Polkadot Runtime Environment

Protocol Specification

January 18, 2018

1 Conventions and Definitions

Definition 1. ***Runtime*** *is the state transition function of the decentralized ledger protocol.*

Definition 2. *A* ***path graph*** *or a* ***path*** *of n nodes, formally referred to as* $\boldsymbol{P_n}$*, is a tree with two nodes of vertex degree 1 and the other n-2 nodes of vertex degree 2. Therefore,* P_n *can be represented by sequences of* $(v_1, ..., v_n)$ *where* $e_i = (v_i, v_{i+1})$ *for* $1 \leqslant i \leqslant n-1$ *is the edge which connect* v_i *and* v_{i+1}*.*

Definition 3. ***radix r tree*** *is a variant of a trie in which:*

- *Every node has at most r children where* $r = 2^x$ *for some x;*
- *Each node that is the only child of a parent, which does not represent a valid key is merged with its parent.*

As a result, in a radix tree, any path whose interior vertices all have only one child and does not represent a valid key in the data set, is compressed into a single edge. This improves space efficiency when the key space is sparse.

Definition 4. *By a* ***sequences of bytes*** *or a* ***byte array****, b, of length n, we refer to*

$$b := (b_0, b_1, ..., b_{n-1}) \text{ such that } 0 \leqslant b_i \leqslant 255$$

We define $\mathbb{B}_n$ *to be the* ***set of all byte arrays of length n****. Furthermore, we define:*

$$\mathbb{B} := \bigcup_{i=0}^{\infty} \mathbb{B}_i$$

Notation 5. *We represent the concatination of byte arrays* $a := (a_0, ..., a_n)$ *and* $b := (b_0, ..., b_m)$ *by:*

$$a \,||\, b := (a_0, ..., a_n, b_0, ..., b_m)$$

Definition 6. *For a given byte b the* ***bitwise representation*** *of b is defined as*

$$b := b^7 ... b^0$$

where

$$b = 2^0 b^0 + 2^1 b^1 + \cdots + 2^7 b^7$$

2 Block

In Polkadot RE, a block is made of two main parts, namely the *block header* and the *list of extrinsics*. *The Extrinsics* represent the generalization of the concept of *transaction*, containing any set of data that is external to the system, and which the underlying chain wishes to validate and keep track of.

2.1 Block Header

The block header is designed to be minimalistic in order to boost the efficiency of the light clients. It is defined formally as follows:

Definition 7. *The* ***header of block B,*** $\mathbf{Head}(\boldsymbol{B})$ *is a 5-tuple containing the following elements:*

- ***parent_hash:*** *is the 32-byte Blake2s hash of the header of the parent of the block indicated henceforth by* $\boldsymbol{H_p}$.
- ***number:*** *formally indicated as* $\boldsymbol{H_i}$ *is an integer, which represents the index of the current block in the chain. It is equal to the number of the ancestor blocks. The genesis block has number 0.*
- ***state_root:*** *formally indicated as* $\boldsymbol{H_r}$ *is the root of the Merkle trie, whose leaves implement the storage for the system.*
- ***extrinsics_root:*** *is the root of the Merkle trie, whose leaves represent individual extrinsics being validated in this block. This element is formally referred to as* $\boldsymbol{H_e}$.
- ***digest:*** *this field is used to store any chain-specific auxiliary data, which could help the light clients interact with the block without the need of accessing the full storage. Polkadot RE does not impose any limitation or specification for this field. Essentially, it can be a byte array of any length. This field is indicated as* $\boldsymbol{H_d}$

Definition 8. *The* ***Block Header Hash of Block B,*** $\boldsymbol{H_h(b)}$, *is the hash of the header of block B encoded by simple codec:*

$$H_b(b) := \text{Blake2}\, s(\text{Enc}_{\text{SC}}(\text{Head}(B)))$$

2.2 Justified Block Header

The Justified Block Header is provided by the consensus engine and presented to the Polkadot RE, for the block to be appended to the blockchain. It contains the following parts:

- **block_header** the complete block header as defined in Section 2.1 and denoted by Head(B).
- **justification**: as defined by the consensus specification indicated by Just(B) [link this to its definition from consensus].
- **authority Ids**: This is the list of the Ids of authorities, which have voted for the block to be stored and is formally referred to as $A(B)$. An authority Id is 32bit.

2.3 Extrinsics

Each block also contains a list of extrinsics. Polkadot RE does not specify or limit the internal of each extrinsics beside the fact that each extrinsics is a blob of encoded data. The extrinsics_root should correspond to the root of the Merkle trie, whose leaves are made of the block extrinsics list.

3 Runtime

Polkadot RE expects to receive the code for the runtime of the chain as a compiled WebAssembly (Wasm) Blob. The current runtime is stored in the state database under the key represented as a byte array:

$$b = 3\,a, 63, 6\,f, 64, 65$$

which is the byte array of ASCII representation of string ":code" (see Section 10). For any call to the runtime, Polkadot RE makes sure that it has most updated runtime as calls to runtime have the ability to change the runtime code.

The initial runtime code of the chain is embedded as an extrinsics into the chain initialization JSON file and is submitted to Polkadot RE (see Section 9).

Subsequent calls to the runtime have the ability to call the storage API (see Section 12) to insert a new Wasm blob into runtime storage slot to upgrade the runtime.

3.1 Entries into Runtime

Polkadot RE assumes that at least the following functions are implemented in the Runtime Wasm blob and has been exported as shown in Snippet 1 :

```
(export "version" (func $version))
(export "authorities" (func $authorities))
(export "execute_block" (func $execute_block))
```

Snippet 1. Snippet to export entries into tho Wasm runtime module

The following sections describe the standard based on which Polkadot RE communicates with each runtime entry.

3.1.1 version

This entry receives no argument, it returns the vesion data encoded in ABI format described in Section 3.3 containing the following data:

Name	Type	Description
`spec_name`	String	runtime identifier
`impl_name`	String	the name of the implementation (e.g. C++)
`authoring_version`	32-bit uint	the version of the authorship interface
`spec_version`	32-bit uint	the version of the runtime specification
`impl_version`	32-bit uint	the version of the runtime implementation
`apis`	ApisVec	List of supported AP

Table 1. Detail of the version data type returns from runtime `version` function

3.1.2 authorities

This entry is to report the set of authorities at a given block. It receives `block_id` as an argument, it returns an array of `authority_id`'s.

3.1.3 execute_block

This entry is responsible to execute all extrinsics in the block and reporting back the changes into the state storage. It receives the block header and the block body as its arguments and it returns a triplet:

Name	Type	Description
`results`	Boolean	Indicating if the execution was su
`storage_changes`	[???]	Contains all changes to the state storage
`change_updat`	[???]	

Table 2. Detail of the data execute_block returns after execution

3.2 Code Executor

Polkadot RE provide a Wasm Virtual Machine (VM) to run the runtime. The Wasm VM exposes the Polkadot RE API to the Runtime. And execute the Runtime as a Wasm module.

3.3 ABI Encoding between Runtime and the Runtime Enviornment

All data exchanged between Polkadot RE and the runtime is encoded using SCALE codec described in Section 8.1.

4 Network API

4.1 Block Submission and Validation

Block validation is the process, by which the client asserts that a block is fit to be added to the blockchain. This means that the block is consistent with the world state and transitions from the state of the system to a new valid state.

Blocks can be handed to the Polkadot RE both from the network stack and from consensus engine.

Both the runtime and the Polkadot RE need to work together to assure block validity. This can be accomplished by Polkadot RE invoking `execute_block` entry into the runtime as a part of the validation process.

Polkadot RE implements the following procedure to assure the validity of the block:

Algorithm 1. IMPORT-AND-VALIDATE-BLOCK$(B, \text{Just}(B))$

1 VERIFY-BLOCK-JUSTIFICATION$(B, \text{Just}(B))$
2 Verify $H_{p(B)} \in$ Blockchain.
3 State-Changes = Runtime.(B)
4 UPDATE-WORLD-STATE(State-Changes)

5 State Storage and the Storage Trie

For storing the state of the system, Polkadot RE implements a hash table storage where the keys are used to access each data entry state. There is no limitation neither on the size of the key nor the size of the data stored under them, besides the fact that they are byte arrays.

5.1 Accessing The System Storage

Polkadot RE implements various functions to facilitate access to the system storage for the runtime. Section 12 lists all of those functions. Here we define the essential ones which are also used by the Polkadot RE.

Definition 9. *The **StateRead** and **StateWrite** functions provide basic access to the State Storage:*

$$v = \text{StateRead}(k)$$

$$\text{StateWrite}(k, v)$$

where v and k are byte arrays.

To authenticate the state of the system, the stored data needs to be re-arranged and hashed in a *radix 16 tree* also known as *base-16 modified Merkle Patricia Tree*, which hereafter we will refer to as the ***Trie***, in order to compute the hash of the whole state storage consistently and efficiently at any given time.

As well, a modification has been made in the storing of the nodes' hash in the Merkle Tree structure to save space on entries storing small entries.

Because the Trie is used to compute the *state root*, H_r, (see Definition 7), which is used to authenticate the validity of the state database, Polkadot RE follows a rigorous encoding algorithm to compute the values stored in the trie nodes to ensure that the computed Merkle hash, H_r, matches across clients.

5.2 The General Tree Structure

As the trie is a radix 16 tree, in this sense, each key value identifies a unique node in the tree. However, a node in a tree might or might not be associated with a key in the storage.

To identify the node corresponding to a key value, k, first we need to encode k in a uniform way:

Definition 10. *The for the purpose labeling the branches of the Trie key k is encoded to k_{enc} using KeyEncode functions:*

$$k_{\text{enc}} := (k_{\text{enc}_1}, ..., k_{\text{enc}_{2n}}) := \text{KeyEncode}(k) \tag{1}$$

such that:

$$\text{KeyEncode}(k): \begin{cases} \mathbb{B} & \to \text{Nibbles}_4 \\ k := (b_1, ..., b_n) := & \mapsto (b_1^1, b_1^2, b_2^1, b_2^2, ..., b_n^1, b_n^2) \\ & := (k_{\text{enc}_1}, ..., k_{\text{enc}_{2n}}) \end{cases}$$

where Nibble_4 *is the set of all nibbles of 4-bit arrays and b_i^1 and b_i^2 are 4-bit nibbles, which are the little endian representations of b_i:*

$$(b_i^1, b_i^2) := (b_1 \bmod 16, b_2 / 16)$$

, where mod is the remainder and / is the integer division operators.

By looking at k_{enc} as a sequence of nibbles, one can walk the radix tree to reach the node identifying the storage value of k.

5.3 The Merkle proof

To prove the consistency of the state storage across the network and its modifications efficiently, the Merkle hash of the storage trie needs to be computed rigorously.

The Merkle hash of the trie is computed recursively. As such, hash value of each node depends on the hash value of all its children and also on its value. Therefore, it suffices to define how to compute the hash value of a typical node as a function of the hash value of its children and its own value.

Definition 11. *Suppose node N of storage state trie has key value k_N, and parent key value of $k_{P(N)}$, such that:*

$$\text{KeyEncode}(k_N) = (k_{\text{enc}_1}, ..., k_{\text{enc}_{i-1}}, k_{\text{enc}_i}, ..., k_{\text{enc}_{2n}})$$

and

$$\text{KeyEncode}(k_{P(N)}) = (k_{\text{enc}_1}, ..., k_{\text{enc}_{i-1}})$$

We define

$$\text{pk}_N := (k_{\text{enc}_i}, ..., k_{\text{enc}_{2n}})$$

to be the ***the partial key*** *of N.*

Definition 12. *For a trie node N,* ***Node Prefix*** *function is a value specifying the node type as follows:*

$$\text{NodePrefix}(N) := \begin{cases} 1 & N \text{is a leaf node} \\ 254 & N \text{is a branch node without value} \\ 255 & N \text{is a branch node with value} \end{cases}$$

Definition 13. *For a given node N, with partial key of* pk_N *and Value v, the* ***encoded representation*** *of N, formally referred to as* $\text{Enc}_{\text{Node}}(N)$ *is determined as follows, in case which:*

- *N is a leaf node:*

$$\text{Enc}_{\text{Node}}(N) := \text{Enc}_{\text{len}}(N) || \text{HPE}(\text{pk}_N) || \text{Enc}_{\text{SC}}(v)$$

- *N is a branch node:*

$$\begin{aligned} &Enc_{Node}(N):= \\ &\text{NodePrefix}(N) || \text{ChildrenBitmap}(N) \| \text{HPE}_{\text{PC}}(v) || \text{Enc}_{\text{SC}}(\text{Enc}_{\text{Node}}) || \\ &\text{Enc}_{\text{SC}}(N_{C_1}) ... \text{Enc}_{\text{SC}}(N_{C_n}) \end{aligned}$$

Where $N_{C_1} ... N_{C_n}$ with $n \leqslant 16$ are the children nodes of N.

Definition 14. *For a given node N, the* ***Merkle value*** *of N, denoted by $H(N)$ is defined as follows:*

$$H: \mathbb{B} \to \bigcup_{i=0}^{32} \mathbb{B}_i$$

$$H(N): \begin{cases} \text{Enc}_{\text{Node}}(N) & \|\text{Enc}_{\text{Node}}(N)\| < 32 \\ \text{Hash}(\text{Enc}_{\text{Node}}(N)) & \|\text{Enc}_{\text{Node}}(N)\| \geqslant 32 \end{cases}$$

6 Extrinsics trie

To validate that the Extrinsics data are stored in a block across clients, Polkadot RE uses the same trie structure as for the state storage described in Section 5.3 to generate the Merkle proof.

7 Consensus Engine

Consensus in Polkadot RE is achieved during the execution of two different procedures. The first procedure is block production and the second is finality. Polkadot RE must run these procedures, if and only if it is running on a validator node.

7.1 Block Tree

In the course of formation of a (distributed) blockchain, it is possible that the chain forks into multiple sub chains in various block positions. We refer to this structure as a *block tree:*

Definition 15. *The **Block Tree** of a blockchain is the union of all different versions of the blockchain observed by all the nodes in the system such as every such block is a node in the graph and B_1 is connected to B_2 if B_1 is a parent of B_2.*

Because every block in the blockchain contains a reference to its parent, it is easy to see that the block tree is actually a tree.

A block tree naturally imposes a partial order relationships on the blocks as follows:

Definition 16. *We say **B is descendant of B'**, formally noted as $B > B'$ if B is a descendent of B' in the block tree.*

7.2 Block Production

7.3 Finality

Polkadot RE uses GRANDPA Finality protocol [?] to finalize blocks. Finality is obtained by consecutive rounds of voting by validator nodes. Validators execute GRANDPA finality process in parallel to Block Production as an independent service. In this section, we describe the different functions that GRANDPA service is supposed to perform to successfully participate in the block finalization process.

7.3.1 Priliminaries

Definition 17. *A **GRANDPA Voter**, v, is represented by a key pair (k_v^{pr}, k_v) where k_v^{pr} represents its private key, is a node running GRANDPA protocol, and broadcasts votes to finilize blocks in a Polkadot RE - based chain. The **set of all GRANDPA voters** is indicated by $\mathbb{V}$.*

Definition 18. ***GRANDPA state**, GS, is defined as*

$$\text{GS} := \{\mathbb{V}, \text{id}_{\mathbb{V}}, r\}$$

where:

$\mathbb{V}$*: is the set of voters.*

$\mathbb{V}_{\mathbf{id}}$*: is an incremental counter tracking membership, which changes in V.*

$\boldsymbol{r}$*: is the voting round number.*

Now we need to define how Polkadot RE counts the number of votes for block B. First a vote is defined as:

Definition 19. *A **GRANDPA vote** or simply a vote for block B is an ordered pair defined as*

$$V(B) := (H_h(B), H_i(B))$$

where $H_h(B)$ and $H_i(B)$ are block hash and block number defined in Definitions 7 and 8 respectively.

Definition 20. *Voters engage in a maximum two sub-rounds of voting for each round r. The first sub-round is called **pre-vote** and the second sub-round is called **pre-commit**.*

By $\boldsymbol{V_v^{r,\mathbf{pv}}}$ and $\boldsymbol{V_v^{r,\mathbf{pc}}}$ we refer to the vote casted by voter v in round r (for block B) during the pre-vote and the pre-commit sub-round respectively.

The GRANDPA protocol dictates how an honest voter should vote in each sub-round, which is described in Algorithm 3. After defining what consititue vote in GRANDPA, we define how GRANDPA counts votes.

Definition 21. *Voter v **equivocates** if they broadcast two or more valid votes to blocks not residing on the same branch of the block tree during one voting sub-round. In such a situation, we say v is an **equivocator** all votes $V_v^{r,\text{stage}}(B)$ casted by v in that round is an **equivocatory vote** and*

$$\mathcal{E}^{r,\text{stage}}$$

represents the set of all equivocators voters in sub-round "stage" of round r. When we want to refer to the number of equivocators whose equivocation has been observed by voter v we refer to it by:

$$\mathcal{E}^{r,\text{stage}}_{\text{obs}(v)}$$

Definition 22. *A vote $V_v^{r,\text{stage}} = V(B)$ is **invalid** if*

- *$H(B)$ does not correspond to a valid block.*
- *B is not an (eventual) descendent of a previously finalized block.*
- *$M_v^{r,\text{stage}}$ does not bear a vaid signature.*
- *$\text{id}_{\mathbb{V}}$ does not match the current $\mathbb{V}$.*
- *If $V_v^{r,\text{stage}}$ is an equivacatory vote.*

Definition 23. *For validator v, **the set of observed direct votes for Block B in round r**, formally denoted by $\text{VD}^{r,\text{stage}}_{\text{obs}(v)}(B)$ is equal to the union of:*

- *set of valid votes $V_{v_i}^{r,\text{stage}}$ casted in round r and received by v such that $V_{v_i}^{r,\text{stage}} = V(B)$.*

Definition 24. *We refer to **the set of total votes observed by voter v in sub-round** stage **of round r** by $\boldsymbol{V^{r,\text{stage}}_{\text{obs}(v)}}$.*

*The **set of all observed vote by v in sub-round stage of round $r$for block B**, $\boldsymbol{V^{r,\text{stage}}_{\text{obs}(v)}(B)}$ is equal to all observed direct votes casted for block B and all B's descendents defined formally as:*

$$V^{r,\text{stage}}_{\text{obs}(v)}(B) := \bigcup_{v_i \in \mathbb{V}, B \geqslant B'} \text{VD}^{r,\text{stage}}_{\text{obs}(v)}(B')$$

*The **total number of observed vote for Block B in round r** is defined to be the size of that set plus total number of equivocators voters:*

$$\#V^{r,\text{stage}}_{\text{obs}(v)}(B) = |V^{r,\text{stage}}_{\text{obs}(v)}(B)| + |\mathcal{E}^{r,\text{stage}}_{\text{obs}(v)}|$$

Definition 25. *The current **pre-voted** block $B_v^{r,\mathrm{pv}}$ is the block with*

$$H_n(B_v^{r,\mathrm{pv}}) = \mathrm{Max}(H_n(B)|\,\forall B\colon \#V_{\mathrm{obs}(v)}^{r,\mathrm{pv}}(B) \geqslant 2/3|\mathbb{V}|)$$

Note that for genesis block Genesis we always have have $\#V_{\mathrm{obs}(v)}^{r,\mathrm{pv}}(B) = |\mathbb{V}|$.

Definition 26. *We say that round r is **unfinalizable,** if for all $B' \geqslant B_v^{r,\mathrm{pv}}$:*

$$|V_{\mathrm{obs}(v)}^{r,\mathrm{pc}}| - |V_{\mathrm{obs}(v)}^{r,\mathrm{pc}}(B')| \geqslant \frac{1}{3}|\mathbb{V}|$$

7.3.2 Voting Messages Specefication

Voting is done by means of broadcasting voting messages to the network. Validators inform their peers about the block finalized in round r by broadcasting finalization message (see Algorithm 3 for more details). These messages are specefied in this section.

Definition 27. *A vote casted by voter v should be broadcasted as a **message $M_v^{r,\mathrm{stage}}$** to the network by voter v with the following structure:*

$$M_v^{r,\mathrm{stage}} := \mathrm{Enc}_{\mathrm{SC}}(r, \mathrm{id}_{\mathbb{V}}, \mathrm{Enc}_{\mathrm{SC}}(\mathrm{stage}, V_v^{r,\mathrm{stage}}, \mathrm{Sig}_{\mathrm{ED25519}}(\mathrm{Enc}_{\mathrm{SC}}(\mathrm{stage}, V_v^{r,\mathrm{stage}}, r, V_{\mathrm{id}}), v_{\mathrm{id}})$$

Where:

r:	*round number*	*64 bit integer*
V_{id}:	*incremental change tracker counter*	*64 bit integer*
v_{id}:	*Ed25519 public key of v*	*4 byte array*
stage:	*0 if it is the pre-vote sub-round*	*1 byte*
	1 if it the pre-commit sub-round	

The **justification for block B in round r** of GRANDPA protocol defined $J^r(B)$ is a vector of pairs of type:

$$(V(B'), (\mathrm{Sign}_{v_i}^{r,\mathrm{pc}}(B'), v_{\mathrm{id}}))$$

in which either

$$B' > B$$

or $V_{v_i}^{r,\mathrm{pc}}(B')$ has equivocatory vote.

In all cases $\mathrm{Sign}_{v_i}^{r,\mathrm{pc}}(B')$ is the signature of voter v_i broadcasted during the pre-commit sub-round of round r.

Definition 28. GRANDPA *finalizing message for block B in round r* *represented as $M_v^{r,\mathbf{Fin}}(B)$ is a message broadcasted by voter v to the network indicating that voter v has finalized block B in round r. It has the following structure:*

$M_v^{r,\mathrm{stage}} := \mathrm{Enc}_{\mathrm{SC}}(r, V(B), J^r(B))$

7.3.3 Initiating the GRANDPA State

A validator needs to initiate its state and sync it with other validators, to be able to participate coherently in the voting process. In particular, considering that voting is happening in different rounds and each round of voting is assigned a unique sequential round number r_v, it needs to determine and set its round counter r in accordance with the current voting round r_n, which is currently undergoing in the network.

As instructed in Algortihm 2, whenever the membership of GRANDPA voters changes, r is set to 0 and V_{id} needs to be incremented.

Algorithm 2. JOIN-LEAVE-GRANDPA-VOTERS ($\mathcal{V}$)

$r \leftarrow 0$
$\mathcal{V}_{\text{id}} \leftarrow$ ReadState('AUTHORITY_SET_KEY')
$\mathcal{V}_{\text{id}} \leftarrow \mathcal{V}_{\text{id}} + 1$
EXECUTE-ONE-GRANDPA-ROUND(r)

Each voter should run Algorithm 5 to verify that a round is completable.

7.3.4 Voting Process in Round r

For each round r, an honest voter v must participate in the voting process by following Algorithm 3.

Algorithm 3. PLAY-GRANDPA-ROUND(r)

$t_{r,v} \leftarrow$ Time
primary $\leftarrow$ DERIVE-PRIMARY
if $v =$ primary:
  BROADCAST($M_v^{r-1,\text{fin}}()$)
else
  RECEIVE-MESSAGES(**until** Time $\geqslant t_{r,v} + 2 \times T$ **or** COMPLETABLE(r))
  $L \leftarrow$ RECEIVED-AS-FINAL() **or** BEST-FINAL-CANDIDATE(r-1):
  if RECEIVED($M_{v_{\text{primary}}}^{r,\text{pv}}(B)$) **and** $B_v^{r,\text{pv}} \geqslant B > L$:
    $N \leftarrow B$
  else
    $N \leftarrow B'$: $H_n(B') = \max\{H_n(B')$: $B' > L\}$
BROADCAST($M_v^{r,\text{pv}}(N)$)
RECEIVE-MESSAGES(**until** $B_v^{r,\text{pv}} \geqslant L$ **and** (Time $\geqslant t_{r,v} + 4 \times T$ **or** COMPLETABLE(r)))
BROADCAST($M_v^{r,\text{pc}}$(BEST-FINAL-CANDIDATE(r))
PLAY-GRANDPA-ROUND($r+1$)

Algorithm 4. BEST-FINAL-CANDIDATE(r)

$\mathcal{C} \leftarrow \{B' | B' \leqslant B_v^{r,\text{pv}}$: $|V_v^{r,\text{pc}}| - \#V_v^{r,\text{pc}}(B') \leqslant 1/3|\mathbb{V}|\}$
if $\mathcal{C} = \phi$:
  return ϕ

4 **else**
5 **return** $E \in \mathcal{C} : H_n(E) = \max\{H_n(B') : B' \in \mathcal{C}\}$

Algorithm 5. Completable(r)

1 $E \leftarrow$ Best-Final-Candidate(r)
1 **if** $E \neq \phi$
2 **return** True
3 **elif** r **is** unfinalizable:
4 **return** True
5 **else**
6 **return** False

Algorithm 6. FinalizeRound(r)

1 $L \leftarrow$ Last-Finalized-Block
2 $E \leftarrow$ Best-Final-Candidate(r)
2 **if** $E \geqslant L$ **and** $V_{\text{obs}(v)}^{r-1,\text{pc}}(E) > 2/3|\mathcal{V}|$
3 Last-Finalized-Block$\leftarrow B^{r,\text{pc}}$
4 **if** $M_v^{r,\text{Fin}}(E) \notin$ Received-Messages:
5 Broadcast($M_v^{r,\text{Fin}}(E)$)

8 Auxilary Encodings

8.1 SCALE Codec

Polkadot RE uses *Simple Concatenated Aggregate Little-Endian" (SCALE) codec* to encode byte arrays that provide canonical encoding and to produce consistent hash values across their implementation, including the Merkle hash proof for the State Storage.

Definition 29. *The **SCALE codec** for **Byte array** A such that*

$$A := b_1\, b_2 \ldots b_n$$

such that $n < 2^{536}$ is a byte array refered to $\text{Enc}_{\text{SC}}(A)$ *and defined as follows:*

$$\text{Enc}_{\text{SC}}(A) := \begin{cases} l_1\, b_1\, b_2 \ldots b_n & 0 \leqslant n < 2^6 \\ i_1\, i_2\, b_1 \ldots b_n & 2^6 \leqslant n < 2^{14} \\ j_1\, j_2\, j_3\, b_1 \ldots b_n & 2^{14} \leqslant n < 2^{30} \\ k_1\, k_2 \ldots k_m\, b_1 \ldots b_n & 2^{14} \leqslant n \end{cases}$$

in which:

$l_1^1\, l_1^0 = 00$
$i_1^1\, i_1^0 = 01$
$j_1^1\, j_1^0 = 10$
$k_1^1\, k_1^0 = 11$

and n is stored in $\mathrm{Enc_{SC}}(A)$ *in little-endian format in base-2 as follows:*

$$n = \begin{cases} l_1^7 \dots l_1^3\, l_1^2 & n < 2^6 \\ i_2^7 \dots i_2^0\, i_1^7 \dots i_1^2 & 2^6 \leqslant n < 2^{14} \\ j_4^7 \dots j_4^0\, j_3^7 \dots j_1^7 \dots j_1^2 & 2^{14} \leqslant n < 2^{30} \\ k_2 + k_3\, 2^8 + k_4\, 2^{2\cdot 8} + \dots + k_m\, 2^{(m-2)8} & 2^{30} \leqslant n \end{cases}$$

where:

$$m = l_1^7 \dots l_1^3\, l_1^2 + 4$$

Definition 30. *The* ***SCALE codec*** *for* ***Tuple*** T *such that:*

$$T := (A_1, ..., A_n)$$

Where A_i*'s are values of different types, is defined as:*

$$\mathrm{Enc_{SC}}(T) := \mathrm{Enc_{SC}}(A_1) | \mathrm{Enc_{SC}}(A_2) | ... | \mathrm{Enc_{SC}}(A_n)$$

8.2 Hex Encoding

Practically, it is more convenient and efficient to store and process data which is stored in a byte array. On the other hand, radix-16 tree keys are broken in 4-bits nibbles. Accordingly, we need a method to encode sequences of 4-bits nibbles into byte arrays canonically:

Definition 31. *Suppose that* $\mathrm{PK} = (k_1, ..., k_n)$ *is a sequence of nibbles, then*
$\mathrm{Enc_{HE}}(\mathrm{PK}) :=$

$$\begin{cases} \mathrm{Nibbles}_4 & \to \mathbb{B} \\ \mathrm{PK} = (k_1, ..., k_n) & \mapsto \begin{cases} (0, k_1 + 16\, k_2, ..., k_{2i-1} + 16\, k_{2i}) & n = 2\, i \\ (k_1, k_2 + 16\, k_3, ..., k_{2i} + 16\, k_{2i+1}) & n = 2\, i + 1 \end{cases} \end{cases}$$

8.3 Partial Key Encoding

Definition 32. *Let* N *be a node in the storage state trie with Partial Key* PK_N*. We define the* ***Partial key length encoding*** *function, formally referred to as* $\mathrm{Enc_{len}}(N)$ *as follows:*

$$\begin{array}{lll} \mathrm{Enc_{len}}(N) & & := \\ \mathrm{NodePrefix}(N) & & + \\ \begin{cases} (\|\mathrm{PK}_N\|) & \mathrm{NisleafNode} \;\&\; \|\mathrm{PK}_N\| < 127 \\ (127) || (\mathrm{LE}(\|\mathrm{PK}_N\| - 127)) & \mathrm{NisaleafNode} \;\&\; \|\mathrm{PK}_N\| \geqslant 127 \end{cases} & & \end{array}$$

where NodePrefix *function is defined in Definition 12.*

9 Genisis Block Specification

10 Predefined Storage keys

11 Runtime upgrade

12 Runtime API

Runtime API is a set of functions, which Polkadot RE exposes to Runtime to access Storage content and other external functions. Some of the functions are exposed to the runtime for efficiency reasons. Here is the list of the functions which Polkadot RE exposes to the runtime:

- `ext_blake2_256`
- `ext_blake2_256_enumerated_trie_root`
- `ext_chain_id`
- `ext_child_storage_root`
- `ext_clear_child_storage`
- `ext_clear_prefix`
- `ext_clear_storage`
- `ext_ed25519_verify`
- `ext_exists_child_storage`
- `ext_free`
- `ext_get_allocated_child_storage`
- `ext_get_allocated_storage`
- `ext_get_child_storage_into`
- `ext_get_storage_into`
- `ext_kill_child_storage`
- `ext_malloc`
- `ext_print_hex`
- `ext_print_num`
- `ext_print_utf8`

- ext_sandbox_instance_teardown
- ext_sandbox_instantiate
- ext_sandbox_invoke
- ext_sandbox_memory_get
- ext_sandbox_memory_new
- ext_sandbox_memory_set
- ext_sandbox_memory_teardown
- ext_set_child_storage
- ext_set_storage
- ext_storage_changes_root
- ext_storage_root
- ext_twox_128
- ext_twox_256
- ext_exists_storage