarrow N[N[\iota].\iota_R].d - N[N[\iota].\iota_L].d$
$N[\iota].V \leftarrow (S.\mathcal{SD}[\alpha].F_S)((S.\mathcal{SD}[\alpha].V_S)(N[\iota].D), N[N[\iota].\iota_L].V, N[N[\iota].\iota_R].V)$
▷ Compute pre-pruning value of the sub-tree summary hash
$h_s \leftarrow$ PLAIN_TREE_OUTPUT$(\langle(b_1, y_1), \ldots, (b_m, y_m)\rangle, H(R.x, H(\varphi, D)))$
$h \leftarrow H(\iota, h_s, N[\iota].V; h_L, N[N[\iota].\iota_L].V; h_R, N[N[\iota].\iota_R].V)$
push$((\iota, h))$
▷ Compute post-pruning value of the sub-tree summary hash
$N[\iota].S \leftarrow \langle(\bot, R.x, N[\iota].\varphi, N[\iota].D)\rangle$
$N[\iota].h_s \leftarrow$ PLAIN_TREE_ROOT$(\langle H(R.x, H(N[\iota].\varphi, N[\iota].D))\rangle)$
$N[\iota].h \leftarrow H(\iota, N[\iota].h_s, N[\iota].V; N[N[\iota].\iota_L].h, N[N[\iota].\iota_L].V; N[N[\iota].\iota_R].h, N[N[\iota].\iota_R].V)$
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.

E Atomicity Partition Type (v1)

E.1 Motivation and General Description

E.1.1 Motivation

Let $u_1, \ldots, u_m$ be units with identifiers $\iota_1, \ldots, \iota_m$ and owner conditions $\varphi_1, \ldots, \varphi_m$, respectively.

The units $u_1, \ldots, u_m$ may belong to different transaction systems (partitions) with identifiers $\alpha_1, \ldots, \alpha_m$, respectively. It is assumed that in all these partitions there are transaction types for changing the ownership conditions of units.

Goal: transfer the units atomically to new owner conditions $\varphi'_1, \ldots, \varphi'_m$ so that either:

- *all transfers happen* – all units $u_1, \ldots, u_m$ are transferred to the new owner conditions $\varphi'_1, \ldots, \varphi'_m$, or
- *none of the transfers happen* – all units will have owner conditions equivalent to the previous conditions $\varphi_1, \ldots, \varphi_m$

All units may potentially be controlled by different parties. We assume that these parties may communicate in order to agree on the atomic transfer, i.e. after communication, all parties know $\iota_1, \ldots, \iota_m, \varphi_1, \ldots, \varphi_m, \alpha_1, \ldots, \alpha_m$. The parties also agree on other transaction specific parameters.

If there is more than one party, this is an *atomic swap*. If there is a single party, this is an *atomic multi-unit transfer*.

E.1.2 General Description of the Atomicity Partition

There is a specific transaction system (partition) with identifier α_0 that provides necessary unique references for atomic multi-unit transactions. We call this *atomicity partition*.

Units of the atomicity partition are the atomic multi-unit transactions, i.e. every such transaction has a unique pseudo-random identifier ι (referred to as *contract identifier*) in the atomicity partition.

There is no ownership or value for the units of the atomicity partition, so it is reasonable to assume that the value of every unit is $v = 0$, and the ownership condition of every unit is $\varphi \equiv 1$, i.e. identically true.

Transactions of the atomicity partition are:

1. reg – registering new atomic multi-unit transaction with contract identifier ι

2. con – confirming an existing multi-unit transaction with contract identifier ι

E.2 Phases of Atomic Multi-Unit Transactions

E.2.1 Phase 1: Preparation

Parties prepare transaction orders $P_1, \ldots, P_m$ that transfer the ownerships of the units $\iota_1, \ldots, \iota_m$ to a special parametrized owner predicates $\varphi_{\text{ato}}(\alpha_0, \iota, t_0, \varphi_1, \varphi'_1; \cdot; \cdot, \cdot), \ldots, \varphi_{\text{ato}}(\alpha_0, \iota, t_0, \varphi_m, \varphi'_m; \cdot; \cdot, \cdot)$

The contract identifier ι is computed as a deterministic pseudo-random function on a 128-bit random nonce r, the payment orders $P_1, \ldots, P_m$ without signatures (owner proofs) and the owner predicates $\varphi_{\text{ato}}(\alpha_0, 0^{256}, t_0, \varphi_i, \varphi'_i; \cdot; \cdot, \cdot)$, where in the place of ι (which does not yet exist) there is 0^{256}.

The predicate $\varphi_{\text{ato}}(\alpha_0, \iota, t_0, \varphi, \varphi'; \cdot; \cdot, \cdot)$ is defined as follows:

$\varphi_{\text{ato}}(\alpha_0, \iota, t_0, \varphi, \varphi'; P; \Pi, s)$, where the pair (Π, s) represents the owner proof, is true if either:

1. $\varphi'(P, s) = 1$, and Π is a proof that status $= 1$ in the partition α_0 in a round with number $t < t_0$ of the status of contract ι, or
2. $\varphi(P, s) = 1$, and Π is a proof that status $= 0$ in the partition α_0 in a round with number $t \geq t_0$ of the status of contract ι, or
3. $\varphi(P, s) = 1$, and Π is a proof in the partition α_0 in a round with number $t \geq t_0$ of the non-existence of contract ι

E.2.2 Phase 2: Registration

One of the parties creates the reg message $\langle \alpha_0, \text{reg}, \iota, A, T_0 \rangle$ with:

1. α_0 – the identifier of the atomicity partition
2. reg – message type
3. ι – contract identifier
4. A – attribute field that contains the identifiers $\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m$, the random nonce r, and the timeout t_0 of the multi-unit transaction
5. T_0 – timeout of the reg message

The party sends the reg message to the atomicity partition.

The new unit with identifier ι is created in the atomicity partition with the data part D containing $\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m, r, t_0$, status $= 0$, confirmed $= \emptyset$.

E.2.3 Phase 3: Confirmation

Parties sign their transaction orders $P_1, \ldots, P_m$ (by adding ownership proofs s_i to P_i) and send them to the corresponding partitions $\alpha_1, \ldots, \alpha_m$.

Parties obtain the unit proofs $\Pi_1, \ldots, \Pi_m$ for the signed transactions $(P_1, s_1), \ldots, (P_m, s_m)$.

Every party i sends the confirmation message $\langle \alpha_0, \text{con}, \iota, A_i, T_0 \rangle$ to the atomicity partition with $A_i = (P_i, s_i, \Pi_i)$.

Having received a con message $\langle \alpha_0, \text{con}, \iota, (P_i, s_i, \Pi_i), T_0 \rangle$, the atomicity partition verifies (P_i, s_i, Π_i), checks if the identifiers α_i, ι_i are consistent with the data part $D = (\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m, \text{status}, \text{confirmed})$ of the contract ι. After a successful verification, the triple (P_i, s_i, Π_i) is added to the set confirmed.

If all transactions have been confirmed, i.e. if confirmed $= \{(P_1, s_1, \Pi_1), \ldots, (P_m, s_m, \Pi_m)\}$, then the atomicity partition checks if the contract identifier ι was correctly computed based on $P_1, \ldots, P_m$, and sets status $\leftarrow 1$.

E.3 Specification of the Atomicity Partition

E.3.1 Parameters, Types, Constants, Functions

System identifier: α_0

Summary value type $\mathbb{V}$: $\{0, 1\}$

Summary trust base: $\mathcal{V} = 0$

Summary check: $\gamma \equiv 1$

Units $u \in U$: multi-unit atomic transactions

Data type $\mathbb{D}$: tuples $(\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m, t_0, \text{status}, \text{confirmed})$ where:

1. $\alpha_1, \ldots, \alpha_m$ – system identifiers of type $\mathbb{A}$
2. $\iota_1, \ldots, \iota_m$ – unit identifiers of type $\mathbb{I}$
3. r – random nonce of type $\mathbb{N}_{128}$
4. t_0 – atomicity timeout of type $\mathbb{N}_{64}$
5. status – status of type $\mathbb{N}_1$
6. confirmed – set of transaction orders with proofs, initially empty

Summary functions:

1. $V_s(D) = D.\text{status}$
2. $F_S(v, v_L, v_R) = 0$
3. $F_S(\perp, v_L, v_R) = 0$

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

Transaction types: $\mathbb{T} = \{\text{reg}, \text{con}\}$

E.3.2 Transactions

E.3.2.1 Register

Transaction $P = \langle \alpha_0, \text{reg}, \iota, A, T_0 \rangle$, with $A = (\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m, r, t_0)$, where:

1. $\alpha_1, \ldots, \alpha_m$ – a set of system identifiers of type $\mathbb{A}$
2. $\iota_1, \ldots, \iota_m$ – a set of unit identifiers, where ι_i is of type $\mathbb{I}_{\alpha_i}$ defined by $\mathcal{SD}$
3. r – random nonce of type $\mathbb{N}_{128}$

4. t_0 – atomicity timeout of type $\mathbb{N}_{64}$

Transaction-specific validity condition: there is no $N[\iota]$ and the current round number $S.n$ does not exceed t_0:

$$\psi_{\text{reg}}((P, s), S) \equiv N[\iota] = \perp \ \wedge \ S.n \leq t_0$$

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

1. **if** $N[\iota] = \perp$ **then**: $\text{AddItem}(\iota, 1, (\alpha_1, \ldots, \alpha_m, \iota_1, \ldots, \iota_m, r, t_0, 0, \emptyset))$

Table 20. Data fields of the reg transaction order.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_0
2.	transaction type	τ	$\mathbb{N}_8$	reg
3.	unit identifier	ι	$\mathbb{I}$	-
4.	system identifiers	$A.\alpha_1, \ldots, \alpha_m$	$\mathbb{A}$	-
5.	unit identifiers	$A.\iota_1, \ldots, \iota_m$	$\mathbb{I}$	-
6.	random nonce	$A.r$	$\mathbb{N}_{128}$	-
7.	atomicity timeout	$A.t_0$	$\mathbb{N}_{64}$	-
8.	message timeout	T_0	$\mathbb{N}_{64}$	-
9.	owner proof	s	$\{0,1\}^*$	$\sqcup$

E.3.2.2 Confirm

Transaction $P = \langle \alpha_0, \text{con}, \iota, A, T_0 \rangle$, with $A = (P_i, s_i, \Pi_i)$, where:

1. P_i – transaction order
2. s_i – owner proof
3. Π_i – unit proof

Transaction-specific validity condition: the confirmation message is not expired, there exist $N[\iota]$, the unit identifier in P_i belongs to the set $\{\iota_1, \ldots, \iota_m\}$, owner proof and the unit proof verify:

$$\psi_{\text{con}}((P, s), S) \equiv S.n \leq N[\iota].D.t_0 \wedge N[\iota] \neq \perp \wedge P_i.\iota \in N[\iota].\{\iota_1, \ldots, \iota_m\} \wedge \text{VerifyUnitProof}(\Pi_i, (P_i, s_i), \mathcal{T}, \mathcal{SD})$$

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

1. $N[\iota].D.\text{confirmed} \leftarrow N[\iota].D.\text{confirmed} \cup A_i$
2. **if** $|N[\iota].D.\text{confirmed}| = m$ **then**: $N[\iota].D.\text{status} \leftarrow 1$

Table 21. Data fields of the con transaction order.

No	Field	Notation	Type	Predefined value
1.	system identifier	α	$\mathbb{A}$	α_0
2.	transaction type	τ	$\mathbb{N}_8$	con
3.	unit identifier	ι	$\mathbb{I}$	-
4.	transaction order	$A.P_i$	variable	-
5.	owner proof	$A.s_i$	$\{0,1\}^*$	-
6.	unit proof	$A.\Pi_i$	$\mathbb{SP}$	-
7.	message timeout	T_0	$\mathbb{N}_{64}$	-
8.	owner proof	s	$\{0,1\}^*$	-