The Polkadot Host
Protocol Specification

Fabio Lama Florian Franzen Syed Hosseini

Web3 Foundation Web3 Foundation Web3 Foundation

May 14, 2021

Table of contents

Chapter 1
Background

1.1. Introduction

Formally, Polkadot is a replicated sharded state machine designed to resolve the scalability and interoperability among blockchains. In Polkadot vocabulary, shards are called *parachains* and Polkadot *relay chain* is part of the protocol ensuring global consensus among all the parachains. The Polkadot relay chain protocol, henceforward called *Polkadot protocol*, can itself be considered as a replicated state machine on its own. As such, the protocol can be specified by identifying the state machine and the replication strategy.

From a more technical point of view, the Polkadot protocol has been divided into two parts, the *Runtime* and the *Host*. The Runtime comprises the state transition logic for the Polkadot protocol and is designed and be upgradable via the consensus engine without requiring hard forks of the blockchain. The Polkadot Host provides the functionality for the Runtime to execute its state transition logic, such as an execution environment, I/O and consensus, shared mostly among peer-to-peer decentralized cryptographically-secured transaction systems, i.e. blockchains whose consensus system is based on the proof-of-stake. The Polkadot Host is planned to be stable and static for the lifetime duration of the Polkadot protocol.

With the current document, we aim to specify the Polkadot Host part of the Polkadot protocol as a replicated state machine. After defining the basic terms in Chapter 1, we proceed to specify the representation of a valid state of the Protocol in Chapter 2. In Chapter 3, we identify the protocol states, by explaining the Polkadot state transition and discussing the detail based on which the Polkadot Host interacts with the state transition function, i.e. Runtime. Following, we specify the input messages triggering the state transition and the system behaviour. In Chapter 4, we specify the communication protocols and network messages required for the Polkadot Host to communicate with other nodes in the network, such as exchanging blocks and consensus messages. In Chapter 6, we specify the consensus protocol, which is responsible for keeping all the replica in the same state. Finally, the initial state of the machine is identified and discussed in Appendix C. A Polkadot Host implementation which conforms with this part of the specification should successfully be able to sync its states with the Polkadot network.

1.2. Definitions and Conventions

Definition 1.1. *A **Discrete State Machine (DSM)** is a state transition system whose set of states and set of transitions are countable and admits a starting state. Formally, it is a tuple of*

$$(\Sigma, S, s_0, \delta)$$

where

- Σ *is the countable set of all possible transitions.*
- S *is a countable set of all possible states.*
- $s_0 \in S$ *is the initial state.*
- δ *is the state-transition function, known as **Runtime** in the Polkadot vocabulary, such that*

$$\delta: S \times \Sigma \to S$$

Definition 1.2. *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 1.3. ***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 1.4. *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 1.5. *We represent the concatenation of byte arrays $a := (a_0,...,a_n)$ and $b := (b_0,...,b_m)$ by:*

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

DEFINITION 1.6. *For a given byte b the* ***bitwise representation*** *of b is defined as*

$$b := b^7 \ldots b^0$$

where

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

DEFINITION 1.7. *Let I be a non-negative integer represented as $I = (B_n \ldots B_0)_{256}$ in base 256. By the* ***little-endian*** *representation of I, we refer to a byte array B_I such that:*

$$B_I := (b_0, b_1, \ldots, b_n) \quad \text{where} \quad b_i := B_i$$

Accordingly, we define the function Enc_{LE}*:*

$$\begin{array}{rccc} \text{Enc}_{\text{LE}}: & \mathbb{Z}^+ & \to & \mathbb{B} \\ & (B_n \ldots B_0)_{256} & \mapsto & (B_0, B_1, \ldots, B_n) \end{array}$$

DEFINITION 1.8. *By* **`UINT32`** *we refer to a non-negative integer stored in a byte array of length 4 using little-endian encoding format.*

DEFINITION 1.9. *A* ***blockchain*** *C is a directed path graph. Each node of the graph is called* ***Block*** *and indicated by* ***B****. The unique sink of C is called* ***Genesis Block****, and the source is called the* ***Head*** *of C. For any vertex (B_1, B_2) where $B_1 \to B_2$ we say B_2 is the* ***parent*** *of B_1 and we indicate it by*

$$B_2 := P(B_1)$$

DEFINITION 1.10. *By* ***UNIX time****, we refer to the unsigned, little-endian encoded 64-bit integer which stores the number of* ***milliseconds*** *that have elapsed since the Unix epoch, that is the time 00:00:00 UTC on 1 January 1970, minus leap seconds. Leap seconds are ignored, and every day is treated as if it contained exactly 86'400 seconds.*

1.2.1. Block Tree

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

Definition 1.11. *The **block tree** of a blockchain, denoted by* BT *is the union of all different versions of the blockchain observed by the Polkadot Host such that every block is a node in the graph and B_1 is connected to B_2 if B_1 is a parent of B_2.*

When a block in the block tree gets finalized, there is an opportunity to prune the block tree to free up resources into branches of blocks that do not contain all of the finalized blocks or those that can never be finalized in the blockchain. For a definition of finality, see Section 6.3.

Definition 1.12. *By **Pruned Block Tree**, denoted by* PBT*, we refer to a subtree of the block tree obtained by eliminating all branches which do not contain the most recent finalized blocks, as defined in Definition 6.39. By **pruning**, we refer to the procedure of* $\mathrm{BT} \leftarrow \mathrm{PBT}$*. When there is no risk of ambiguity and is safe to prune BT, we use* BT *to refer to* PBT*.*

Definition 1.13 gives the means to highlight various branches of the block tree.

Definition 1.13. *Let G be the root of the block tree and B be one of its nodes. By* **Chain(B)**, *we refer to the path graph from G to B in (P)*BT*. Conversely, for a chain $C=$*Chain(B)*, we define **the head of C** to be B, formally noted as $B :=$* Head(C)*. We define $|C|$, the length of C as a path graph. If B' is another node on* Chain(B)*, then by* SubChain(B', B) *we refer to the subgraph of* Chain(B) *path graph which contains both B and B' and by* $|$SubChain(B', B)$|$ *we refer to its length. Accordingly, $\mathbb{C}_{B'}((P)\mathrm{BT})$ is the set of all subchains of (P)*BT *rooted at B'. The set of all chains of (P)*BT*, $\mathbb{C}_G((P)\mathrm{BT})$ is denoted by $\mathbb{C}$ ((P)BT) or simply $\mathbb{C}$, for the sake of brevity.*

Definition 1.14. *We define the following complete order over $\mathbb{C}$ such that for $C_1, C_2 \in \mathbb{C}$ if $|C_1| \neq |C_2|$ we say $C_1 > C_2$ if and only if $|C_1| > |C_2|$.*

If $|C_1| = |C_2|$ we say $C_1 > C_2$ if and only if the block arrival time of Head(C_1) *is less than the block arrival time of* Head(C_2) *as defined in Definition 6.18. We define the* **Longest-Chain(BT)** *to be the maximum chain given by this order.*

Definition 1.15. Longest-Path(BT) *returns the path graph of (P)*BT *which is the longest among all paths in (P)*BT *and has the earliest block arrival time as defined in Definition 6.18.* Deepest-Leaf(BT) *returns the head of* Longest-Path(BT) *chain.*

Because every block in the blockchain contains a reference to its parent, it is easy to see that the block tree is de facto a tree. A block tree naturally imposes partial order relationships on the blocks as follows:

Definition 1.16. *We say **B is descendant of B'**, formally noted as **$B > B'$** if B is a descendant of B' in the block tree.*

□

Chapter 2

State Specification

2.1. State Storage and Storage Trie

For storing the state of the system, Polkadot Host implements a hash table storage where the keys are used to access each data entry. There is no assumption either on the size of the key nor on the size of the data stored under them, besides the fact that they are byte arrays with specific upper limits on their length. The limit is imposed by the encoding algorithms to store the key and the value in the storage trie.

2.1.1. Accessing System Storage

The Polkadot Host implements various functions to facilitate access to the system storage for the Runtime. See Section 3.1 for a an explaination of those functions. Here we formalize the access to the storage when it is being directly accessed by the Polkadot Host (in contrast to Polkadot runtime).

Definition 2.1. *The **StoredValue** function retrieves the value stored under a specific key in the state storage and is formally defined as :*

$$\text{StoredValue:} \quad \mathcal{K} \to \mathcal{V}$$
$$k \mapsto \begin{cases} v & \text{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{cases}$$

where $\mathcal{K} \subset \mathbb{B}$ and $\mathcal{V} \subset \mathbb{B}$ are respectively the set of all keys and values stored in the state storage.

2.1.2. The General Tree Structure

In order to ensure the integrity of the state of the system, the stored data needs to be re-arranged and hashed in a *Merkle radix-16 Tree*, which hereafter we refer to as the ***State Trie*** or simply as the ***Trie***. This rearrangment is necessary to be able to compute the Merkle hash of the whole or part of the state storage, consistently and efficiently at any given time.

The Trie is used to compute the *state root*, H_r, (see Definition 3.6), whose purpose is to authenticate the validity of the state database. Thus, the Polkadot Host 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 the Polkadot Host implementations.

The Trie is a *radix-16* tree as defined in Definition 1.3. 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.

When traversing the Trie to a specific node, its key can be reconstructed by concatenating the subsequences of the key which are stored either explicitly in the nodes on the path or implicitly in their position as a child of their parent.

To identify the node corresponding to a key value, k, first we need to encode k in a consistent with the Trie structure way. Because each node in the trie has at most 16 children, we represent the key as a sequence of 4-bit nibbles:

Definition 2.2. *For the purpose of labeling the branches of the state trie, the key k is encoded to k_{enc} using KeyEncode functions, formally referred to by KeyEncode(k):*

$$k_{\text{enc}} := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) := \text{KeyEncode}(k) \tag{2.1}$$

such that:

$$\text{KeyEncode}(k): \begin{cases} \mathbb{B} & \to \text{Nibbles}^4 \\ k := (b_1, \ldots, b_n) := & \mapsto (b_1^1, b_1^2, b_2^1, b_2^2, \ldots, b_n^1, b_n^2) \\ & := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) \end{cases}$$

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 big endian representations of* b_i*:*

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

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.

2.1.3. Trie Structure

In this subsection, we specify the structure of the nodes in the Trie as well as the Trie structure:

NOTATION 2.3. *By* $\mathcal{N}$*, we refer to the* ***set of the nodes of Polkadot state trie****. By* $N \in \mathcal{N}$*, we refer to an individual node in the trie.*

DEFINITION 2.4. *The State Trie is a radix-16 tree. Each node, N, in the Trie is identified with a unique key* k_N *such that:*

- k_N *is the shared prefix of the key of all the descendants of N in the Trie.*

and, at least one of the following statements holds:

- (k_N, v) *corresponds to an existing entry in the State Storage.*
- *N has more than one child.*

Conversely, if (k, v) *is an entry in the State Trie then there is a node* $N \in \mathcal{N}$ *such that* $k_N = k$*.*

NOTATION 2.5. *A* ***branch*** *node, formally referred to by* $\mathcal{N}_b$*, is a node which has one child or more. A branch node can have at most 16 children. A* ***leaf*** *node, referred to by* $\mathcal{N}_l$*, is a childless node. Accordingly:*

$$\mathcal{N}_b := \{N \in \mathcal{N} \mid N \text{ is a branch node}\}$$
$$\mathcal{N}_l := \{N \in \mathcal{N} \mid N \text{ is a leaf node}\}$$

For each node, part of k_N is built while the trie is traversed from root to N part of k_N is stored in N as formalized in Definition 2.6.

DEFINITION 2.6. *For any* $N \in \mathcal{N}$*, its key* k_N *is divided into an* ***aggregated prefix key****,* $\mathbf{pk_N^{Agr}}$*, aggregated by Algorithm 2.1 and a* ***partial key****,* $\mathbf{pk_N}$ *of length* $0 \leqslant l_{\text{pk}_N} \leqslant 65535$ *in nibbles such that:*

$$\text{pk}_N := (k_{\text{enc}_i}, \ldots, k_{\text{enc}_{i+l_{\text{pk}_N}}})$$

where pk_N *is a suffix subsequence of* k_N*; i is the length of* pk_N^{Agr} *in nibbles and so we have:*

$$\text{KeyEncode}(k_N) = \text{pk}_N^{\text{Agr}} || \text{pk}_N = (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{i-1}}, k_{\text{enc}_i}, k_{\text{enc}_{i+l_{\text{pk}_N}}})$$

Part of pk_N^{Agr} is explicitly stored in N's ancestors. Additionally, for each ancestor, a single nibble is implicitly derived while traversing from the ancestor to its child included in the traversal path using the Index_N function defined in Definition 2.7.

DEFINITION 2.7. *For* $N \in \mathcal{N}_b$ *and* N_c *child of N, we define* $\mathbf{Index_N}$ *function as:*

$$\text{Index}_N: \{N_c \in \mathcal{N} \mid N_c \text{ is a child of } N\} \to \text{Nibbles}_1^4$$
$$N_c \mapsto i$$

such that

$$k_{N_c} = k_N || i || \text{pk}_{N_c}$$

Assuming that P_N is the path (see Definition 1.2) from the Trie root to node N, Algorithm 2.1 rigorously demonstrates how to build pk_N^{Agr} while traversing P_N.

Algorithm 2.1. Aggregate-Key($P_N := (\text{TrieRoot} = N_1, \ldots, N_j = N)$)

$\text{pk}_N^{\text{Agr}} \leftarrow \phi$
$i \leftarrow 1$
while $(N_i \neq N)$
    $\text{pk}_N^{\text{Agr}} \leftarrow \text{pk}_N^{\text{Agr}} || \text{pk}_{N_i}$
    $\text{pk}_N^{\text{Agr}} \leftarrow \text{pk}_N^{\text{Agr}} || \text{Index}_{N_i}(N_{i+1})$
    $i \leftarrow i + 1$
$\text{pk}_N^{\text{Agr}} \leftarrow \text{pk}_N^{\text{Agr}} || \text{pk}_{N_i}$
return pk_N^{Agr}

Definition 2.8. *A node $N \in \mathcal{N}$ stores the **node value**, v_N, consisting of the following concatenated data:*

Node Header	Partial key	Node Subvalue

Formally noted as follows:

$$v_N := \text{Head}_N || \text{Enc}_{\text{HE}}(\text{pk}_N) || \text{sv}_N$$

where Head_N, pk_N, $\text{Enc}_{\text{nibbles}}$ *and* sv_N *are defined in Definitions 2.9,2.6, B.14 and 2.12, respectively.*

Definition 2.9. *The **node header** of node N, Head_N, consists of $l + 1 \geqslant 1$ bytes $\text{Head}_{N,1}, \ldots, \text{Head}_{N,l+1}$ such that:*

Node Type	pk length	pk length extra byte 1	pk key length extra byte 2	...	pk length extra byte l
$\text{Head}_{N,1}^{6-7}$	$\text{Head}_{N,1}^{0-5}$	$\text{Head}_{N,2}$	....	...	$\text{Head}_{N,l+1}$

In which $\text{Head}_{N,1}^{6-7}$, the two most significant bits of the first byte of Head_N are determined as follows:

$$\text{Head}_{N,1}^{6-7} := \begin{cases} 00 & \text{Special case} \\ 01 & \text{Leaf Node} \\ 10 & \text{Branch Node with } k_N \notin \mathcal{K} \\ 11 & \text{Branch Node with } k_N \in \mathcal{K} \end{cases}$$

where $\mathcal{K}$ is defined in Definition 2.1.

$\text{Head}_{N,1}^{0-5}$, the 6 least significant bits of the first byte of Head_N are defined to be:

$$\text{Head}_{N,1}^{0-5} := \begin{cases} \|\text{pk}_N\|_{\text{nib}} & \|\text{pk}_N\|_{\text{nib}} < 63 \\ 63 & \|\text{pk}_N\|_{\text{nib}} \geqslant 63 \end{cases}$$

In which $\|\mathbf{pk_N}\|_{\mathbf{nib}}$ is the length of pk_N in number nibbles. $\text{Head}_{N,2}, \ldots, \text{Head}_{N,l+1}$ bytes are determined by Algorithm 2.2.

Algorithm 2.2. Partial-Key-Length-Encoding($\text{Head}_{N,1}^{6-7}, \text{pk}_N$)

if $\|\text{pk}_N\|_{\text{nib}} \geqslant 2^{16}$
    return Error
$\text{Head}_{N,1} \leftarrow 64 \times \text{Head}_{N,1}^{6-7}$

if $\|\text{pk}_N\|_{\text{nib}} < 63$
    $\text{Head}_{N,1} \leftarrow \text{Head}_{N,1} + \|\text{pk}_N\|_{\text{nib}}$
    return Head_N
$\text{Head}_{N,1} \leftarrow \text{Head}_{N,1} + 63$
$l \leftarrow \|\text{pk}_N\|_{\text{nib}} - 63$
$i \leftarrow 2$
while $(l > 255)$
    $\text{Head}_{N,i} \leftarrow 255$
    $l \leftarrow l - 255$
    $i \leftarrow i + 1$
$\text{Head}_{N,i} \leftarrow l$
return Head_N

2.1.4. Merkle Proof

To prove the consistency of the state storage across the network and its modifications both efficiently and effectively, the Trie implements a Merkle tree structure. The hash value corresponding to each node needs to be computed rigorously to make the inter-implementation data integrity possible.

The Merkle value of each node should depend on the Merkle value of all its children as well as on its corresponding data in the state storage. This recursive dependancy is encompassed into the subvalue part of the node value which recursively depends on the Merkle value of its children and *children tries*. As Section 2.2.1 clarifies, the Merkle value of each **child trie** must be updated first before the final Polkadot state root can be calculated.

As mentioned in Section 2.1.2, the Trie is a Merkle tree. The hash function used for its Merkle structure is a variant of Blake2*b* hash function defined in Section A.2 . Specifically, the node value, v_N (see Definition 2.8) itself is presented as instead of its Blake2*b* hash when it occupies less space than the latter as it is defined in the following:

DEFINITION 2.10. *For a given node N, the **Merkle value** of N, denoted by $H(N)$ is defined as follows:*

$$H: \mathbb{B} \to \cup_{i \to 0}^{32} \mathbb{B}_{32}$$
$$H(N): \begin{cases} v_N & \|v_N\| < 32 \ \textit{and} \ N \neq R \\ \text{Blake2}b(v_N) & \|v_N\| \geqslant 32 \ \textit{or} \ N = R \end{cases}$$

*Where v_N is the node value of N defined in Definition 2.8 and R is the root of the Trie. The **Merkle hash** of the Trie is defined to be $H(R)$.*

The node value v_N depends on node subvalue sv_N which uses the auxilary function introduced in Definition 2.11 to encode and decode information stored in a branch node.

DEFINITION 2.11. *Suppose $N_b, N_c \in \mathcal{N}$ and N_c is a child of N_b. We define where bit $b_i := 1$ if N has a child with partial key i. Therefore, we define **ChildrenBitmap** functions as follows:*

$$\text{ChildrenBitmap}: \begin{array}{l} \mathcal{N}_b \to \mathbb{B}_2 \\ N \mapsto (b_{15}, \ldots, b_8, b_7, \ldots b_0)_2 \end{array}$$

where

$$b_i := \begin{cases} 1 & \exists N_c \in \mathcal{N}: k_{N_c} = k_{N_b} || i || \text{pk}_{N_c} \\ 0 & \textit{otherwise} \end{cases}$$

Having defined functions H and ChildrenBitmap, we are able to define the subvalue of a node as follows:

Definition 2.12. *For a given node N, the* ***subvalue*** *of N, formally referred to as sv_N, is determined as follows:*

$$\text{sv}_N := \begin{cases} \text{StoredValue}_{\text{SC}} \\ \text{Enc}_{\text{SC}}(\text{ChildrenBitmap}(N)) \| \text{StoredValue}_{\text{SC}} \| \text{Enc}_{\text{SC}}(H(N_{C_1})) \dots \text{Enc}_{\text{SC}}(H(N_{C_n})) \end{cases}$$

where the first variant is a leaf node and the second variant is a branch node.

$$\text{StoredValue}_{\text{SC}} := \begin{cases} \text{Enc}_{\text{SC}}(\text{StoredValue}(k_N)) & \text{if} \quad \text{StoredValue}(k_N) = v \\ \boldsymbol{\phi} & \text{if} \quad \text{StoredValue}(k_N) = \phi \end{cases}$$

$N_{C_1} \dots N_{C_n}$ with $n \leqslant 16$ are the children nodes of the branch node N and Enc_{SC} is the SCALE encoding defined in Definition B.1. StoredValue is defned in Definition 2.1. H and ChildrenBitmap(N) are defined in Definitions 2.10 and 2.11 respectively.

2.2. Child Storage

As clarified in Section 2.1, the Polkadot state storage implements a hash table for inserting and reading key-value entries. The child storage works the same way but is stored in a separate and isolated environment. Entries in the child storage are not directly accessible via querying the main state storage.

The Polkadot Host supports as many child storages as required by Runtime and identifies each separate child storage by its unique identifying key. Child storages are usually used in situations where Runtime deals with multiple instances of a certain type of objects such as Parachains or Smart Contracts. In such cases, the execution of the Runtime entry might result in generating repeated keys across multiple instances of certain objects. Even with repeated keys, all such instances of key-value pairs must be able to be stored within the Polkadot state.

In these situations, the child storage can be used to provide the isolation necessary to prevent any undesired interference between the state of separated instances. The Polkadot Host makes no assumptions about how child storages are used, but provides the functionality for it. This is described in more detail in the Host API, as described in Section 2.2.

2.2.1. Child Tries

The child trie specification is the same as the one described in Section 2.1.3. Child tries have their own isolated environment. Nonetheless, the main Polkadot state trie depends on them by storing a node (K_N, V_N) which corresponds to an individual child trie. Here, K_N is the child storage key associated to the child trie, and V_N is the Merkle value of its corresponding child trie computed according to the procedure described in Section 2.1.4

The Polkadot Host APIs as defined in 2.2 allows the Runtime to provide the key K_N in order to identify the child trie, followed by a second key in order to identify the value within that child trie. Every time a child trie is modified, the Merkle proof V_N of the child trie stored in the Polkadot state must be updated first. After that, the final Merkle proof of the Polkadot state can be computed. This mechanism provides a proof of the full Polkadot state including all its child states.

□

Chapter 3

State Transition

Like any transaction-based transition system, Polkadot's state is changed by executing an ordered set of instructions. These instructions are known as *extrinsics*. In Polkadot, the execution logic of the state-transition function is encapsulated in a Runtime as defined in Definition 1.1. For easy upgradability this Runtime is presented as a Wasm blob. Nonetheless, the Polkadot Host needs to be in constant interaction with the Runtime. The detail of such interaction is further described in Section 3.1.

In Section 3.2, we specify the procedure of the process where the extrinsics are submitted, pre-processed and validated by Runtime and queued to be applied to the current state.

To make state replication feasible, Polkadot journals and batches series of its extrinsics together into a structure known as a *block*, before propagating them to other nodes, similar to most other prominent distributed ledger systems. The specification of the Polkadot block as well as the process of verifying its validity are both explained in Section 3.3.

3.1. Interacting with the Runtime

The Runtime as defined in Definition 1.1 is the code implementing the logic of the chain. This code is decoupled from the Polkadot Host to make the the logic of the chain easily upgradable without the need to upgrade the Polkadot Host itself. The general procedure to interact with the Runtime is described in Algorithm 3.1.

Algorithm 3.1. Interact-With-Runtime(F: runtime entry to call,
$H_b(B)$: Block hash indicating the state at the end of B,
$A_1, A_2, \ldots, A_n$: arguments to be passed to the runtime entry)

1: $\mathcal{S}_B \leftarrow$ Set-State-At($H_b(B)$)
2: $A \leftarrow \text{Enc}_{\text{SC}}((A_1, \ldots, A_n))$
3: Call-Runtime-Entry($R_B, \mathcal{RE}_B, F, A, A_{\text{len}}$)

In this section, we describe the details upon which the Polkadot Host is interacting with the Runtime. In particular, Set-State-At and Call-Runtime-Entry procedures called in Algorithm 3.1 are explained in Notation 3.2 and Definition 3.10 respectively. R_B is the Runtime code loaded from $\mathcal{S}_B$, as described in Notation 3.1, and $\mathcal{RE}_B$ is the Polkadot Host API, as described in Notation D.1.

3.1.1. Loading the Runtime Code

The Polkadot Host 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:

$$b := 3A,63,6F,64,65$$

which is the ASCII byte representation of the string "`:code`" (see Section C). As a result of storing the Runtime as part of the state, the Runtime code itself becomes state sensitive and calls to Runtime can change the Runtime code itself. Therefore the Polkadot Host needs to always make sure to provide the Runtime corresponding to the state in which the entry has been called. Accordingly, we introduce the following notation to refer to the Runtime code at a specific state:

Notation 3.1. *By R_B, we refer to the Runtime code stored in the state storage at the end of the execution of block B.*

The initial Runtime code of the chain is provided as part of the genesis state (see Section C) and subsequent calls to the Runtime have the ability to, in turn, upgrade the Runtime by replacing this Wasm blob with the help of the storage API (see Section D).

3.1.2. Code Executor

The Polkadot Host executes the calls of Runtime entries inside a Wasm Virtual Machine (VM), which in turn provides the Runtime with access to the Polkadot Host API. This part of the Polkadot Host is referred to as the ***Executor***.

Definition 3.2 introduces the notation for calling the runtime entry which is used whenever an algorithm of the Polkadot Host needs to access the runtime.

Notation 3.2. *By*

$$\textsc{Call-Runtime-Entry}(R, \mathcal{RE}, \texttt{Runtime-Entry}, A, A_{\text{len}})$$

we refer to the task using the executor to invoke the `Runtime-Entry` *while passing an $A_1, \ldots, A_n$ argument to it and using the encoding described in Section 3.1.2.2.*

It is acceptable behavior that the Runtime panics during execution of a function in order to indicate an error. The Polkadot Host must be able to catch that panic and recover from it.

In this section, we specify the general setup for an Executor that calls into the Runtime. In Section E we specify the parameters and return values for each Runtime entry separately.

3.1.2.1. Memory Management

The Polkadot Host is responsible for managing the WASM heap memory starting at the exported symbol `__heap_base` as a part of implementing the allocator Host API (see Section D.9) and the same allocator should be used for any other heap allocation to be used by the Polkadot Runtime.

The size of the provided WASM memory should be based on the value of the `:heappages` storage key (an unsigned 64-bit integer), where each page has the size of 64KB. This memory shoule be made available to the Polkadot Runtime for import under the symbol name `memory`.

3.1.2.2. Sending Data to a Runtime Entry

In general, all data exchanged between the Polkadot Host and the Runtime is encoded using SCALE codec described in Section B.1. Therefore all runtime entries have the following identical Wasm function signatures:

```
(func $runtime_entry (param $data i32) (param $len i32) (result i64))
```

In each invocation of a Runtime entry, the argument(s) which are supposed to be sent to the entry, need to be SCALE encoded into a byte array B (see Definition B.1) and copied into a section of Wasm shared memory managed by the shared allocator described in Section 3.1.2.1.

When the Wasm method `runtime_entry`, corresponding to the entry, is invoked, two `i32` integers are passed as arguments. The first argument `data` is set to the memory adress of the byte array B in Wasm memory. The second argument `len` sets the length of the encoded data stored in B.

3.1.2.3. Receiving Data from a Runtime Entry

The value which is returned from the invocation is an `i64` integer, representing two consecutive `i32` integers in which the least significant one indicates the pointer to the offset of the result returned by the entry encoded in SCALE codec in the memory buffer. The most significant one provides the size of the blob.

3.1.2.4. Handling Runtimes update to the State

In order for the Runtime to carry on various tasks, it manipulates the current state by means of executing calls to various Polkadot Host APIs (see Appendix D). It is the duty of Host APIs to determine the context in which these changes should persist. For example, if Polkdot Host needs to validate a transaction using `TaggedTransactionQueue_validate_transaction` entry (see Section E.3.4.1), it needs to sandbox the changes to the state just for that Runtime call and prevent the global state of the system from being influence by the call to such a Runtime entry. This includes reverting the state of function calls which return errors or panic.

As a rule of thumb, any state changes resulting from Runtime enteries are not persistant with the exception of state changes resulting from calling `Core_execute_block` (see Section E.3.1.2) while Polkadot Host is importing a block (see Section 3.3.2).

For more information on managing multiple variant of state see Section 3.3.3.

3.2. Extrinsics

The block body consists of an array of extrinsics. In a broad sense, extrinsics are data from outside of the state which can trigger state transitions. This section describes extrinsics and their inclusion into blocks.

3.2.1. Preliminaries

The extrinsics are divided into two main categories defined as follows:

Definition 3.3. ***Transaction extrinsics*** *are extrinsics which are signed using either of the key types described in section A.5 and broadcasted between the nodes.* ***Inherent extrinsics*** *are unsigned extrinsics which are generated by Polkadot Host and only included in the blocks produced by the node itself. They are broadcasted as part of the produced blocks rather than being gossiped as individual extrinsics.*

The Polkadot Host does not specify or limit the internals of each extrinsics and those are defined and dealt with by the Runtime (defined in Definition 1.1). From the Polkadot Host point of view, each extrinsics is simply a SCALE-encoded blob as defined in Section B.1.

3.2.2. Transactions

Transaction are submitted and exchanged through *transaction messages* (see Section 4.8.3). Upon receiving a Transactions message, the Polkadot Host decodes the SCALE-encoded blob and splits it into individually SCALE-encoded transactions.

Alternative transaction can be submitted to the host by offchain worker through the `ext_offchain_submit_transaction` Host API, defined in Section D.5.2.

Any new transaction should be submitted to the `validate_transaction` Runtime function, defined in Section E.3.4.1. This will allow the Polkadot Host to check the validity of the received transaction against the current stat and if it should be gossiped to other peers. If `validate_transaction` considers the submitted transaction as valid, the Polkadot Host should store it for inclusion in future blocks. The whole process of handeling new transactions is described in more detail by Algorithm 3.2.

Additionally valid transactions that are supposed to be gossiped are propagated to connected peers of the Polkadot Host. While doing so the Polkadot Host should keep track of peers already aware of each transaction. This includes peers which have already gossiped the transaction to the node as well as those to whom the transaction has already been sent. This behavior is mandated to avoid resending duplicates and unnecessarily overloading the network. To that aim, the Polkadot Host should keep a *transaction pool* and a *transaction queue* defined as follows:

Definition 3.4. *The* ***Transaction Queue*** *of a block producer node, formally referred to as* TQ *is a data structure which stores the transactions ready to be included in a block sorted according to their priorities (Definition 4.8.3). The* ***Transaction Pool****, formally referred to as* TP*, is a hash table in which the Polkadot Host keeps the list of all valid transactions not in the transaction queue.*

Algorithm 3.2 updates the transaction pool and the transaction queue according to the received message:

Algorithm 3.2. Validate-Transactions-and-Store(M_T: Transaction Message)

$L \leftarrow \text{Dec}_{\text{SC}}(M_T)$ [not all tx are received via M_T]
for T **in** L **such that** $E \notin$ TQ **and** $E \notin$ TP:
  $B_d \leftarrow$ Head(Longest-Chain((BT))
  $N \leftarrow H_n(B_d)$
  $R \leftarrow$ Call-Runtime-Entry(
    `TaggedTransactionQueue_validate_transaction`, N, T
)
  if R indicates E is Valid:
    if Requires(R)$\subset$
      $\bigcup_{\forall T \in (\text{TQ})}$ Provided-Tags(T) $\cup$ $\bigcup_{i<d, \forall T, T \in B_i}$ Provided-Tags(T):
      Insert-At(TQ, T, Requires(R), Priority(R))
    else
      Add-To(TP,T)
    Maintain-Transaction-Pool
    if ShouldPropagate(R):
      Propagate(T)

In which

- Dec_{Sc} decodes the SCALE encoded message.
- Longest-Chain is defined in Definition 1.14.
- `TaggedTransactionQueue_validate_transaction` is a Runtime entry specified in Section E.3.4.1 and Requires(R), Priority(R) and Propagate(R) refer to the corresponding fields in the tuple returned by the entry when it deems that T is valid.
- Provided-Tags(T) is the list of tags that transaction T provides. The Polkadot Host needs to keep track of tags that transaction T provides as well as requires after validating it.
- Insert-At(TQ, T, Requires(R), Priority(R)) places T into TQ approperietly such that the transactions providing the tags which T requires or have higher priority than T are ahead of T.
- Maintain-Transaction-Pool is described in Algorithm 3.3.
- ShouldPropagate indictes whether the transaction should be propagated based on the `Propagate` field in the `ValidTransaction` type as defined in Definition E.13, which is returned by `TaggedTransactionQueue_validate_transaction`.
- Propagate(T) sends T to all connected peers of the Polkadot Host who are not already aware of T.

Algorithm 3.3. Maintain-Transaction-Pool

[This is scaning the pool for ready transactions and moving them to the TQ and dropping transactions which are not valid]

3.2.3. Inherents

Inherents are unsigned extrinsic inserted into a block by the block author and as a result are not stored in the transaction pool or gossiped across the network. Instead they are generated by the Polkadot Host by passing the required inherent data, as listed in Table 3.1, to the Runtime method `BlockBuilder_inherent_extrinsics` (Section E.3.3.3). The then returned extrinsics should be included in the current block as explained in Algorithm 6.7. [define uncles]

Identifier	Value type	Description
timstap0	u64	Unix epoch time in number of milliseconds
uncles00	array of block headers	Provides a list of potential uncle block headers[3.6] for a given block

Table 3.1. List of inherent data

Definition 3.5. Inherent-Data *is a hashtable (Definition B.8), an array of key-value pairs consisting of the inherent 8-byte identifier and its value, representing the totality of inherent extrinsics included in each block. The entries of this hash table which are listed in Table 3.1 are collected or generated by the Polkadot Host and then handed to the Runtime for inclusion as dercribed in Algorithm 6.7.*

3.3. State Replication

Polkadot nodes replicate each other's state by syncing the history of the extrinsics. This, however, is only practical if a large set of transactions are batched and synced at the time. The structure in which the transactions are journaled and propagated is known as a block (of extrinsics) which is specified in Section 3.3.1. Like any other replicated state machines, state inconsistency can occure between Polkadot replicas. Section 3.3.3 is giving an overview of how a Polkadot Host node manages multiple variants of the state.

3.3.1. Block Format

A Polkadot block consists a *block header* (Section 3.3.1.1) and a *block body* (Section 3.3.1.3). The *block body* in turn is made up out of a list of *extrinsics*, which represent the generalization of the concept of *transactions*. *Extrinsics* can contain any set of external data the underlying chain wishes to validate and track.

3.3.1.1. Block Header

The block header is designed to be minimalistic in order to allow efficienct handeling by light clients. It is defined formally as follows:

Definition 3.6. *The* ***header of block B***, $\mathbf{Head(B)}$ *is a 5-tuple containing the following elements:*

- ***parent_hash:*** *formally indicated as* $\boldsymbol{H_p}$*, is the 32-byte Blake2b hash (Section A.2) of the SCALE encoded parent block header as defined in Definition 3.8.*
- ***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 state 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 field which is reserved for the Runtime to validate the integrity of the extrinsics composing the block body. For example, it can hold the root hash of the Merkle trie which stores an ordered list of the extrinsics being validated in this block. The* extrinsics_root *is set by the runtime and its value is opaque to the Polkadot Host. 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 as well as consensus-related data including the block signature. This field is indicated as* $\boldsymbol{H_d}$ *and its detailed format is defined in Definition 3.7*

Definition 3.7. *The header* ***digest*** *of block B formally referred to by* $\boldsymbol{H_d(B)}$ *is an array of* ***digest items*** H_d^i*'s , known as digest items of varying data type (see Definition B.4) such that*

$$H_d(B) := H_d^1, \ldots, H_d^n$$

where each digest item can hold one of the type described in Table 3.2:

Type Id	*Type name*	*sub-components*
2	*Changes trie root*	$\mathbb{B}_{32}$
6	*Pre-Runtime*	$E_{\text{id}}, \mathbb{B}$
4	*Consensus Message*	$E_{\text{id}}, \mathbb{B}$
5	*Seal*	$E_{\text{id}}, \mathbb{B}$

Table 3.2. *The detail of the varying type that a digest item can hold.*

Where E_{id} is the unique consensus engine identifier defined in Section 6.3 and

- ***Changes trie root*** *contains the root of the Changes Trie at block B, as described in Section 3.3.4. Note that this is future-reserved and currently* ***not*** *used in Polkadot.*
- ***Pre-runtime*** *digest items represent messages from a consensus engine to the Runtime (e.g. see Definition 6.20).*
- ***Consensus Message*** *digest items represent messages from the Runtime to the consensus engine (see Section 6.1.2).*
- ***Seal*** *is the data produced by the consensus engine and proving the authorship of the block producer. In particular, the Seal digest item must be the last item in the digest array and must be stripped off by the Polkadot Host before the block is submitted to any Runtime function including for validation. The Seal must be added back to the digest afterward. The detail of the Seal digest item is laid out in Definition 6.21.*

Definition 3.8. *The* ***block header hash of block B***, $\boldsymbol{H_h(B)}$, *is the hash of the header of block B encoded by SCALE codec:*

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

3.3.1.2. Justified Block Header

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

- **block_header** the complete block header as defined in Section ? and denoted by Head(B).
- **justification**: as defined by the consensus specification indicated by Just(B) as defined in Definition 6.29.
- **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 256-bit.

3.3.1.3. Block Body

The Block Body consists of an sequence of extrinsics, each encoded as a byte array. The content of an extrinsic is completely opaque to the Polkadot Host. As such, from the point of the Polkadot Host, and is simply a SCALE encoded array of byte arrays. Formally:

Definition 3.9. *The* ***body of Block*** *B represented as* $\mathbf{Body(B)}$ *is defined to be*

$$\text{Body}(B) := \text{Enc}_{\text{SC}}(E_1, \ldots, E_n)$$

Where each $E_i \in \mathbb{B}$ is a SCALE encoded extrinsic.

3.3.2. Importing and Validating Block

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

New blocks can be received by the Polkadot Host via other peers (see Section 4.8.2) or from the Host's own consensus engine (see Section 6.2). Both the Runtime and the Polkadot Host then need to work together to assure block validity. A block is deemed valid if the block author had authorship rights for the slot in which the block was produce as well as if the transactions in the block constitute a valid transition of states. The former criterion is validated by the Polkadot Host according to the block production consensus protocol. The latter can be verified by the Polkadot Host invoking `Core_execute_block` entry into the Runtime as defined in section E.3.1.2 as a part of the validation process. Any state changes created by this function on successful execution are persisted.

The Polkadot Host implements the following procedure to assure the validity of the block:

Algorithm 3.4. Import-and-Validate-Block(B, Just(B))

```
1:  Set-Storage-State-At(P(B))
2:  if Just(B) ≠ ∅
3:      Verify-Block-Justification(B, Just(B))
4:      if B is Finalized and P(B) is not Finalized
5:          Mark-as-Final(P(B))
6:  if H_p(B) ∉ PBT
7:      return
8:  Verify-Authorship-Right(Head(B))
9:  B ← Remove-Seal(B)
10: R ← Call-Runtime-Entry(Core_execute_block, B)
11: B ← Add-Seal(B)
12: if R = True
13:     Persist-State
```

In which

- Remove-Seal removes the Seal digest from the block as described in Definition 3.7 before submitting it to the Runtime.
- Add-Seal adds the Seal digest back to the block as described in Definition 3.7 for later propagation.
- Persist-State implies the persistence of any state changes created by `Core_execute_block` on successful execution.
- PBT is the pruned block tree defined in Definition 1.11.
- Verify-Authorship-Right is part of the block production consensus protocol and is described in Algorithm 6.5.
- *Finalized block* and *finality* is defined in Section 6.3.

3.3.3. Managaing Multiple Variants of State

Unless a node is committed to only update its state according to the finalized block (See Definition 6.39), it is inevitable for the node to store multiple variants of the state (one for each block). This is, for example, necessary for nodes participating in the block production and finalization.

While the state trie structure described in Section 2.1.3 facilitates and optimizes storing and switching between multiple variants of the state storage, the Polkadot Host does not specify how a node is required to accomplish this task. Instead, the Polkadot Host is required to implement Set-State-At operation which behaves as defined in Definition 3.10:

Definition 3.10. *The function*

Set-State-At(B)

in which B is a block in the block tree (See Definition 1.11), sets the content of state storage equal to the resulting state of executing all extrinsics contained in the branch of the block tree from genesis till block B including those recorded in Block B.

For the definition of the state storage see Section 2.1.

3.3.4. Changes Trie

[NOTE: Changes Tries are still work-in-progress and are currently **not** used in Polkadot. Additionally, the implementation of Changes Tries might change considerably.]

Polkadot focuses on light client friendliness and therefore implements functionalities that allows identifying changes in the state of the blockchain without the requirement to search through the entire chain. The **Changes Trie** is a radix-16 tree data structure as defined in Definition 1.3 and maintained by the Polkadot Host. It stores different types of storage changes made by each individual block separately.

The primary method for generating the Changes Trie is provided to the Runtime with the `ext_storage_changes_root` Host API as described in Section D.1.9. The Runtime calls that function shortly before finalizing the block, the Polkadot Host must then generate the Changes Trie based on the storage changes which occured during block production or execution. In order to provide this API function, it is imperative that the Polkadot Host implements a mechanism to keep track of the changes created by individual blocks, as mentioned in Sections 2.1 and 3.3.3.

The Changes Trie stores three different types of changes.

Definition 3.11. *The **inserted key-value pair stored in the nodes of Changes Trie** is formally defined as:*

$$(K_C, V_C)$$

Where K_C is a SCALE-encoded Tuple

$$\text{Enc}_{\text{sc}}\big(\big(\text{Type}_{V_C}, H_i(B_i), K\big)\big)$$

and

$$V_C = \text{Enc}_{\text{SC}}(C_{\text{value}})$$

is a SCALE encoded byte array.

Furthermore, K represents the changed storage key, $H_i(B_i)$ refers to the block number at which this key is inserted into the Changes Trie (See Definition 3.6) and Type_{V_C} is an index defining the type C_{Value} according to Table 3.3.

Type	Description	C_{Value}
1	list of extrinsics indices (section 3.3.4.1) where e_i refers to the index of the extrinsic within the block	$\{e_i, \ldots, e_k\}$
2	list of block numbers (section 3.3.4.2)	$\{H_i(B_k), \ldots, H_i(B_m)\}$
3	Child Changes Trie (section 3.3.4.3)	$H_r(\text{Child-Changes-Trie})$

Table 3.3. Possible types of keys of mappings in the Changes Trie

The Changes Trie itself is not part of the block, but a separately maintained database by the Polkadot Host. The Merkle proof of the Changes Trie must be included in the block digest as described in Definition 3.7 and gets calculated as described in section 2.1.4. The root calculation only considers pairs which were generated on the individual block and does not consider pairs which were generated at previous blocks.

[This seperately maintained database by the Polkadot Host is intended to be used by "proof servers", where its implementation and behavior has not been fully defined yet. This is considered future-reserved]

As clarified in the individual sections of each type, not all of those types get generated on every block. But if conditions apply, all those different types of pairs get inserted into the same Changes Trie, therefore only one Changes Trie Root gets generated for each block.

3.3.4.1. Key to extrinsics pairs

This key-value pair stores changes which occure in an individual block. Its value is a SCALE encoded array containing the indices of the extrinsics that caused any changes to the specified key. The key-value pair is defined as (clarified in section 3.3.4):

$$(1, H_i(B_i), K) \rightarrow \{e_i, \ldots, e_k\}$$

The indices are unsigned 32-bit integers and their values are based on the order in which each extrinsics appears in the block (indexing starts at 0). The Polkadot Host generates those pairs for every changed key on each and every block. Child storages have their own Changes Trie, as described in section 3.3.4.3.

[clarify special key value of 0xffffffff]

3.3.4.2. Key to block pairs

This key-value pair stores changes which occured in a certain range of blocks. Its value is a SCALE encoded array containing block numbers in which extrinsics caused any changes to the specified key. The key-value pair is defined as (clarified in section 3.3.4):

$$(2, H_i(B_i), K) \rightarrow \{H_i(B_k), \ldots, H_i(B_m)\}$$

The block numbers are represented as unsigned 32-bit integers. There are multiple "levels" of those pairs, and the Polkadot Host does **not** generate those pairs on every block. The genesis state contains the key `:changes_trie` where its unsigned 64-bit value is a tuple of two 32-bit integers:

- `interval` - The interval (in blocks) at which those pairs should be created. If this value is less or equal to 1 it means that those pairs are not created at all.
- `levels` - The maximum number of "levels" in the hierarchy. If this value is 0 it means that those pairs are not created at all.

For each level from 1 to `levels`, the Polkadot Host creates those pairs on every $\texttt{interval}^{\texttt{level}}$-nth block, formally applied as:

Algorithm 3.5. Key-To-Block-Pairs(B_i, I: interval, L: levels

for each $l \in \{1, \ldots, L\}$
3. if $H_i(B_i) = I^l$
4. Insert-Blocks($H_i(B_i)$, I^l)

- B_i implies the block at which those pairs gets inserted into the Changes Trie.
- Insert-Blocks - Inserts every block number within the range $H_i(B_i) - I^l + 1$ to $H_i(B_i)$ in which any extrinsic changed the specified key.

For example, let's say `interval` is set at `4` and `levels` is set at `3`. This means there are now three levels which get generated at three different occurences:

1. **Level 1** - Those pairs are generated at every $\mathbf{4^1}$-nth block, where the pair value contains the block numbers of every block that changed the specified storage key. This level only considers block numbers of the last four ($=4^1$) blocks.
    - Example: this level occurs at block 4, 8, 12, 16, 32, etc.
2. **Level 2** - Those pairs are generated at every $\mathbf{4^2}$-nth block, where the pair value contains the block numbers of every block that changed the specified storage key. This level only considers block numbers of the last 16 ($=4^2$) blocks.
    - Example: this level occurs at block 16, 32, 64, 128, 256, etc.

3. **Level 3** - Those pairs are generated at every $\mathbf{4^3}$-nth block, where the pair value contains the block numbers of every block that changed the specified storage key. this level only considers block number of the last 64 ($=4^3$) blocks.
    - Example: this level occurs at block 64, 128, 196, 256, 320, etc.

3.3.4.3. Key to Child Changes Trie pairs

The Polkadot Host generates a separate Changes Trie for each child storage, using the same behavior and implementation as describe in section 3.3.4.1. Additionally, the changed child storage key gets inserted into the primary, non-Child Changes Trie where its value is a SCALE encoded byte array containing the Merkle root of the Child Changes Trie. The key-value pair is defined as:

$$(3, H_i(B_i), K) \rightarrow H_r(\text{Child-Changes-Trie})$$

The Polkadot Host creates those pairs for every changes child key for each and every block.

□

Chapter 4

Networking

Chapter Status: This chapter in its current form is incomplete and considered work in progress. Authors appreciate receiving request for clarification or any reports regarding deviation from the current Polkadot network protocol. This can be done through filing an issue in Polkadot Specification repository [?].

4.1. Introduction

The Polkadot network is decentralized and does not rely on any central authority or entity for achieving its fullest potential of provided functionality. The networking protocol is based on a family of open protocols, including protocol implemented `libp2p` e.g. the distributed Kademlia hash table which is used for peer discovery.

This chapter walks through the behaviour of the networking implementation of the Polkadot Host and defines the network messages. The implementation details of the `libp2p` protocols used are specified in external sources as described in Section 4.2.

4.2. External Documentation

Complete specification of the Polkadot networking protocol relies on the following external protocols:

- libp2p - `libp2p` is a modular peer-to-peer networking stack composed of many modules and different parts. `libp2p` includes the multiplexing protocols `mplex` and `yamux`.
- libp2p addressing - The Polkadot Host uses the `libp2p` addressing system to identify and connect to peers.
- Kademlia - `Kademlia` is a distributed hash table for decentralized peer-to-peer networks. The Polkadot Host uses Kademlia for peer discovery.
- Noise - The Noise protocol is a framework for building cryptographic protocols. The Polkadot Host uses Noise to establish the encryption layer to remote peers.
- mplex - `mplex` is a multiplexing protocol developed by `libp2p`. The protocol allows dividing a connection to a peer into multiple substreams, each substream serving a specific purpose. Generally, Polkadot Host implementers are encouraged to prioritize implementing `yamux`, since it is the de-facto standard in Polkadot. `mplex` is only required to communicate with js-lip2p.
- yamux - `yamux` is a multiplexing protocol like `mplex` and developed by HashiCorp. It is the de-facto standard for the Polkadot Host. This protocol should be prioritized over `mplex`. Section 4.7 describes the subprotocol in more detail.
- Protocol Buffers - Protocol Buffers is a language-neutral, platform-neutral mechanism for serializing structured data and is developed by Google. The Polkadot Host uses Protocol Buffers to serialize specific messages, as clarified in Section 4.8.

4.3. Node Identities

Each Polkadot Host node maintains an ED25519 key pair which is used to identify the node. The public key is shared with the rest of the network allowing the nodes to establish secure communication channels.

Each node must have its own unique ED25519 key pair. When two or more nodes use the same key, the network will interpret those nodes as a single node, which will result in undefined behaviour and can result in equivocation. Furthermore, the node's `PeerId` as defined in Definition 4.1 is derived from its public key. `PeerId` (4.1) is used to identify each node when they are discovered in the course of the discovery mechanism described in Section 4.4.

DEFINITION 4.1. *The Polkadot node's* `PeerId`*, formally referred to as* P_{id}*, is derived from the ED25519 public key and is structured as defined in the libp2p specification (*`https://docs.libp2p.io/concepts/peer-id/`*).*

4.4. Discovery mechanism

The Polkadot Host uses various mechanisms to find peers within the network, to establish and maintain a list of peers and to share that list with other peers from the network as follows:

- **Bootstrap nodes** are hard-coded node identities and addresses provided by the genesis state specification as described in Appendix C.
- **mDNS** is a protocol that performs a broadcast to the local network. Nodes that might be listening can respond to the broadcast. The libp2p mDNS specification defines this process in more detail. This protocol is an optional implementation detail for Polkadot Host implementers and is not required to participate in the Polkadot network.
- **Kademlia requests** invoking Kademlia `FIND_NODE` requests, where nodes respond with their list of available peers. Kademlia requests are performed on a specific substream as described in Section 4.7.

4.5. Connection establishment

Polkadot nodes connect to peers by establishing a TCP connection. Once established, the node initiates a handshake with the remote peers on the encryption layer. An additional layer on top of the encryption layer, known as the multiplexing layer, allows a connection to be split into substreams, as described by the yamux specification, either by the local or remote node.

The Polkadot node supports two types of substream protocols. Section 4.7 describes the usage of each type in more detail:

- **Request-Response substreams**: After the protocol is negotiated by the multiplexing layer, the initiator sends a single message containing a request. The responder then sends a response, after which the substream is then immediately closed. The requests and responses are prefixed with their LEB128 encoded length.
- **Notification substreams**. After the protocol is negotiated, the initiator sends a single handshake message. The responder can then either accept the substream by sending its own handshake or reject it by closing the substream. After the substream has been accepted, the initiator can send an unbound number of individual messages. The responder keeps its sending side of the substream open, despite not sending anything anymore, and can later close it in order to signal to the initiator that it no longer wishes to communicate.

  Handshakes and messages are prefixed with their LEB128 encoded lengths. A handshake can be empty, in which case the length prefix would be `0`.

Connections are established by using the following protocols:

- `/noise` - a protocol that is announced when a connection to a peer is established.
- `/multistream/1.0.0` - a protocol that is announced when negotiating an encryption protocol or a substream.
- `/yamux/1.0.0` - a protocol used during the `mplex` or `yamux` negotiation. See Section 4.7 for more information.

The Polkadot Host can establish a connection with any peer of which it knows the address. The Polkadot Host supports multiple networking protocols:

- **TCP/IP** with addresses in the form of `/ip4/1.2.3.4/tcp/` to establish a TCP connection and negotiate encryption and a multiplexing layer.

- **Websockets** with addresses in the form of `/ip4/1.2.3.4/ws/` to establish a TCP connection and negotiate the Websocket protocol within the connection. Additionally, encryption and multiplexing layer is negotiated within the WebSocket connection.
- **DNS** addresses in form of `/dns/website.domain/tcp/` and `/dns/website.domain/ws/`.

The addressing system is described in the libp2p addressing specification. After a base-layer protocol is established, the Polkadot Host will apply the Noise protocol to establish the encryption layer as described in Section 4.6.

4.6. Encryption Layer

Polkadot protocol uses the `libp2p` Noise framework to build an encryption protocol. The Noise protocol is a framework for building encryption protocols. `libp2p` utilizes that protocol for establishing encrypted communication channels. Refer to the libp2p Secure Channel Handshake specification for a detailed description.

Polkadot nodes use the XX handshake pattern to establish a connection between peers. The three following steps are required to complete the handshake process:

1. The initiator generates a keypair and sends the public key to the responder. The Noise specification and the libp2p PeerId specification describe keypairs in more detail.
2. The responder generates its own key pair and sends its public key back to the initiator. After that, the responder derives a shared secret and uses it to encrypt all further communication. The responder now sends its static Noise public key (which may change anytime and does not need to be persisted on disk), its `libp2p` public key and a signature of the static Noise public key signed with the `libp2p` public key.
3. The initiator derives a shared secret and uses it to encrypt all further communication. It also sends its static Noise public key, `libp2p` public key and signature to the responder.

After these three steps, both the initiator and responder derive a new shared secret using the static and session-defined Noise keys, which are used to encrypt all further communication.

4.7. Protocols and Substreams

After the node establishes a connection with a peer, the use of multiplexing allows the Polkadot Host to open substreams. `libp2p` uses the `mplex` protocol or the `yamux` protocol to manage substreams and to allow the negotiation of *application-specific protocols*, where each protocol serves a specific utility.

The Polkadot Host uses multiple substreams whose usage depends on a specific purpose. Each substream is either a *Request-Response substream* or a *Notification substream*, as described in Section 4.5.

- `/ipfs/ping/1.0.0` - Open a standardized `libp2p` substream to a peer and initialize a ping to verify if a connection is still alive. If the peer does not respond, the connection is dropped. This is a *Request-Response substream*.

  Further specification and reference implementation are available in the libp2p documentation.
- `/ipfs/id/1.0.0` - Open a standardized `libp2p` substream to a peer to ask for information about that peer. This is a *Request-Response substream*.

  Further specification and reference implementation are available in the libp2p documentation.
- `/dot/kad` - Open a standardized substream for Kademlia `FIND_NODE` requests. This is a *Request-Response substream*, as defined by the `libp2p` standard.

  Further specification and reference implementation are available on Wikipedia respectively the golang Github repository.
- `/dot/light/2` - a request and response protocol that allows a light client to request information about the state. This is a *Request-Response substream*.

[light client messages are currently not documented]

- `/dot/block-announces/1` - a substream/notification protocol which sends blocks to connected peers. This is a *Notification substream*.
  The messages are specified in Section 4.8.1.
- `/dot/sync/2` - a request and response protocol that allows the Polkadot Host to perform information about blocks. This is a *Request-Response substream*.
  The messages are specified in Section 4.8.2.
- `/dot/transactions/1` - a substream/notification protocol which sends transactions to connected peers. This is a *Notification substream*.
  The messages are specified in Section 4.8.3.
- `/paritytech/grandpa/1` - a substream/notification protocol that sends GRANDPA votes to connected peers. This is a *Notification substream*.
  The messages are specified in Section 4.8.4.
  [This substream will change in the future. See issue #7252.]
- `/paritytech/beefy/1` - a substream/notification protocol which sends signed BEEFY statements, as described in Section 6.5, to connected peers. This is a *Notification* substream.
  The messages are specified in Section 4.8.4.5.

Note: the `/dot/` prefixes on those substreams are known as protocol identifiers and are used to segregate communications to specific networks. This prevents any interference with other networks. `/dot/` is used exclusively for Polkadot. Kusama, for example, uses the `/ksmcc3/` protocol identifier.

4.8. Network Messages

The Polkadot Host must actively communicate with the network in order to participate in the validation process or act as a full node.

Note: The Polkadot network originally only used SCALE encoding for all message formats. Meanwhile, Protobuf has been adopted for certain messages. The encoding of each message is explicitly mentioned in their corresponding definition. Encoding and message formats are subject to change.

4.8.1. Announcing blocks

When the node creates or receives a new block, it must be announced to the network. Other nodes within the network will track this announcement and can request information about this block. The mechanism for tracking announcements and requesting the required data is implementation-specific.

Block announcements, requests and responses are sent over the `/dot/block-announces/1` substream as defined in Definition 4.2.

Definition 4.2. *The `BlockAnnounceHandshake` initializes a substream to a remote peer. Once established, all `BlockAnnounce` messages, as defined in Definition 4.3, and created by the node are sent to the `/dot/block-announces/1` substream.*

The `BlockAnnounceHandshake` is a SCALE-encoded structure of the following format:

$$\text{BA}_h \;=\; \text{Enc}_{\text{SC}}(R, N_B, h_B, h_G)$$

where:

$$R \;=\; \begin{cases} 1 & \textit{The node is a full node} \\ 2 & \textit{The node is a light client} \\ 4 & \textit{The node is a validator} \end{cases}$$

$$\begin{aligned} N_B &= \textit{Best block number according to the node} \\ h_B &= \textit{Best block hash according to the node} \\ h_G &= \textit{Genesis block hash according to the node} \end{aligned}$$

Definition 4.3. *The* `BlockAnnounce` *message is sent to the specified substream and indicates to remote peers that the node has either created or received a new block.*

The `BlockAnnounce` *message is a SCALE-encoded structure of the following format:*

$$\text{BA} = \text{Enc}_{\text{SC}}(\text{Head}(B), b)$$

where:

$$\begin{aligned} \text{Head}(B) &= \textit{Header of the announced block} \\ b &= \begin{cases} 0 & \textit{Is not part of the best chain} \\ 1 & \textit{Is the best block according to the node} \end{cases} \end{aligned}$$

4.8.2. Requesting blocks

Block requests can be used to retrieve a range of blocks from peers. Those messages are sent over the `/dot/sync/2` substream.

Definition 4.4. *The* `BlockRequest` *message is a Protobuf serialized structure of the following format:*

Type	***Id***	***Description***	***Value***
uint32	*1*	*Bits of block data to request*	B_f
oneof		*Start from this block*	B_S
bytes	*4*	*End at this block (optional)*	B_e
Direction	*5*	*Sequence direction*	
uint32	*6*	*Maximum amount (optional)*	B_m

Table 4.1. `BlockRequest` *Protobuf message.*

where

- B_f *indicates all the fields that should be included in the request. Its* ***big-endian*** *encoded bitmask that applies to all desired fields with bitwise OR operations. For example, the* B_f *value to request* `Header` *and* `Justification` *is* `0001 0001` *(17).*

Field	***Value***
Header	*0000 0001*
Body	*0000 0010*
Justification	*0001 0000*

Table 4.2. *Bits of block data to be requested.*

- B_s *is a Protobuf structure indicating a varying data type of the following values:*

Type	***Id***	***Description***
bytes	*2*	*The block hash*
bytes	*3*	*The block number*

Table 4.3. *Protobuf message indicating the block to start from.*

- B_e *is either the block hash or block number depending on the value of* B_s*. An implementation-defined maximum is used when unspecified.*
- `Direction` *is a Protobuf structure indicating the sequence direction of the requested blocks. The structure is a varying data type, as defined in Definition B.4, of the following format:*

Id	***Description***
0	*Enumerate in ascending order (from child to parent)*
1	*Enumerate in descending order (from parent to canonical child)*

Table 4.4. `Direction` *Protobuf structure.*

- B_m *is the number of blocks to be returned. An implementation defined maximum is used when unspecified.*

DEFINITION 4.5. *The* `BlockResponse` *message is received after sending a* `BlockRequest` *message to a peer. The message is a Protobuf serialized structure of the following format:*

Type	***Id***	***Description***
repeated BlockData	*1*	*Block data for the requested sequence*

Table 4.5. `BlockResponse` *Protobuf message.*

where `BlockData` *is a Protobuf structure containing the requested blocks. Do note that the optional values are either present or absent depending on the requested fields (bitmask value). The structure has the following format:*

Type	***Id***	***Description***	***Value***
bytes	*1*	*Block header hash*	*Def. 3.8*
bytes	*2*	*Block header (optional)*	*Def. 3.6*
repeated bytes	*3*	*Block body (optional)*	*Def. 3.9*
bytes	*4*	*Block receipt (optional)*	
bytes	*5*	*Block message queue (optional)*	
bytes	*6*	*Justification (optional)*	*Def. 6.29*
bool	*7*	*Indicates whether the justification is empty (i.e. should be ignored).*	

Table 4.6. BlockData *Protobuf structure.*

4.8.3. Transactions

Transactions, as defined and described in Section 3.2, are sent directly to peers with which the Polkadot Host has an open transaction substream, as defined in Definition 4.6. Polkadot Host implementers should implement a mechanism that only sends a transaction once to each peer and avoids sending duplicates. Sending duplicate transactions might result in undefined consequences such as being blocked for bad behaviour by peers.

The mechanism for managing transactions is further described in Section 3.2.

DEFINITION 4.6. *The* ***transactions message*** *is the structure of how the transactions are sent over the network. It is represented by M_T and is defined as follows:*

$$M_T := \text{Enc}_{\text{SC}}(C_1, \ldots, C_n)$$

in which:

$$C_i := \text{Enc}_{\text{SC}}(E_i)$$

Where each E_i is a byte array and represents a separate extrinsic. The Polkadot Host is agnostic about the content of an extrinsic and treats it as a blob of data.

Transactions are sent over the `/dot/transactions/1` *substream.*

4.8.4. GRANDPA Messages

The exchange of GRANDPA messages is conducted on the `/paritytech/grandpa/1` substream. The process for the creation and distributing these messages is described in Section 6.3. The underlying messages are specified in this section.

DEFINITION 4.7. *A* ***GRANDPA gossip message*** *is a variant, as defined in Definition B.4, which identifies the message type that is cast by a voter. This type, followed by the sub-component, is sent to other validators.*

Id	*Type*	*Definiton*	*Repropagated*
0	*GRANDPA vote message*	*4.8*	*yes*
1	*GRANDPA commit message*	*4.10*	*yes*
2	*GRANDPA neighbor message*	*4.11*	*no*
3	*GRANDPA catch-up request message*	*4.12*	*no*
4	*GRANDPA catch-up message*	*4.13*	*no*

Table 4.7. *GRANDPA gossip message types*

4.8.4.1. GRANDPA Vote Messages

Definition 4.8. *A **GRANDPA vote message** by voter v, $\boldsymbol{M_v^{r,\text{stage}}}$, is gossip to the network by voter v with the following structure:*

$$\begin{aligned} M_v^{r,\text{stage}}(B) &:= \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}}, \mathit{SigMsg}) \\ \mathit{SigMsg} &:= (\mathit{msg}, \text{Sig}_{v_i}^{r,\text{stage}}, v_{\text{id}}) \\ \mathit{msg} &:= \text{Enc}_{\text{SC}}(\text{stage}, V_v^{r,\text{stage}}(B)) \end{aligned}$$

Where:

r	*round number*	*unsigned 64-bit integer*
$\text{id}_{\mathbb{V}}$	*authority set Id (Definition 6.24)*	*unsigned 64-bit integer*
$\text{Sig}_{v_i}^{r,\text{stage}}$	*signature (Definition 6.28)*	*512-bit array*
v_{id}	*Ed25519 public key of v*	*256-bit array*
stage	*0 if it's a pre-vote sub-round*	*8-bit integer*
	1 if it's a pre-commit sub-round	*8-bit integer*
	2 if it's a primary proposal message	*8-bit integer*
$V_v^{r,\text{stage}}(B)$	*GRANDPA vote for block B (Definition 6.26)*	*256-bit array, 32-bit integer*

This message is the sub-component of the GRANDPA gossip message as defined in Definition 4.7 of type Id 0.

4.8.4.2. GRANDPA Commit Message

Definition 4.9. *The **GRANDPA compact justification format** is an optimized data structure to store a collection of pre-commits and their signatures to be submitted as part of a commit message. Instead of storing an array of justifications, it uses the following format:*

$$J_{v_{0\ldots n}}^{r,\text{comp}} := (\{V_{v_0}^{r,\text{pc}}, \ldots, V_{v_n}^{r,\text{pc}}\}, \{(\text{Sig}_{v_0}^{r,\text{pc}}, v_{\text{id}_0}), \ldots, (\text{Sig}_{v_n}^{r,\text{pc}}, v_{\text{id}_n})\})$$

Where:

$V_{v_i}^{r,\text{pc}}$	*pre-commit vote of authority v_i (Definition 6.26)*	*256-bit array, 32-bit integer*
$\text{Sig}_{v_i}^{r,\text{pc}}$	*pre-commit signature of authority v_i (Definition 6.28)*	*512-bit array*
v_{id_i}	*public key of authority v_i*	*256-bit array*

Definition 4.10. *A* **GRANDPA** ***commit message*** *for block B in round r $\boldsymbol{M_v^{r,\text{Fin}}(B)}$ 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:*

$$M_v^{r,\text{Fin}}(B) := \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}}, V_v^r(B), J_{\tilde{v}_{0\ldots n}}^{r,\text{comp}})$$

Where:

r	*round number*	*unsigned 64-bit integer*
$\text{id}_{\mathbb{V}}$	*authority set Id (Definition 6.24)*	*unsigned 64-bit integer*
$V_v^r(B)$	*GRANDPA vote for block B (Definition 6.26)*	*256-bit array, 32-bit integer*
$J_{\tilde{v}_{0\ldots n}}^{r,\text{comp}}$	*compacted GRANDPA justifications (Definition 4.9) containing observed pre-commits of authorities $\tilde{v}_0$ to $\tilde{v}_n$*	*variable size*

This message is the sub-component of the GRANDPA gossip message as defined in Definition 4.7 of type Id 1.

4.8.4.3. GRANDPA Neighbor Message

Neighbor messages are sent to all connected peers but they are not repropagated on reception. A message should be send whenever the messages values change and at least every 5 minutes. The sender should take the recipients state into account and avoid sending messages to peers that are using a different voter sets or are in a different round. Messages received from a future voter set or round can be dropped and ignored.

DEFINITION 4.11. *A **GRANDPA neighbor message** is defined as*

$$M^{\text{neigh}} := \text{Enc}_{\text{SC}}(\text{version}, r, \text{id}_{\mathbb{V}}, H_h(B_{\text{last}}))$$

Where:

version	*version of neighbor message, currently 1*	*unsignes 8-bit integer*
r	*round number*	*unsigned 64-bit integer*
$\text{id}_{\mathbb{V}}$	*authority set Id (Definition 6.24)*	*unsigned 64-bit integer*
$H_i(B_{\text{last}})$	*block number of last finialized block* B_{last}	*unsigned 32-bit integer*

This message is the sub-component of the GRANDPA gossip message as defined in Definition 4.7 of type Id 2.

4.8.4.4. GRANDPA Catch-up Messages

Whenever a Polkadot node detects that it is lagging behind the finality procedure, it needs to initiate a *catch-up* procedure. GRANDPA Neighbor messages (see Section 4.11) reveal the round number for the last finalized GRANDPA round which the node's peers have observed. This provides the means to identify a discrepancy in the latest finalized round number observed among the peers. If such a discrepancy is observed, the node needs to initiate the catch-up procedure explained in Section 6.4.1.

In particular, this procedure involves sending a *catch-up request* and processing *catch-up response* messages specified here:

DEFINITION 4.12. *A **GRANDPA catch-up request message** for round r, $\boldsymbol{M_{i,v}^{\text{Cat}-q}(\text{id}_{\mathbb{V}}, r)}$, is a message sent from node i to its voting peer node v requesting the latest status of a GRANDPA round $r' > r$ of the authority set $\mathbb{V}_{\text{id}}$ along with the justification of the status and has the following structure:*

$$M_{i,v}^{r,\text{Cat}-q} := \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}})$$

This message is the sub-component of the GRANDPA Gossip message as defined in Definition 4.7 of type Id 3.

DEFINITION 4.13. ***GRANDPA catch-up response message** for round, $\boldsymbol{M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)}$, is a message sent by a node v to node i in response of a catch-up request $M_{v,i}^{\text{Cat}-q}(\text{id}_{\mathbb{V}}, r')$ in which $r \geqslant r'$ is the latest GRANDPA round which v has prove of its finalization and has the following structure:*

$$M_{v,i}^{r,\text{Cat}-s} := \text{Enc}_{\text{SC}}(\text{id}_{\mathbb{V}}, r, J_{0\ldots n}^{r,\text{pv}}(B), J_{0\ldots m}^{r,\text{pc}}(B), H_h(B'), H_i(B'))$$

Where B is the highest block which v believes to be finalized in round r. B' is the highest ancestor of all blocks voted on in the arrays of justifications $J_{0\ldots n}^{r,\text{pv}}(B)$ and $J_{0\ldots m}^{r,\text{pc}}(B)$ with the exception of the equivocationary votes.

This message is the sub-component of the GRANDPA Gossip message as defined in Definition 4.7 of type Id 4.

4.8.4.5. GRANDPA BEEFY

[NOTE: The BEEFY protocol is currently in early development and subject to change]

This section defines the messages required for the GRANDPA BEEFY protocol as described in Section 6.5. Those messages are sent over the `/paritytech/beefy/1` substream.

Definition 4.14. *A commitment, C, contains the information extracted from the finalized block at height $H_i(B_{\text{last}})$ as specified in the message body.*

C is a datastructe of the following format:

$$C \;=\; (R_h, H_i(B_{\text{last}}), \text{id}_{\mathbb{V}})$$

where

- *R_h is the MMR root of all the block header hashes leading up to the latest, finalized block.*
- *$H_i(B_{\text{last}})$ is the block number this commitment is for. Namely the latest, finalized block.*
- *$\text{id}_{\mathbb{V}}$ is the current authority set Id as defined in Definition 6.24.*

Definition 4.15. *A vote message, M_v, is direct vote created by the Polkadot Host on every BEEFY round and is gossiped to its peers. The message is a datastructure of the following format:*

$$M_v \;=\; \text{Enc}_{\text{SC}}(C, A_{\text{id}}^{\text{bfy}}, A_{\text{sig}})$$

where

- *C is the commitment as defined in Definition 4.14.*
- *$A_{\text{id}}^{\text{bfy}}$ is the ECDSA public key of the Polkadot Host.*
- *A_{sig} is the signature created with $A_{\text{id}}^{\text{bfy}}$ by signing the statement R_h in C.*

Definition 4.16. *A signed commitment, M_{sc}, is a datastructure of the following format:*

$$\begin{aligned} M_{\text{sc}} &= \text{Enc}_{\text{SC}}(C, S_n) \\ S_n &= (A_0^{\text{sig}}, \ldots, A_n^{\text{sig}}) \end{aligned}$$

where

- *C is the commitment as defined in Definition 4.14.*
- *S_n is an array where its exact size matches the number of validators in the current authority set as specified by $\text{id}_{\mathbb{V}}$ (Definition 6.24) in C. Individual items are of the type `Option` as defined in Definition B.5 which can contain a signature of a validator which signed the same statement (R_h in C) and is active in the current authority set. It's critical that the signatures are sorted based on their corresponding public key entry in the authority set.*

  For example, the signature of the validator at index 3 in the authority set must be placed at index 3 in S_n. If not signature is available for that validator, then the `Option` variant `None` is inserted. This sorting allows clients to map public keys to their corresponding signatures.

Definition 4.17. *A signed commitment witness, M_{sc}^w, is a light version of the signed commitment as defined in Definition 4.16. Instead of containing the entire list of signatures, it only claims which validator signed the statement.*

The message is a datastructure of the following format:

$$M_{\text{sc}}^w \;=\; \text{Enc}_{\text{SC}}(C, V_{0\ldots n}, R_{\text{sig}})$$

where

- *C is the commitment as defined in Definition 4.14.*
- *$V_{0\ldots n}$ is an array where its exact size matches the number of validators in the current authority set as specified by $\text{id}_{\mathbb{V}}$ in C. Individual items are booleans which indicate whether the validator has signed the statement (true) or not (false). It's critical that the boolean indicators are sorted based on their corresponding public key entry in the authority set.*

For example, the boolean indicator of the validator at index 3 in the authority set must be placed at index 3 in V_n. This sorting allows clients to map public keys to their corresponding boolean indicators.

- R_{sig} *is the MMR root of the signatures in the original signed commitment as defined in Definition 4.16.*

□

Chapter 5

Bootstrapping

This chapter provides an overview over the tasks a Polkadot Host needs to performs in order to join and participate in the Polkadot network. While this chapter does not go into any new specifications of the protocol, it has been included to provide implementors with a pointer to what these steps are and where they are defined. In short, the following steps should be taken by all bootstrapping nodes:

1. The node needs to populate the state storage with the official Genesis state which can be obtained from [?].
2. The node should maintains a set of around 50 active peers at any time. New peers can be found using the `libp2p` discovery protocols (Section 4.4)
3. The node should open and maintain the various required streams (Section 4.7) with each of its active peers.
4. Furthermore, the node should send block requests (Section 4.8.2) to these peers to receive all blocks in the chain and execute each of them.
5. Exchange neighbor packets (Section 4.8.4.3)

Validator nodes should take the following, additional steps.

1. Verify that the Host's session key is included in the current Epoch's authority set (Section 6.1.1).
2. Run the BABE lottery (Section 6.2) and wait for the next assigned slot in order to produce a block.
3. Gossip any produced blocks to all connected peers (Section 4.8.1).
4. Run the catch-up protocol (Section 6.4.1) to make sure that the node is participating in the current round and not a past round.
5. Run the GRANDPA rounds protocol (Section 6.3).

□

Chapter 6

Consensus

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

6.1. Common Consensus Structures

6.1.1. Consensus Authority Set

Because Polkadot is a proof-of-stake protocol, each of its consensus engines has its own set of nodes represented by known public keys, which have the authority to influence the protocol in pre-defined ways explained in this Section. To verify the validity of each block, the Polkadot node must track the current list of authorities for that block as formalized in Definition 6.1

Definition 6.1. *The **authority list** of block B for consensus engine C noted as $\mathbf{Auth_C(B)}$ is an array that contains the following pair of types for each of its authorities $A \in \text{Auth}_C(B)$:*

$$(\text{pk}_A, w_A)$$

pk_A *is the session public key of authority A as defined in Definition A.4. And w_A is a* `u64` *value, indicating the authority weight. The value of $\text{Auth}_C(B)$ is part of the Polkadot state. The value for $\text{Auth}_C(B_0)$ is set in the genesis state (see Section C) and can be retrieved using a runtime entry corresponding to consensus engine C.*

Remark 6.2. In Polkadot, all authorities have the weight $w_A = 1$. The weight w_A in Definition 6.1 exists for potential improvements in the protocol and could have a use-case in the future.

6.1.2. Runtime-to-Consensus Engine Message

The authority list (see Definition 6.1) is part of the Polkadot state and the Runtime has the authority to update this list in the course of any state transitions. The Runtime informs the corresponding consensus engine about the changes in the authority set by adding the appropriate consensus message as defined in Definition 6.3, in the form of a digest item to the block header of block B which caused the transition in the authority set.

Definition 6.3. *Consensus Message is a digest item of type 4 as defined in Definition 3.7 and consists of the pair:*

$$(E_{\text{id}}, \text{CM})$$

Where $E_{\text{id}} \in \mathbb{B}_4$ is the consensus engine unique identifier which can hold the following possible values

$$E_{\text{id}} := \begin{cases} \text{"BABE"} & \text{For messages related to BABE protocol referred to as } E_{\text{id}}(\text{BABE}) \\ \text{"FRNK"} & \text{For messages related to GRANDPA protocol referred to as } E_{\text{id}}(\text{FRNK}) \end{cases}$$

and CM is of varying data type which can hold one of the types described in Table 6.1 or 6.2:

Type Id	***Type***	***Sub-components***
1	*Next Epoch Data*	$(\text{Auth}_{\text{BABE}}, \mathcal{R})$
2	*On Disabled*	Auth_{ID}
3	*Next Config Data*	$(c, s_{2\text{nd}})$

Table 6.1. *The consensus digest item for BABE authorities*

Where:

- $Auth_{\text{BABE}}$ *is the authority list for the next epoch, as defined in definition 6.1.*
- $\mathcal{R}$ *is the 32-byte randomness seed for the next epoch, as defined in definition 6.22*
- Auth_{ID} *is an unsigned 64-bit integer pointing to an individual authority in the current authority list.*
- c *is the probability that a slot will not be empty, as defined in definition 6.11. It is encoded as a tuple of two unsigned 64-bit integers* $(c_{\text{nominator}}, c_{\text{denominator}})$ *which are used to compute the rational* $c = \frac{c_{\text{nominator}}}{c_{\text{denominator}}}$.
- $s_{2\text{nd}}$ *is the second slot configuration encoded as an 8-bit enum.*

Type Id	***Type***	***Sub-components***
1	*Scheduled Change*	$(\text{Auth}_C, N_{\text{delay}})$
2	*Forced Change*	$(\text{Auth}_C, N_{\text{delay}})$
3	*On Disabled*	Auth_{ID}
4	*Pause*	N_{delay}
5	*Resume*	N_{delay}

Table 6.2. *The consensus digest item for GRANDPA authorities*

Where:

- $Auth_C$ *is the authority list as defined in definition 6.1.*
- $N_{\text{delay}} := |\text{SUBCHAIN}(B, B')|$ *is an unsigned 32-bit integer indicating the length of the subchain starting at* B*, the block containing the consensus message in its header digest and ending when it reaches* N_{delay} *length as a path graph. The last block in that subchain,* B'*, depending on the message type, is either finalized or imported (and therefore validated by the block production consensus engine according to Algorithm 3.4. See below for details).*
- Auth_{ID} *is an unsigned 64-bit integer pointing to an individual authority in the current authority list.*

The Polkadot Host should inspect the digest header of each block and delegate consensus messages to their consensus engines.

The BABE consensus engine should react based on the type of consensus messages it receives as follows:

- **Next Epoch Data:** The Runtime issues this message on every first block of an epoch $\mathcal{E}_n$. The supplied authority set and randomness are intended to be used in the next epoch $\mathcal{E}_{n+1}$.
- **On Disabled**: An index to the individual authority in the current authority list that should be immediately disabled until the next authority set changes. When an authority gets disabled, the node should stop performing any authority functionality for that authority, including authoring blocks. Similarly, other nodes should ignore all messages from the indicated authority which pertain to their authority role.
- **Next Config Data:** These messages are only issued on configuration change and in the first block of an epoch. The supplied configuration data are intended to be used from the next epoch onwards.

The GRANDPA consensus engine should react based on the type of consensus messages it receives as follows:

- **Scheduled Change**: Schedule an authority set change after the given delay of $N_{\text{delay}} := |\text{SUBCHAIN}(B, B')|$ where B' in the definition of N_{delay}, is a block *finalized* by the finality consensus engine. The earliest digest of this type in a single block will be respected. No change should be scheduled if one is already finalized and the delay has not passed completely. If such an inconsistency occurs, the scheduled change should be ignored.

- **Forced Change**: Force an authority set change after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, B')|$ where B' in the definition of N_{delay}, is an *imported* block that has been validated by the block production consensus engine. Hence, the authority changeset is valid for every subchain containing B and where the delay has been exceeded. If one or more blocks gets finalized before the change takes effect, the authority set change should be disregarded. The earliest digest of this type in a single block will be respected. No change should be scheduled if one is already finalized and the delay has not passed completely. If such an inconsistency occurs, the scheduled change should be ignored.
- **On Disabled**: **This message is currently ignored by the Polkadot Host and will be for the forseeable future.** An index to the individual authority in the current authority list that should be immediately disabled until the next authority set changes. This message initial intension was to cause an imediatly suspension of all authority functionality with the specified authority.
- **Pause**: A signal to pause the current authority set after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, B')|$ where B' in the definition of N_{delay}, is a block *finalized* by the finality consensus engine. After finalizing block B', the authorities should stop voting.
- **Resume**: A signal to resume the current authority set after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, B')|$ where B' in the definition of N_{delay}, is an *imported* block and validated by the block production consensus engine. After authoring block B', the authorities should resume voting.

The active GRANDPA authorities can only vote for blocks that occurred after the finalized block in which they were selected. Any votes for blocks before the `Scheduled Change` came into effect would get rejected.

6.2. Block Production

The Polkadot Host uses BABE protocol [?] for block production. It is designed based on Ouroboros praos [?]. BABE execution happens in sequential non-overlapping phases known as an ***epoch***. Each epoch on its turn is divided into a predefined number of slots. All slots in each epoch are sequentially indexed starting from 0. At the beginning of each epoch, the BABE node needs to run Algorithm 6.1 to find out in which slots it should produce a block and gossip to the other block producers. In turn, the block producer node should keep a copy of the block tree and grow it as it receives valid blocks from other block producers. A block producer prunes the tree in parallel by eliminating branches that do not include the most recent finalized blocks according to Definition 1.12.

6.2.1. Preliminaries

Definition 6.4. *A **block producer**, noted by $\mathcal{P}_j$, is a node running the Polkadot Host which is authorized to keep a transaction queue and which it gets a turn in producing blocks.*

Definition 6.5. ***Block authoring session key pair*** $(\mathbf{sk}_j^s, \mathbf{pk}_j^s)$ *is an SR25519 key pair which the block producer $\mathcal{P}_j$ signs by their account key (see Definition A.1) and is used to sign the produced block as well as to compute its lottery values in Algorithm 6.1.*

Definition 6.6. *A block production **epoch**, formally referred to as $\mathcal{E}$, is a period with a pre-known starting time and fixed-length during which the set of block producers stays constant. Epochs are indexed sequentially, and we refer to the n^{th} epoch since genesis by $\mathcal{E}_n$. Each epoch is divided into equal-length periods known as block production **slots**, sequentially indexed in each epoch. The index of each slot is called a **slot number**. The equal length duration of each slot is called the **slot duration** and indicated by $\mathcal{T}$. Each slot is awarded to a subset of block producers during which they are allowed to generate a block.*

Remark 6.7. Substrate refers to an epoch as "session" in some places, however, epoch should be the preferred and official name for these periods.

NOTATION 6.8. *We refer to the number of slots in epoch $\mathcal{E}_n$ by sc_n. sc_n is set to the* `duration` *field in the returned data from the call of the Runtime entry* `BabeApi_configuration` *(see E.3.8.1) at genesis. For a given block B, we use the notation $\boldsymbol{s_B}$ to refer to the slot during which B has been produced. Conversely, for slot s, $\mathcal{B}_s$ is the set of Blocks generated at slot s.*

Definition 6.9 provides an iterator over the blocks produced during a specific epoch.

DEFINITION 6.9. *By* SUBCHAIN($\mathcal{E}_n$) *for epoch $\mathcal{E}_n$, we refer to the path graph of* BT *containing all the blocks generated during the slots of epoch $\mathcal{E}_n$. When there is more than one block generated at a slot, we choose the one which is also on* LONGEST-CHAIN(BT).

DEFINITION 6.10. *A block producer* ***equivocates*** *if they produce more than one block at the same slot. The proof of equivocation are the given distinct headers that were signed by the validator and which include the slot number.*

The Polkadot Host must detect equivocations committed by other validators and submit those to the Runtime as described in Section E.3.8.6.

6.2.2. Block Production Lottery

DEFINITION 6.11. *The* ***BABE constant*** *$c \in (0,1)$ is the probability that a slot will not be empty and used in the winning threshold calculation (see Definition 6.12).*

The babe constant (Definition 6.11) is initialized at genesis to the value returned by calling `BabeApi_configuration` (see E.3.8.1). For efficiency reasons, it is generally updated by the Runtime through the "Next Config Data" consensus message (see Definition 6.3) in the digest of the first block of an epoch for the next epoch.

DEFINITION 6.12. *The* ***Winning threshold*** *denoted by $\boldsymbol{\tau_{\varepsilon_n}}$ is the threshold that is used alongside the result of Algorithm 6.1 to decide if a block producer is the winner of a specific slot. τ_{ε_n} is calculated as follows:*

$$\tau_{\varepsilon_n} := 1 - (1-c)^{\frac{1}{|\text{AuthorityDirectory}^{\mathcal{E}_n}|}}$$

where the $\text{AuthorityDirectory}^{\mathcal{E}_n}$ *is the set of BABE authorities for epoch ε_n and $c \in (0,1)$ is the BABE constant as defined in definition 6.11.*

A block producer aiming to produce a block during $\mathcal{E}_n$ should run Algorithm 6.1 to identify the slots it is awarded. These are the slots during which the block producer is allowed to build a block. The sk is the block producer lottery secret key and n is the index of the epoch for whose slots the block producer is running the lottery.

ALGORITHM 6.1. BLOCK-PRODUCTION-LOTTERY(sk: the session secret key of the producer, n: the epoch index)

```
r ← EPOCH-RANDOMNESS(n)
for i := 1 to sc_n
    (π, d) ← VRF(r, i, sk)
    A[i] ← (d, π)
return A
```

For any slot i in epoch n where $d < \tau$, the block producer is required to produce a block. For the definitions of EPOCH-RANDOMNESS and VRF functions, see Section 6.2.5 and Section A.4 respectively.

6.2.3. Slot Number Calculation

DEFINITION 6.13. *Let s_i and s_j be two slots belonging to epochs $\mathcal{E}_k$ and $\mathcal{E}_l$. By* **SLOT-OFFSET($\boldsymbol{s_i}$, $\boldsymbol{s_j}$)** *we refer to the function whose value is equal to the number of slots between s_i and s_j (counting s_j) on the time continuum. As such, we have* $\text{SLOT-OFFSET}(s_i, s_i) = 0$.

It is imperative for the security of the network that each block producer correctly determines the current slot numbers at a given time by regularly estimating the local clock offset in relation to the network (Definition 6.14).

Definition 6.14. *The **relative time synchronization** is a tuple of a slot number and a local clock timestamp* $(s_{\text{sync}}, t_{\text{sync}})$ *describing the last point at which the slot numbers have been synchronized with the local clock.*

Algorithm 6.2. Slot-Time(s: slot number)

1: **return** $t_{\text{sync}} + \text{Slot-Offset}(s_{\text{sync}}, s) \times \mathcal{T}$

Note 6.15. The calculation described in this section is still to be implemented and deployed. For now, each block producer is required to synchronize its local clock using NTP instead. The current slot s is then calculated by $s = t_{\text{unix}} / \mathcal{T}$ where t_{unix} is the current UNIX time in seconds since 1970-01-01 00:00:00 UTC. That also entails that slot numbers are currently not reset at the beginning of each epoch.

Polkadot does this synchronization without relying on any external clock source (e.g. through the *Network Time Protocol* or the *Global Positioning System*). To stay in synchronization, each producer is therefore required to periodically estimate its local clock offset in relation to the rest of the network.

This estimation depends on the two fixed parameters $\boldsymbol{k}$ (Definition 6.16) and $\boldsymbol{s_{\text{cq}}}$ (Definition 6.17). These are chosen based on the results of a formal security analysis, currently assuming a $1\,s$ clock drift per day and targeting a probability lower than 0.5% for an adversary to break BABE in 3 years with resistance against a network delay up to $1/3$ of the slot time and a Babe constant (Definition 6.11) of $c = 0.38$.

Definition 6.16. *The **pruned best chain** $\boldsymbol{C^{\lceil k}}$ is the longest chain selected according to Definition 1.14 with the last k Blocks pruned. We chose $k = 140$. The **last (probabilistically) finalized block** describes the last block in this pruned best chain.*

Definition 6.17. *The **chain quality** $\boldsymbol{s_{\text{cq}}}$ represents the number of slots that are used to estimate the local clock offset. Currently, it is set to $s_{\text{cq}} = 3000$.*

The prerequisite for such a calculation is that each producer stores the arrival time of each block (Definition 6.18) measured by a clock that is otherwise not adjusted by any external protocol.

Definition 6.18. *The **block arrival time** of block B for node j formally represented by $\boldsymbol{T_B^j}$ is the local time of node j when node j has received block B for the first time. If the node j itself is the producer of B, T_B^j is set equal to the time that the block is produced. The index j in T_B^j notation may be dropped and B's arrival time is referred to by T_B when there is no ambiguity about the underlying node.*

[Currently still lacks a clear definition of when block arrival times are considered valid and how to differentiated imported block on initial sync from "fresh" blocks that were just produced.]

Definition 6.19. *A **sync period** is an interval at which each validator (re-)evaluates its local clock offsets. The first sync period $\mathfrak{E}_1$ starts just after the genesis block is released. Consequently, each sync period $\mathfrak{E}_i$ starts after $\mathfrak{E}_{i-1}$. The length of the sync period is equal to s_{qc} as defined in Definition 6.17 and expressed in the number of slots.*

All validators are then required to run Algorithm 6.3 at the beginning of each sync period (Definition 6.19) to update their synchronization using all block arrival times of the previous period. The algorithm should only be run once all the blocks in this period have been finalized, even if only probabilistically (Definition 6.16). The target slot to which to synchronize should be the first slot in the new sync period.

ALGORITHM 6.3. MEDIAN-ALGORITHM($\mathfrak{E}_j$: sync period used for the estimate, s_{sync}: slot time to estimate)

1: $T_s \leftarrow \{\}$
2: **for** B_i **in** $\mathfrak{E}_j$
3: $\quad t^{B_i}_{\text{estimate}} \leftarrow T_{B_i} + \text{SLOT-OFFSET}(s_{B_i}, s_{\text{sync}}) \times \mathcal{T}$
4: $\quad T_s \leftarrow T_s \cup t^{B_i}_{\text{estimate}}$
5: **return** Median(T_s)

$\mathcal{T}$ is the slot duration defined in Definition 6.6.

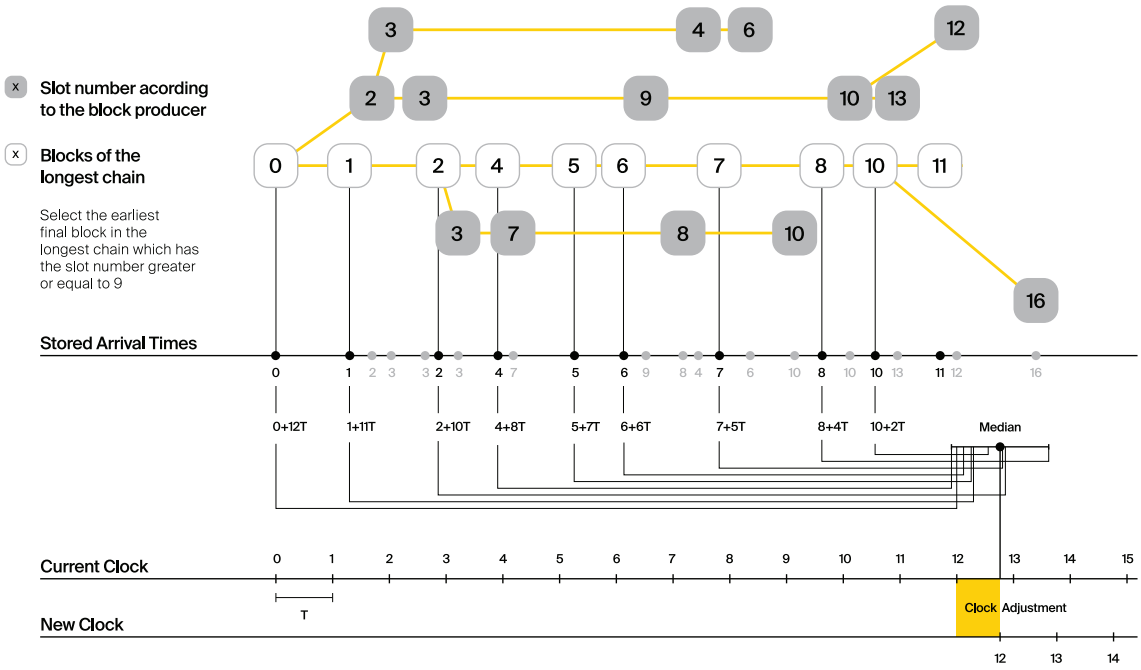

Figure 6.1. An exemplary result of Median Algorithm in first sync epoch with $s_{\text{cq}} = 9$ and $k = 1$.

6.2.4. Block Production

Throughout each epoch, each block producer should run Algorithm 6.4 to produce blocks during the slots it has been awarded during that epoch. The produced block needs to carry the *BABE header* as well as the *block signature* as Pre-Runtime and Seal digest items defined in Definition 6.20 and 6.21 respectively.

DEFINITION 6.20. *The* ***BABE Header*** *of block B, referred to formally by* $\boldsymbol{H_{\text{BABE}}(B)}$ *is a tuple and consists of the following components:*

$$(d, \pi, j, s)$$

in which:

π, d: *the results of the block lottery for slot s.*
j: *the index of the block producer in the authority directory of the current epoch*
s: *the slot at which the block is produced.*

$H_{\text{BABE}}(B)$ must be included as a digest item of Pre-Runtime type in the header digest $H_d(B)$ as defined in Definition 3.7.

DEFINITION 6.21. *The* ***Block Signature*** *S_B is a signature of the block header hash (see Definition3.8) and defined as*

$$\text{Sig}_{\text{SR25519,sk}^s_j}(H_h(B))$$

S_B should be included in $H_d(B)$ as the Seal digest item according to Definition 3.7 of value:

$$(E_{\text{id}}(\text{BABE}), S_B)$$

*in which, E_{id}(BABE) is the BABE consensus engine unique identifier defined in Definition 6.3. The Seal digest item is referred to as the **BABE Seal**.*

Algorithm 6.4. Invoke-Block-Authoring(sk, pk, n, BT: Current Block Tree)

```
A ← Block-production-lottery(sk, n)
for s ← 1 to sc_n
    Wait(until Slot-Time(s))
    (d, π) ← A[s]
    if d < τ
        C_Best ← Longest-Chain(BT)
        B_s ← Build-Block(C_Best)
        Add-Digest-Item(B_s,Pre-Runtime,E_id(BABE), H_BABE(B_s))
        Add-Digest-Item(B_s,Seal,S_B)
        Broadcast-Block(B_s)
```

Add-Digest-Item appends a digest item to the end of the header digest $H_d(B)$ according to Definition 3.7.

6.2.5. Epoch Randomness

Definition 6.22. *For epoch $\mathcal{E}$, there is a 32-byte **randomness seed** $\mathcal{R}_{\mathcal{E}}$ computed based on the previous epochs VRF outputs. For $\mathcal{E}_0$ and $\mathcal{E}_1$, the randomness seed is provided in the genesis state.*

At the beginning of each epoch, $\mathcal{E}_n$ the host will receive the randomness seed $\mathcal{R}_{\mathcal{E}_{n+1}}$(Definition 6.22) necessary to participate in the block production lottery in the next epoch $\mathcal{E}_{n+1}$ from the Runtime, through the *Next Epoch Data* consensus message (Definition 6.3) in the digest of the first block.

6.2.6. Verifying Authorship Right

When a Polkadot node receives a produced block, it needs to verify if the block producer was entitled to produce the block in the given slot by running Algorithm 6.5 where:

- T_B is B's arrival time defined in Definition 6.18.
- $H_d(B)$ is the digest sub-component of Head(B) defined in Definitions 3.6 and 3.7.
- The Seal D_s is the last element in the digest array $H_d(B)$ as defined in Definition 3.7.
- Seal-Id is the type index showing that a digest item of variable type is of *Seal* type (See Definitions B.7 and 3.7)
- AuthorityDirectory$^{\mathcal{E}_c}$ is the set of Authority ID for block producers of epoch $\mathcal{E}_c$.
- verify-Slot-Winner is defined in Algorithm 6.6.

Algorithm 6.5. Verify-Authorship-Right(Head$_s(B)$: The header of the block being verified)

```
s ← Slot-Number-At-Given-Time(T_B)
E_c ← Current-Epoch()
(D_1, ..., D_length(H_d(B))) ← H_d(B)
D_s ← D_length(H_d(B))
H_d(B) ← (D_1, ..., D_length(H_d(B))−1) //remove the seal from the digest
```

6: $(\text{id}, \text{Sig}_B) \leftarrow \text{Dec}_{\text{SC}}(D_s)$
7: **if** $\text{id} \neq$ SEAL-ID
8: **error** "Seal missing"
9: AuthorID $\leftarrow$AuthorityDirectory$^{\mathcal{E}_c}[H_{\text{BABE}}(B)$.SingerIndex]
10: VERIFY-SIGNATURE(AuthorID, $H_h(B), \text{Sig}_B$)
11: **if** $\exists B' \in \text{BT}: H_h(B) \neq H_h(B)$ **and** $s_B = s'_B$ **and** $\text{SignerIndex}_B = \text{SignerIndex}_{B'}$
12: **error** "Block producer is equivocating"
13: VERIFY-SLOT-WINNER($(d_B, \pi_B), s$,AuthorID)

Algorithm 6.6 runs as a part of the verification process, when a node is importing a block, in which:

- EPOCH-RANDOMNESS is defined in Definition 6.22.
- $H_{\text{BABE}}(B)$ is the BABE header defined in Definition 6.20.
- VERIFY-VRF is described in Section A.4.
- τ is the winning threshold defined in 6.12.

ALGORITHM 6.6. VERIFY-SLOT-WINNER(B: the block whose winning status to be verified)

1: $\mathcal{E}_c \leftarrow$ CURRENT-EPOCH
2: $\rho \leftarrow$ EPOCH-RANDOMNESS(c)
3: VERIFY-VRF$(\rho, H_{\text{BABE}}(B).(d_B, \pi_B), H_{\text{BABE}}(B).s, c)$
4: **if** $d_B \geqslant \tau$
5: **error** "Block producer is not a winner of the slot"

(d_B, π_B): Block Lottery Result for Block B,
s_n: the slot number,
n: Epoch index
AuthorID: The public session key of the block producer

6.2.7. Block Building Process

The blocks building process is triggered by Algorithm 6.4 of the consensus engine which runs Alogrithm 6.7.

ALGORITHM 6.7. BUILD-BLOCK(C_{Best}: The chain is where at its head, the block to be constructed, is s: Slot number)

1: $P_B \leftarrow$ HEAD(C_{Best})
2: Head$(B) \leftarrow (H_p \leftarrow H_h(P_B), H_i \leftarrow H_i(P_B) + 1, H_r \leftarrow \phi, H_e \leftarrow \phi, H_d \leftarrow \phi)$
3: CALL-RUNTIME-ENTRY(`Core_initialize_block`, Head(B))
4: *I-D* $\leftarrow$ CALL-RUNTIME-ENTRY(`BlockBuilder_inherent_extrinsics`, INHERENT-DATA)
5: **for** *E* **in** *I-D*
6: CALL-RUNTIME-ENTRY(`BlockBuilder_apply_extrinsics,E`)
7: **while not** END-OF-SLOT(s)
8: $E \leftarrow$ NEXT-READY-EXTRINSIC()
9: $R \leftarrow$ CALL-RUNTIME-ENTRY(`BlockBuilder_apply_extrinsics`,E)
10: **if** BLOCK-IS-FULL(R)
11: **break**
12: **if** SHOULD-DROP(R)
13: DROP(E)

14: Head(B) ← Call-Runtime-Entry(`BlockBuilder_finalize_block`,B)
15: B ← Add-Seal(B)

- Head(B) is defined in Definition 3.6.
- Call-Runtime-Entry is defined in Notation 3.2.
- Inherent-Data is defined in Definition 3.5.
- End-Of-Slot indicates the end of the BABE slot as defined in Algorithm 6.3 respectively Definition 6.6.
- Next-Ready-Extrinsic indicates picking an extrinsic from the extrinsics queue (Definition 3.4).
- Block-Is-Full indicates that the maximum block size is being used.
- Should-Drop determines based on the result R whether the extrinsic should be dropped or remain in the extrinsics queue and scheduled for the next block. The `ApplyExtrinsicResult` as defined in Definition E.3 describes this behavior in more detail.
- Drop indicates removing the extrinsic from the extrinsic queue (Definition 3.4).
- Add-Seal adds the seal to the block as defined in Definition 3.7 before sending it to peers. The seal is removed again before submitting it to the Runtime.

6.3. Finality

The Polkadot Host uses GRANDPA Finality protocol [?] to finalize blocks. Finality is obtained by consecutive rounds of voting by the 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 performs to successfully participate in the block-finalization process.

6.3.1. Preliminaries

Definition 6.23. *A **GRANDPA Voter**, v, represented by a key pair $(k_v^{\text{pr}}, v_{\text{id}})$ where k_v^{pr} represents an* ED25519 *private key, and v_{id} represents the public key of v. It is a node running a GRANDPA protocol and broadcasting votes to finalize blocks in a Polkadot Host-based chain. The **set of all GRANDPA voters** for a given block B is indicated by $\mathbb{V}_B$. In that regard, we have [change function name, only call at genesis, adjust V_B over the sections]*

$$\mathbb{V}_B = \texttt{grandpa\_authorities}(B)$$

where `grandpa_authorities` *is the entry into the Runtime described in Section E.3.7.1. We refer to $\mathbb{V}_B$ as $\mathbb{V}$ when there is no chance of ambiguity.*

*Analogously, we say that a Polkadot node is a **non-voter node** for block B, if it does not own any of the key pairs in $\mathbb{V}_B$.*

Definition 6.24. *The **authority set Id** ($\text{id}_{\mathbb{V}}$) is an incremental counter which tracks the amount of authority list changes that occurred (Definition 6.3). Starting with the value of zero at genesis, the Polkadot Host increments this value by one every time a **Scheduled Change** or a **Forced Change** occurs. The authority set Id is an unsigned 64-bit integer.*

Definition 6.25. ***GRANDPA state**,* GS*, is defined as [verify V_id and id_V usage, unify]*

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

where:

$\mathbb{V}$*: is the set of voters.*
$\mathbf{id}_{\mathbb{V}}$*: is the authority set ID as defined in Definition 6.24.*
$\boldsymbol{r}$*: is the voting round number.*

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

DEFINITION 6.26. *A **GRANDPA vote** or $V(B)$, represents 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 the block hash and the block number defined in Definitions 3.6 and 3.8 respectively.

DEFINITION 6.27. *Voters engage in a maximum of 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 cast by voter v in round r (for block B) during the pre-vote and the pre-commit sub-round respectively.

Voting is done by means of broadcasting voting messages to the network. The structure of these messages is described in Section 4.8.4. Validators inform their peers about the block finalized in round r by broadcasting a commit message (see Algorithm 6.9 for more details).

DEFINITION 6.28. $\mathbf{Sign}_{v_i}^{r,\mathbf{stage}}$ *refers to the signature of a voter for a specific message in a round and is formally defined as:*

$$\mathrm{Sign}_{v_i}^{r,\mathrm{stage}} := \mathrm{Sig}_{\mathrm{ED25519}}(\mathrm{msg}, r, \mathrm{id}_{\mathbb{V}})$$

Where:

msg	*the message to be signed*	*arbitrary*
r:	*round number*	*unsigned 64-bit integer*
$\mathrm{id}_{\mathbb{V}}$	*authority set Id (Definition 6.24) of v*	*unsigned 64-bit integer*

DEFINITION 6.29. *The **justification** for block B in round r, $\boldsymbol{J^{r,\mathbf{stage}}(B)}$, is a vector of pairs of the type:*

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

in which either

$$B' \geqslant B$$

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

*In all cases, $\mathrm{Sign}_{v_i}^{r,\mathrm{stage}}(B')$, as defined in Definition 6.28, is the signature of voter $v_i \in \mathbb{V}_B$ broadcasted during either the pre-vote (stage = pv) or the pre-commit (stage = pc) sub-round of round r. A **valid justification** must only contain up-to-one valid vote from each voter and must not contain more than two equivocatory votes from each voter.*

DEFINITION 6.30. *We say $J^{r,\mathrm{pc}}(B)$ **justifies the finalization** of $B' \geqslant B$ **for a non-voter node n** if the number of valid signatures in $J^{r,\mathrm{pc}}(B)$ for B' is greater than $\frac{2}{3}|\mathbb{V}_B|$.*

Note that $J^{r,\mathrm{pc}}(B)$ can only be used by a non-voter node to finalize a block. In contrast, a voter node can only be assured of the finality of block B by actively participating in the voting process. That is by invoking Algorithm 6.9. See Definition 6.39 for more details.

The GRANDPA protocol dictates how an honest voter should vote in each sub-round, which is described in Algorithm 6.9. After defining what constitutes a vote in GRANDPA, we define how GRANDPA counts votes.

DEFINITION 6.31. *Voter v **equivocates** if they broadcast two or more valid votes to blocks during one voting sub-round. In such a situation, we say that v is an **equivocator** and any vote $V_v^{r,\mathrm{stage}}(B)$ cast by v in that sub-round is an **equivocatory vote**, and*

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

represents the set of all equivocator 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:*

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

The Polkadot Host must detect equivocations committed by other validators and submit those to the Runtime as described in Section E.3.7.2.

Definition 6.32. *A vote* $V^{r,\text{stage}}_v = V(B)$ *is* ***invalid*** *if*

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

Definition 6.33. *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* $V^{r,\text{stage}}_{v_i}$ *cast in round* r *and received by* v *such that* $V^{r,\text{stage}}_{v_i} = V(B)$*.*

Definition 6.34. *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 votes by v in the sub-round stage of round r for block B****,* $\boldsymbol{V^{r,\text{stage}}_{\text{obs}(v)}(B)}$ *is equal to all of the observed direct votes cast for block* B *and all of the* B*'s descendants 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 votes for Block B in round r*** *is defined to be the size of that set plus the total number of equivocator voters:*

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

Note that for genesis state we always have $\#V^{r,\text{pv}}_{\text{obs}(v)}(B) = |\mathbb{V}|$.

Definition 6.35. *Let* $V^{r,\text{stage}}_{\text{unobs}(v)}$ *be the set of voters whose vote in the given stage has not been received. We define the* ***total number of potential votes for Block B in round r*** *to be:*

$$\#V^{r,\text{stage}}_{\text{obv}(v),\text{pot}}(B) := |V^{r,\text{stage}}_{\text{obs}(v)}(B)| + |V^{r,\text{stage}}_{\text{unobs}(v)}| + \text{Min}\left(\frac{1}{3}|\mathbb{V}|, |\mathbb{V}| - |V^{r,\text{stage}}_{\text{obs}(v)}(B)| - |V^{r,\text{stage}}_{\text{unobs}(v)}|\right)$$

Definition 6.36. *The current* ***pre-voted*** *block* $\boldsymbol{B^{r,\text{pv}}_v}$*, also know as GRANDPA GHOST is the block chosen by Algorithm 6.12:*

$$B^{r,\text{pv}}_v := \text{GRANDPA-GHOST}(r)$$

Finally, we define when a voter v sees a round as completable, that is when they are confident that $B^{r,\text{pv}}_v$ is an upper bound for what is going to be finalized in this round.

Definition 6.37. *We say that round* r *is* ***completable*** *if* $|V^{r,\text{pc}}_{\text{obs}(v)}| + \mathcal{E}^{r,\text{pc}}_{\text{obs}(v)} > \frac{2}{3}\mathbb{V}$ *and for all* $B' > B^{r,\text{pv}}_v$*:*

$$|V^{r,\text{pc}}_{\text{obs}(v)}| - \mathcal{E}^{r,\text{pc}}_{\text{obs}(v)} - |V^{r,\text{pc}}_{\text{obs}(v)}(B')| > \frac{2}{3}|\mathbb{V}|$$

Note that in practice we only need to check the inequality for those $B' > B^{r,\text{pv}}_v$ where $|V^{r,\text{pc}}_{\text{obs}(v)}(B')| > 0$.

6.3.2. Initiating the GRANDPA State

In order to participate coherently in the voting process, a validator must initiate its state and sync it with other active validators. In particular, considering that voting is happening in different distinct rounds where each round of voting is assigned a unique sequential round number r_v, it needs to determine and set its round counter r equal to the voting round r_n currently undergoing in the network. Algorithm 6.8 mandates the initialization procedure for GRANDPA protocol for a joining validator.

ALGORITHM 6.8. INITIATE-GRANDPA(
r_{last}: last round number (See the following),
,B_{last}: the last block which has been finalized on the chain (see Definition 6.39)
)

1: LAST-FINALIZED-BLOCK$\leftarrow B_{\text{last}}$
2: **if** $r_{\text{last}} = 0$
3: BEST-FINAL-CANDIDATE$(0)\leftarrow B_{\text{last}}$
4: GRANDPA-GHOST$(0)\leftarrow B_{\text{last}}$
5: $r_n \leftarrow r_{\text{last}+1}$
6: PLAY-GRANDPA-ROUND(r_n)

r_{last} is equal to the latest round the voter has observed that other voters are voting on. The voter obtains this information through various gossiped messages including those mentioned in Definition 6.39.

r_{last} is set to 0 if the GRANDPA node is initiating the GRANDPA voting process as a part of a new authority set. This is because the GRANDPA round number reset to 0 for every authority set change.

6.3.2.1. Voter Set Changes

Voter set changes are signalled by Runtime via a consensus engine message as described in Section 6.1.2. When Authorities process such messages they must not vote on any block with a higher number than the block at which the change is supposed to happen. The new authority set should reinitiate GRANDPA protocol by executing Algorithm 6.8.

6.3.3. Voting Process in Round r

For each round r, an honest voter v must participate in the voting process by following Algorithm 6.9.

ALGORITHM 6.9. PLAY-GRANDPA-ROUND(r)

1: $t_{r,v} \leftarrow$ Current local time
2: primary $\leftarrow$ DERIVE-PRIMARY(r)
3: **if** $v =$ primary
4: BROADCAST$(M_v^{r-1,\text{Fin}}$(BEST-FINAL-CANDIDATE$(r\text{-}1)))$
5: **if** BEST-FINAL-CANDIDATE$(r-1) \geqslant$ LAST-FINALIZED-BLOCK
6: BROADCAST$(M_v^{r-1,\text{Prim}}$(BEST-FINAL-CANDIDATE$(r\text{-}1)))$
7: RECEIVE-MESSAGES(**until** Time $\geqslant t_{r,v} + 2 \times T$ **or** r **is** completable)
8: $L \leftarrow$ BEST-FINAL-CANDIDATE$(r\text{-}1)$
9: $N \leftarrow$ BEST-PREVOTE-CANDIDATE(r)
10: BROADCAST$(M_v^{r,\text{pv}}(N))$
11: RECEIVE-MESSAGES(**until** $B_v^{r,\text{pv}} \geqslant L$ **and** (Time $\geqslant t_{r,v} + 4 \times T$ **or** r **is** completable))
12: BROADCAST$(M_v^{r,\text{pc}}(B_v^{r,\text{pv}}))$

13: Attempt-To-Finalize-Round(r)
14: Receive-Messages(**until** r **is** completable **and** Finalizable($r-1$)
and Last-Finalized-Block$\geqslant$Best-Final-Candidate(r-1))
15: Play-Grandpa-round($r+1$)

Where:

– T is sampled from a log-normal distribution whose mean and standard deviation are equal to the average network delay for a message to be sent and received from one validator to another.

– Derive-Primary is described in Algorithm 6.10.

– The condition of *completablitiy* is defined in Definition 6.37.

– Best-Final-Candidate function is explained in Algorithm 6.11.

– Attempt-To-Finalize-Round(r) is described in Algorithm 6.15.

– Finalizabl is defined in Algorithm 6.14.

Algorithm 6.10. Derive-Primary(r: the GRANDPA round whose primary to be determined)

1: **return** $r \bmod |\mathbb{V}|$

$\mathbb{V}$ is the GRANDPA voter set as defined in Definition 6.23.

Algorithm 6.11. Best-Final-Candidate(r)

1: $B_v^{r,\mathrm{pv}} \leftarrow$ GRANDPA-GHOST(r)
2: **if** $r = 0$
3: **return** $B_v^{r,\mathrm{pv}}$
4: **else**
5: $\mathcal{C} \leftarrow \{B' | B' \leqslant B_v^{r,\mathrm{pv}} : \#V^{r,\mathrm{pc}}_{\mathrm{obv}(v),\mathrm{pot}}(B') > 2/3|\mathbb{V}|\}$
6: **if** $\mathcal{C} = \phi$
7: **return** $B_v^{r,\mathrm{pv}}$
8: **else**
9: **return** $E \in \mathcal{C} : H_n(E) = \mathrm{Max}\,(H_n(B') : B' \in \mathcal{C})$

$\#V^{r,\mathrm{stage}}_{\mathrm{obv}(v),\mathrm{pot}}$ is defined in Definition 6.35.

Algorithm 6.12. GRANDPA-GHOST(r)

1: **if** $r = 0$:
2: $G \leftarrow B_{\mathrm{last}}$
3: **else**
4: $L \leftarrow$ Best-Final-Candidate($r-1$)
5: $\mathcal{G} = \{\forall B > L | \#V^{r,\mathrm{pv}}_{\mathrm{obs}(v)}(B) \geqslant 2/3|\mathbb{V}|\}$
6: **if** $\mathcal{G} = \phi$
7: $G \leftarrow L$
8: **else**
9: $G \in \mathcal{G} : H_n(G) = \mathrm{Max}(H_n(B)|\,\forall B \in \mathcal{G})$
10: **return** G

Where

- B_{last} is the last block which has been finalized on the chain (see Definitin 6.39)
- $\#V^{r,\text{pv}}_{\text{obs}(v)}(B)$ is defined in Definition 6.34.

ALGORITHM 6.13. BEST-PREVOTE-CANDIDATE(r: voting round to cast the pre-vote in)

$B^{r,\text{pv}}_v \leftarrow$ GRANDPA-GHOST(r)
if RECEIVED($M^{r,\text{prim}}_{v_{\text{primary}}}(B)$) **and** $B^{r,\text{pv}}_v \geqslant B > L$
    $N \leftarrow B$
else
    $N \leftarrow B^{r,\text{pv}}_v$

ALGORITHM 6.14. FINALIZABLE(r: voting round)

if r **is not** Completable
    return False
$G \leftarrow$ GRANDPA-GHOST($J^{r,\text{pv}}(B)$)
if $G = \phi$
    return False
$E_r \leftarrow$ BEST-FINAL-CANDIDATE(r)
if $E_r \neq \phi$ **and** $E_{r-1} \leqslant E_r \leqslant G$
    return True
else
    return False

The condition of *completablitiy* is defined in Definition 6.37.

ALGORITHM 6.15. ATTEMPT-TO-FINALIZE-ROUND(r)

$L \leftarrow$ LAST-FINALIZED-BLOCK
$E \leftarrow$ BEST-FINAL-CANDIDATE(r)
if $E \geqslant L$ **and** $V^{r,\text{pc}}_{\text{obs}(v)}(E) > 2/3|\mathbb{V}|$
    LAST-FINALIZED-BLOCK$\leftarrow E$
    if $M^{r,\text{Fin}}_v(E) \notin$ RECEIVED-MESSAGES
        BROADCAST($M^{r,\text{Fin}}_v(E)$)
        return
schedule-call ATTEMPT-TO-FINALIZE-ROUND(r) **when** RECEIVE-MESSAGES

Note that we might not always succeed in finalizing our best final candidate due to the possibility of equivocation. Example 6.38 serves to demonstrate such a situation:

Example 6.38. Let us assume that we have 100 voters and there are two blocks in the chain ($B_1 < B_2$). At round 1, we get 67 pre-votes for B_2 and at least one pre-vote for B_1 which means that GRANDPA-GHOST(1) = B_2.

Subsequently, potentially honest voters who could claim not seeing all the pre-votes for B_2 but receiving the pre-votes for B_1 would pre-commit to B_1. In this way, we receive 66 pre-commits for B_1 and 1 pre-commit for B_2. Henceforth, we finalize B_1 since we have a threshold commit (67 votes) for B_1.

At this point, though, we have BEST-FINAL-CANDIDATE(r)=B_2 as $\#V^{r,\text{stage}}_{\text{obv}(v),\text{pot}}(B_2) = 67$ and $2 > 1$.

However, at this point, the round is already completable as we know that we have GRANDPA-GHOST(1) = B_2 as an upper limit on what we can finalize and nothing greater than B_2 can be finalized at $r=1$. Therefore, the condition of Algorithm 6.9:14 is satisfied and we must proceed to round 2.

Nonetheless, we must continue to attempt to finalize round 1 in the background as the condition of 6.15:3 has not been fulfilled.

This prevents us from proceeding to round 3 until either:

– We finalize B_2 in round 2, or

– We receive an extra pre-commit vote for B_1 in round 1. This will make it impossible to finalize B_2 in round 1, no matter to whom the remaining pre-commits are going to be cast for (even with considering the possibility of 1/3 of voter equivocating) and therefore we have Best-Final-Candidate(r)=B_1.

Both scenarios unblock the Algorithm 6.9:14 Last-Finalized-Block$\geqslant$Best-Final-Candidate(r-1)) albeit in different ways: the former with increasing the Last-Finalized-Block and the latter with decreasing Best-Final-Candidate(r-1).

6.4. Block Finalization

Definition 6.39. *A Polkadot relay chain node n should consider block B as **finalized** if any of the following criteria hold for $B' \geqslant B$:*

- $V^{r,\mathrm{pc}}_{\mathrm{obs}(n)}(B') > 2/3|\mathbb{V}_{B'}|$.
- *it receives a $M^{r,\mathrm{Fin}}_{v}(B')$ message in which $J^r(B)$ justifies the finalization (according to Definition 6.29).*
- *it receives a block data message for B' with* Just(B') *defined in Section 3.3.1.2 which justifies the finalization.*

for

- any round r if the node n is *not* a GRANDPA voter.
- only for rounds r for which the node n has invoked Algorithm 6.9 if n is a GRANDPA voter.

Note that all Polkadot relay chain nodes are supposed to process GRANDPA commit messages regardless of their GRANDPA voter status.

6.4.1. Catching up

When a Polkadot node (re)joins the network during the process described in Chapter 5, it requests the history of state transition in the form of blocks, which it is missing. Each finalized block comes with the Justification of its finalization as defined in Definition 6.29. Through this process, they can synchronize the authority list currently performing the finalization process.

6.4.1.1. Sending the catch-up requests

When a Polkadot node has the same authority list as a peer node who is reporting a higher number for the "finalized round" field, it should send a catch-up request message, as specified in Definition 4.12, to the reporting peer in order to catch-up to the more advanced finalized round, provided that the following criteria hold:

– the peer node is a GRANDPA voter, and

– the last known finalized round for the Polkadot node is at least 2 rounds behind the finalized round for the peer.

6.4.1.2. Processing the catch-up requests

Only GRANDPA voter nodes are required to respond to the catch-up responses. When a GRANDPA voter node receives a catch-up request message it needs to execute Algorithm 6.16.

ALGORITHM 6.16. PROCESSCATCHUPREQUEST($M_{i,v}^{\text{Cat}-q}(\text{id}_{\mathbb{V}}, r)$: The catch-up message received from peer i (See Definition 4.12))

1: **if** $M_{i,v}^{\text{Cat}-q}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}} \neq \text{id}_{\mathbb{V}}$
2: **error** "Catching up on different set"
3: **if** $i \notin \mathbb{P}$
4: **error** "Requesting catching up from a non-peer"
5: **if** $r >$ LAST-COMPLETED-ROUND
6: **error** "Catching up on a round in the future"
7: SEND(i,$M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)$)

In which:

- $\text{id}_{\mathbb{V}}$ is the voter set id with which the serving node is operating
- r is the round number for which the catch-up is requested for.
- $\mathbb{P}$ is the set of immediate peers of node v.
- LAST-COMPLETED-ROUND is [define: https://github.com/w3f/polkadot-spec/issues/161]
- $M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)$ is the catch-up response defined in Definition 4.13.

6.4.1.3. Processing catch-up responses

A Catch-up response message contains critical information for the requester node to update their view on the active rounds which are being voted on by GRANDPA voters. As such, the requester node should verify the content of the catch-up response message and subsequently updates its view of the state of the finality of the Relay chain according to Algorithm 6.17.

ALGORITHM 6.17. PROCESS-CATCHUP-RESPONSE($M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)$: the catch-up response received from node v (See Definition 4.13))

1: $M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}}, r, J^{r,\text{pv}}(B), J^{r,\text{pc}}(B), H_h(B'), H_i(B') \leftarrow \text{Dec}_{\text{SC}}(M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r))$
2: **if** $M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}} \neq \text{id}_{\mathbb{V}}$
3: **error** "Catching up on different set"
4: **if** $r \leqslant$ LEADING-ROUND
5: **error** "Catching up in to the past"
6: **if** $J^{r,\text{pv}}(B)$ is not valid
7: **error** "Invalid pre-vote justification"
8: **if** $J^{r,\text{pc}}(B)$ is not valid
9: **error** "Invalid pre-commit justification"
10: $G \leftarrow$ GRANDPA-GHOST($J^{r,\text{pv}}(B)$)
11: **if** $G = \phi$
12: **error** "GHOST-less Catch-up"
13: **if** r **is not** completable
14: **error** "Catch-up round is not completable"
15: **if** $J^{r,\text{pc}}(B)$ justifies B' finalization
16: **error** "Unjustified Catch-up target finalization"
17: LAST-COMPLETED-ROUND$\leftarrow r$
18: **if** $i \in \mathbb{V}$
19: PLAY-GRANDPA-ROUND($r+1$)

6.5. Bridge design (BEEFY)

[NOTE: The BEEFY protocol is currently in early development and subject to change]

The BEEFY (Bridge Effiency Enabling Finality Yielder) is a secondary protocol to GRANDPA to support efficient bridging between the Polkadot network (relay chain) and remote, segregated blockchains, such as Ethereum, which were not built with the Polkadot interchain operability in mind. The protocol allows participants of the remote network to verify finality proofs created by the Polkadot relay chain validators. In other words: clients in the Ethereum network should able to verify that the Polkadot network is at a specific state.

Storing all the information necessary to verify the state of the remote chain, such as the block headers, is too expensive. BEEFY stores the information in a space-efficient way and clients can request additional information over the protocol.

6.5.1. Preliminaries

Definition 6.40. *Merkle Mountain Ranges, **MMR**, as defined in Definition [TODO] are used as an efficient way to send block headers and signatures to light clients.*

Definition 6.41. *The **statement** is the same piece of information which every relay chain validator is voting on. Namely, the MMR root of all the block header hashes leading up to the latest, finalized block.*

Definition 6.42. ***Witness data** contains the statement as defined in Definition 6.41, an array indicating which validator of the Polkadot network voted for the statement (but not the signatures themselves) and a MMR root of the signatures. The indicators of which validator voted for the statement are just claimes and provide no proofs . The network message is defined in Definition 4.17 and the relayer saves it on the chain of the remote network.*

Definition 6.43. *A **light client** is an abstract entity in a remote network such as Ethereum. It can be a node or a smart contract with the intent of requesting finality proofs from the Polkadot network. A light client reads the witness data as defined in Definition 6.42 from the chain, then requests the signatures directly from the relayer in order to verify those.*

The light client is expected to know who the validators are and has access to their public keys.

Definition 6.44. *A **relayer** (or "prover") is an abstract entity which takes finality proofs from the Polkadot network and makes those available to the light clients. Inherently, the relayer tries to convince the light clients that the finality proofs have been voted for by the Polkadot relay chain validators. The relayer operates offchain and can for example be a node or a collection of nodes.*

6.5.2. Voting on Statements

The Polkadot Host signs a statement as defined in Definition 6.41 and gossips it as part of a vote as defined in Definition 4.15 to its peers on every new, finalized block. The Polkadot Host uses ECDSA for signing the statement, since Ethereum has better compatibility for it compared to SR25519 or ED25519. [how does one map the validator set keys to the corresponding ECDSA keys?]

6.5.3. Committing Witnesses

The relayer as defined in Definition 6.44 participates in the Polkadot network by collecting the gossiped votes as defined in Definition 4.15. Those votes are converted into the witness data structure as defined in Definition 6.42. and the relayer saves the data on the chain of the remote network. The occurrence of saving witnesses on remote networks is undefined.

How the witness data is saved on the remote chain varies among networks or implementations. On Ethereum, for example, the relayer could call a smart contract which saves the witness data on chain and light clients can fetch this data.

[Add note about 2/3 majority]

6.5.4. Requesting Signed Commitments

A light client as defined in Definition 6.43 fetches the witness data as defined in Definition 6.42 from the chain. Once the light client knows which validators apparently voted for the specified statement, it needs to request the signatures from the relayer to verify whether the claims are actually true. This is achieved by requesting signed commitments as defined in Definition 4.16.

How those signed commitments are requested by the light client and delivered by the relayer varies among networks or implementations. On Ethereum, for example, the light client can request the signed commitments in form of a transaction, which results in a response in form of a transaction. [If the signed commitments are just transactions, which are stored in the blockchain, why bother with witnesses?]

□

Chapter 7

Availability & Validity

7.1. Introduction

Validators are responsible for guaranteeing the validity and availability of PoV blocks. There are two phases of validation that takes place in the AnV protocol.

The primary validation check is carried out by parachain validators who are assigned to the parachain which has produced the PoV block as described in Section 7.4. Once parachain validators have validated a parachain's PoV block successfully, they have to announce that according to the procedure described in Section 7.5 where they generate a statement that includes the parachain header with the new state root and the XCMP message root. This candidate receipt and attestations, which carries signatures from other parachain validators is put on the relay chain.

As soon as the proposal of a PoV block is on-chain, the parachain validators break the PoV block into erasure-coded chunks as described in Section 7.19 and distribute them among all validators. See Section 7.8 for details on how this distribution takes place.

Once validators have received erasure-coded chunks for several PoV blocks for the current relay chain block (that might have been proposed a couple of blocks earlier on the relay chain), they announce that they have received the erasure coded chunks on the relay chain by voting on the received chunks, see Section 7.9 for more details.

As soon as $>2/3$ of validators have made this announcement for any parachain block we *act on* the parachain block. Acting on parachain blocks means we update the relay chain state based on the candidate receipt and considered the parachain block to have happened on this relay chain fork.

After a certain time, if we did not collect enough signatures approving the availability of the parachain data associated with a certain candidate receipt we decide this parachain block is unavailable and allow alternative blocks to be built on its parent parachain block, see 7.9.1.

The secondary check described in Section 7.11, is done by one or more randomly assigned validators to make sure colluding parachain validators may not get away with validating a PoV block that is invalid and not keeping it available to avoid the possibility of being punished for the attack.

During any of the phases, if any validator announces that a parachain block is invalid then all validators obtain the parachain block and check its validity, see Section 7.12.3 for more details.

All validity and invalidity attestations go onto the relay chain, see Section 7.10 for details. If a parachain block has been checked at least by certain number of validators, the rest of the validators continue with voting on that relay chain block in the GRANDPA protocol. Note that the block might be challenged later.

7.2. Preliminaries

Definition 7.1. *The Polkadot project uses the **SCALE codec** to encode common data types such as integers, byte arrays, varying data types as well as other data structure. The SCALE codec is defined in a separate document known as "The Polkadot Host - Protocol Specification". [@fabio: link to document]*

Definition 7.2. *In the remainder of this chapter we assume that ρ is a Polkadot Parachain and B is a block which has been produced by ρ and is supposed to be approved to be ρ's next block. By R_ρ we refer to the **validation code** of parachain ρ as a WASM blob, which is responsible for validating the corresponding Parachain's blocks.*

DEFINITION 7.3. *The **witness proof** of block B, denoted by $\boldsymbol{\pi_B}$, is the set of all the external data which has gathered while the ρ runtime executes block B. The data suffices to re-execute R_ρ against B and achieve the final state indicated in the $H(B)$.*

This witness proof consists of light client proofs of state data that are generally Merkle proofs for the parachain state trie. We need this because validators do not have access to the parachain state, but only have the state root of it.

DEFINITION 7.4. *Accordingly we define the **proof of validity block** or **PoV** block in short, $\mathbf{PoV_B}$, to be the tuple:*

$$(B, \pi_B)$$

A PoV block is an extracted Merkle subtree, attached to the block. [@fabio: clarif this]

7.2.1. Extra Validation Data

Validators must submit extra validation data to Runtime R_ρ in order to build candidates, to fully validate those and to vote on their availability. Depending on the context, different types of information must be used.

Parachain validators get this extra validation data from the current relay chain state. Note that a PoV block can be paired with different extra validation data depending on when and which relay chain fork it is included in. Future validators would need this extra validation data because since the candidate receipt as defined in Definition 7.13 was included on the relay chain the needed relay chain state may have changed.

DEFINITION 7.5. *R_ρ^{up} is an varying data type (Definition 7.1) which implies whether the parachain is allowed to upgrade its validation code.*

$$R_\rho^{\text{up}} := \text{Option}(H_i(B_{\text{chain}}^{\text{relay}}) + n)$$

[@fabio: adjust formula?]

If this is Some*, it contains the number of the minimum relay chain height at which the upgrade will be applied, assuming an upgrade is currently signaled [@fabio: where is this signaled?]. A parachain should enact its side of the upgrade at the end of the first parachain block executing in the context of a relay-chain block with at least this height. This may be equal to the current perceived relay-chain block height, in which case the code upgrade should be applied at the end of the signaling block.*

DEFINITION 7.6. *The **validation parameters**, v_B^{VP}, is an extra input to the validation function, i.e. additional data from the relay chain state that is needed. It's a tuple of the following format:*

$$\text{vp}_B := (B, \text{head}(B_p), v_B^{\text{GVS}}, R_\rho^{\text{up}})$$

where each value represents:

- *B: the parachain block itself.*
- *$\text{head}(B_p)$: the parent head data (Definition 7.12) of block B.*
- *v_B^{GVP}: the global validation parameters (7.7).*
- *R_ρ^{up}: implies whether the parachain is allowed to upgrade its validation code (Definition 7.5).*

DEFINITION 7.7. *The **global validation parameters**, v_B^{GVP}, defines global data that apply to all candidates in a block.*

$$v_B^{\text{GVS}} := (\text{Max}_{\text{size}}^R, \text{Max}_{\text{size}}^{\text{head}}, H_i(B_{\text{chain}}^{\text{relay}}))$$

where each value represents:

- *$\text{Max}_{\text{size}}^R$: the maximum amount of bytes of the parachain Wasm code permitted.*
- *$\text{Max}_{\text{size}}^{\text{head}}$: the maximum amount of bytes of the head data (Definition 7.12) permitted.*

- $H_i(B^{\text{relay}}_{\text{chain}})$*: the relay chain block number this is in the context of.*

Definition 7.8. *The* ***local validation parameters****,* v_B^{LVP}*, defines parachain-specific data required to fully validate a block. It is a tuple of the following format:*

$$v_B^{\text{LVP}} := (\text{head}(B_p), \text{UINT128}, \text{Blake2}b(R_\rho), R_\rho^{\text{up}})$$

where each value represents:

- $\text{head}(B_p)$*: the parent head data (Definition 7.12) of block* B*.*
- UINT128*: the balance of the parachain at the moment of validation.*
- $\text{Blake2}b(R_\rho)$*: the Blake2b hash of the validation code used to execute the candidate.*
- R_ρ^{up}*: implies whether the parachain is allowed to upgrade its validation code (Definition 7.5).*

Definition 7.9. *The* ***validation result****,* r_B*, is returned by the validation code* R_ρ *if the provided candidate is is valid. It is a tuple of the following format:*

$$r_B := (\text{head}(B), \text{Option}(P_\rho^B), (\text{Msg}_0, ..., \text{Msg}_n), \text{UINT32})$$
$$\text{Msg} := (\mathbb{O}, \text{Enc}_{\text{SC}}(b_0, ..b_n))$$

where each value represents:

- $\text{head}(B)$*: the new head data (Definition 7.12) of block* B*.*
- $\text{Option}(P_\rho^B)$*: a varying data (Definition 7.1) containing an update to the validation code that should be scheduled in the relay chain.*
- Msg*: parachain "upward messages" to the relay chain.* $\mathbb{O}$ *identifies the origin of the messages and is a varying data type (Definition 7.1) and can be one of the following values:*

$$\mathbb{O} = \begin{cases} 0, & \textit{Signed} \\ 1, & \textit{Parachain} \\ 2, & \textit{Root} \end{cases}$$

  [@fabio: define the concept of "origin"]

  The following SCALE encoded array, $\text{Enc}_{\text{SC}}(b_0, ..b_n)$*, contains the raw bytes of the message which varies in size.*

- UINT32*: number of downward messages that were processed by the Parachain. It is expected that the Parachain processes them from first to last.*

Definition 7.10. *Accordingly we define the* ***erasure-encoded blob*** *or* ***blob*** *in short,* $\bar{\boldsymbol{B}}$*, to be the tuple:*

$$(B, \pi_B, v_B^{\text{GVP}}, v_B^{\text{LVP}})$$

where each value represents:

- B*: the parachain block.*
- π_B*: the witness data.*
- v_B^{GVP}*: the global validation parameters (Definition 7.7).*
- v_B^{LVP}*: the local validation parameters (Definition 7.8).*

Note that in the code the blob is referred to as "AvailableData".

7.3. Overal process

The Figure 7.1 demonstrates the overall process of assuring availability and validity in Polkadot [complete the Diagram].

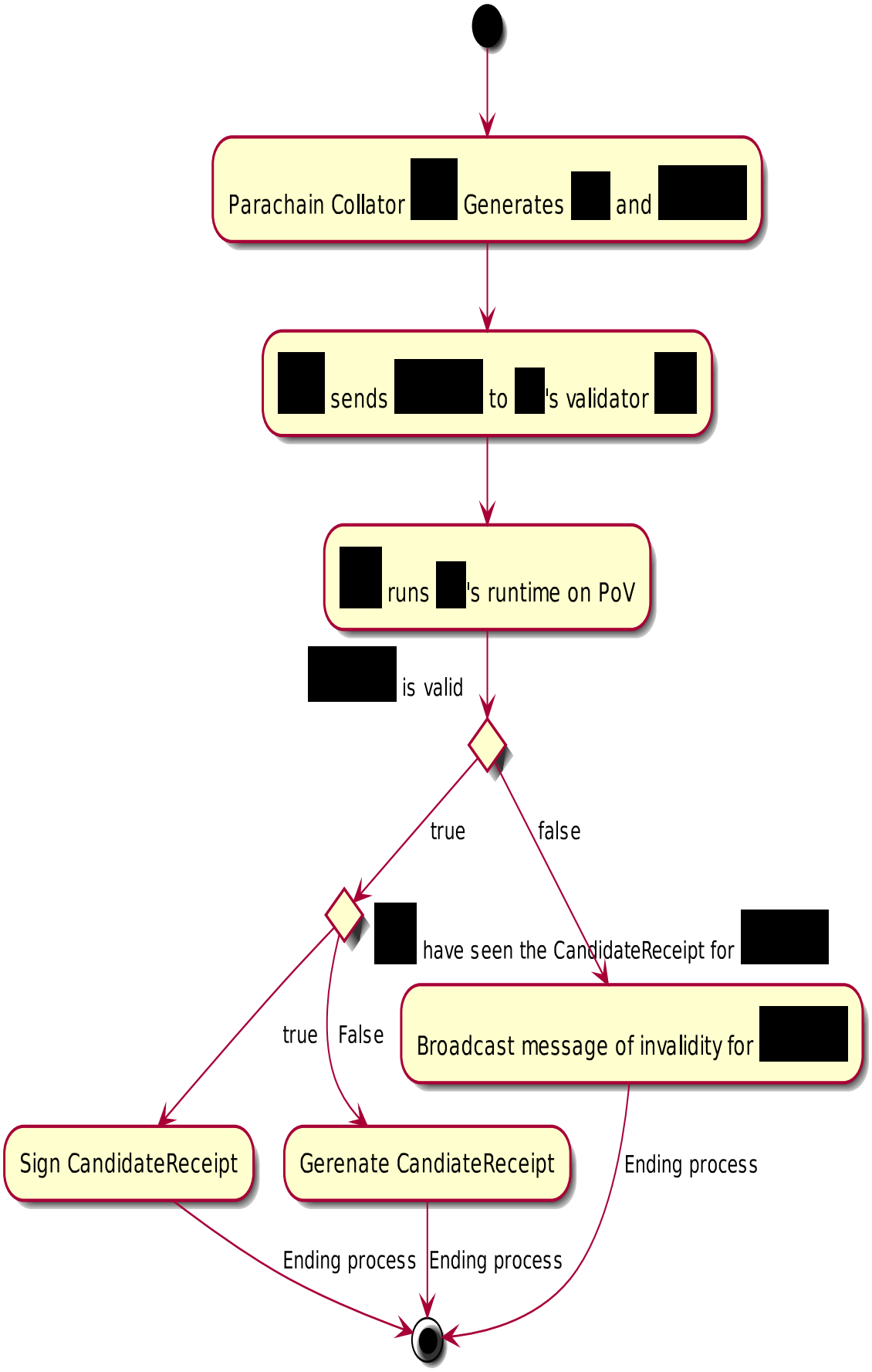

Figure 7.1. Overall process to acheive availability and validity in Polkadot

7.4. Candidate Selection

Collators produce candidates (Definition 7.11) and send those to validators. Validators verify the validity of the received candidates (Algo. 7.1) by executing the validation code, R_ρ, and issue statements (Definition 7.16) about the candidates to connected peers. The validator ensures the that every candidate considered for inclusion has at least one other validator backing it. Candidates without backing are discarded.

The validator must keep track of which candidates were submitted by collators, including which validators back those candidates in order to penalize bad behavior. This is described in more detail in section 7.5

Definition 7.11. *A **candidate**, $C_{\text{coll}}(\text{PoV}_B)$, is issues by collators and contains the PoV block and enough data in order for any validator to verify its validity. A candidate is a tuple of the following format:*

$$C_{\text{coll}}(\text{PoV}_B) := (\text{id}_p, h_b(B_{\substack{\text{relay}\\\text{parent}}}), \text{id}_C, \text{Sig}^{\text{Collator}}_{\text{SR25519}}, \text{head}(B), h_b(\text{PoV}_B))$$

where each value represents:

- id_p*: the Parachain Id this candidate is for.*
- $h_b(B_{\substack{\text{relay}\\\text{parent}}})$*: the hash of the relay chain block that this candidate should be executed in the context of.*
- id_C*: the Collator relay-chain account ID as defined in Definition [@fabio].*
- $\text{Sig}^{\text{Collator}}_{\text{SR25519}}$*: the signature on the 256-bit Blake2 hash of the block data by the collator.*
- $\text{head}(B)$*: the head data (Definition 7.12) of block B.*
- $h_b(\text{PoV}_B)$*: the 32-byte Blake2 hash of the PoV block.*

Definition 7.12. *The **head data**, $\text{head}(B)$, of a parachain block is a tuple of the following format:*

$$\text{head}(B) := (H_i(B), H_p(B), H_r(B))$$

Where $H_i(B)$ is the block number of parachain block B, $H_p(B)$ is the 32-byte Blake2 hash of the parent block header and $H_r(B)$ represents the root of the post-execution state. [@fabio: clarify if H_p is the hash of the header or full block] [@fabio: maybe define those symbols at the start (already defined in the Host spec)?]

Algorithm 7.1. PrimaryValidation(B, π_B, relay chain parent block $B^{\text{relay}}_{\text{parent}}$)

1: Retrieve v_B from the relay chain state at $B^{\text{relay}}_{\text{parent}}$
2: Run Algorithm 7.2 using B, π_B, v_B

Algorithm 7.2. ValidateBlock(B, π_B, v_B)

1: retrieve the runtime code R_ρ that is specified by v_B from the relay chain state.
2: check that the initial state root in π_B is the one claimed in v_B
3: Execute R_ρ on B using π_B to simulate the state.
4: If the execution fails, return fail.
5: Else return success, the new header data h_B and the outgoing messages M.

[@fabio: same as head data?]

7.5. Candidate Backing

Validators back the validity respectively the invalidity of candidates by extending those into candidate receipts as defined in Definition 7.13 and communicate those receipts by issuing statements as defined in Definition 7.16. Validator v needs to perform Algorithm 7.3 to announce the statement of primary validation to the Polkadot network. If the validator receives a statement from another validator, the candidate is confirmed based on algorithm 7.4.

As algorithm 7.3 and 7.4 clarifies, the validator should blacklist collators which send invalid candidates and announce this misbehavior. If another validator claims that an invalid candidates is actually valid, that misbehavior must be announced, too. [@fabio]

The validator tries to back as many candidates as it can, but does not attempt to prioritize specific candidates. Each validator decides on its own - on whatever metric - which candidate will ultimately get included in the block.

DEFINITION 7.13. *A **candidate receipt**, $C_{\text{receipt}}(\text{PoV}_B)$, is an extension of a candidate as defined in Definition 7.11 which includes additional information about the validator which verified the PoV block. The candidate receipt is communicated to other validators by issuing a statement as defined in Definition 7.16.*

This type is a tuple of the following format:

$$C_{\text{receipt}}(\text{PoV}_B) := (\text{id}_p, h_b(B_{\text{parent}}^{\text{relay}}), \text{head}(B), \text{id}_C, \text{Sig}_{\text{SR25519}}^{\text{Collator}}, h_b(\text{PoV}_B), \text{Blake2}b(\text{CC}(\text{PoV}_B)))$$

where each value represents:

- id_p*: the Parachain Id this candidate is for.*
- $h_b(B_{\text{parent}}^{\text{relay}})$*: the hash of the relay chain block that this candidate should be executed in the context of.*
- $\text{head}(B)$*: the head data (Definition 7.12) of block B. [@fabio (collator module relevant?)].*
- id_C*: the collator relay-chain account ID as defined in Definition [@fabio].*
- $\text{Sig}_{\text{SR25519}}^{\text{Collator}}$*: the signature on the 256-bit Blake2 hash of the block data by the collator.*
- $h_b(\text{PoV}_B)$*: the hash of the PoV block.*
- $\text{Blake2}b(\text{CC}(\text{PoV}_B))$*: The hash of the commitments made as a result of validation, as defined in Definition 7.14.*

DEFINITION 7.14. ***Candidate commitments**, $\text{CC}(\text{PoV}_B)$, are results of the execution and validation of parachain (or parathread) candidates whose produced values must be committed to the relay chain. A candidate commitments is represented as a tuple of the following format:*

$$\text{CC}(\text{PoV}_B) := (\mathbb{F}, \text{Enc}_{\text{SC}}(\text{Msg}_0, .., \text{Msg}_n), H_r(B), \text{Option}(R_\rho))$$
$$\text{Msg} := (\mathbb{O}, \text{Enc}_{\text{SC}}(b_0, ..b_n))$$

where each value represents:

- $\mathbb{F}$*: fees paid from the chain to the relay chain validators.*
- Msg*: parachain messages to the relay chain. $\mathbb{O}$ identifies the origin of the messages and is a varying data type (Definition 7.1) and can be one of the following values:*

$$\mathbb{O} = \begin{cases} 0, & \textit{Signed} \\ 1, & \textit{Parachain} \\ 2, & \textit{Root} \end{cases}$$

[@fabio: define the concept of "origin"]

The following SCALE encoded array, $\text{Enc}_{\text{SC}}(b_0, ..b_n)$, contains the raw bytes of the message which varies in size.

- $H_r(B)$*: the root of a block's erasure encoding Merkle tree [@fabio: use different symbol for this?].*
- $\text{Option}(R_\rho)$*: A varying datatype (Definition 7.1) containing the new runtime code for the parachain. [@fabio: clarify further]*

DEFINITION 7.15. *A **Gossip PoV block** is a tuple of the following format:*

$$(h_b(B_{\text{parent}}^{\text{relay}}), h_b(C_{\text{coll}}(\text{PoV}_B)), \text{PoV}_B)$$

where $h_b(B^{\text{relay}}_{\text{parent}})$ *is the block hash of the relay chain being referred to and* $h_b(C_{\text{coll}}(\text{PoV}_B))$ *is the hash of some candidate localized to the same Relay chain block.*

Definition 7.16. *A* ***statement*** *notifies other validators about the validity of a PoV block. This type is a tuple of the following format:*

$$(\text{Stmt}, \text{id}_{\mathbb{V}}, \text{Sig}^{\text{Valdator}}_{\text{SR25519}})$$

where $\text{Sig}^{\text{Validator}}_{\text{SR25519}}$ *is the signature of the validator and* $\text{id}_{\mathbb{V}}$ *refers to the index of validator according to the authority set. [@fabio: define authority set (specified in the Host spec)].* Stmt *refers to a statement the validator wants to make about a certain candidate.* Stmt *is a varying data type (Definition 7.1) and can be one of the following values:*

$$\text{Stmt} = \begin{cases} 0, & \textit{Seconded, followed by: } C_{\text{receipt}}(\text{PoV}_B) \\ 1, & \textit{Validity, followed by: } \text{Blake2}(C_{\text{coll}}(\text{PoV}_B)) \\ 2, & \textit{Invalidity, followed by: } \text{Blake2}(C_{\text{coll}}(\text{PoV}_B)) \end{cases}$$

The main semantic difference between 'Seconded' and 'Valid' comes from the fact that every validator may second only one candidate per relay chain block; this places an upper bound on the total number of candidates whose validity needs to be checked. A validator who seconds more than one parachain candidate per relay chain block is subject to slashing.

Validation does not directly create a seconded statement, but is rather upgraded by the validator when it choses to back a valid candidate as described in Algorithm 7.3.

Algorithm 7.3. PrimaryValidationAnnouncement(PoV_B)

1: **Init** Stmt;
if ValidateBlock(PoV_B) is **valid then**
2: Stmt ← SetValid(PoV_B)
else
4: Stmt ← SetInvalid(PoV_B)
5: BlacklistCollatorOf(PoV_B)
end if
6: Propagate(Stmt)

- ValidateBlock: Validates PoV_B as defined in Algorithm 7.2.
- SetValid: Creates a valid statement as defined in Definition 7.16.
- SetInvalid: Creates an invalid statement as defined in Definition 7.16.
- BlacklistCollatorOf: blacklists the collator which sent the invalid PoV block, preventing any new PoV blocks from being received. The amount of time for blacklisting is unspecified.
- Propagate: sends the statement to the connected peers.

Algorithm 7.4. ConfirmCandidateReceipt($\text{Stmt}_{\text{peer}}$)

1: **Init** Stmt;
2: PoV_B ← Retrieve($\text{Stmt}_{\text{peer}}$)
if ValidateBlock(PoV_B) is **valid then**
if AlreadySeconded($B^{\text{relay}}_{\text{chain}}$) **then**
3: Stmt ← SetValid(PoV_B)
else if then
5: Stmt ← SetSeconded(PoV_B)
end if

else

9: Stmt ← SETINVALID(PoV_B)

10: ANNOUNCEMISBEHAVIOROF(PoV_B)

end if

11: PROPAGATE(Stmt)

- $\text{Stmt}_{\text{peer}}$: a statement received from another validator.
- RETRIEVE: Retrieves the PoV block from the statement (7.16).
- VALIDATEBLOCK: Validates PoV_B as defined in Algorithm 7.2.
- ALREADYSECONDED: Verifies if a parachain block has already been seconded for the given Relay Chain block. Validators that second more than one (1) block per Relay chain block are subject to slashing. More information is available in Definition 7.16.
- SETVALID: Creates a valid statement as defined in Definition 7.16.
- SETSECONDED: Creates a seconded statement as defined in Definition 7.16. Seconding a block should ensure that the next call to ALREADYSECONDED reliably affirms this action.
- SETINVALID: Creates an invalid statement as defined in Definition 7.16.
- BLACKLISTCOLLATOROF: blacklists the collator which sent the invalid PoV block, preventing any new PoV blocks from being received. The amount of time for blacklisting is unspecified.
- ANNOUNCEMISBEHAVIOROF: announces the misbehavior of the validator who claimed a valid statement of invalid PoV block as described in algorithm [@fabio].
- PROPAGATE: sends the statement to the connected peers.

7.5.1. Inclusion of candidate receipt on the relay chain

[@fabio: should this be a subsection?]

DEFINITION 7.17. ***Parachain Block Proposal***, *noted by P_ρ^B is a candidate receipt for a parachain block B for a parachain ρ along with signatures for at least 2/3 of $\mathcal{V}_\rho$.*

A block producer which observe a Parachain Block Proposal as defined in definition 7.17 ⟨syed|may/should|?⟩ include the proposal in the block they are producing according to Algorithm 7.5 during block production procedure.

ALGORITHM 7.5. INCLUDEPARACHAINPROPOSAL(P_ρ^B)

1: TBS

7.6. PoV Distribution

[@fabio]

7.6.1. Primary Validation Disagreement

⟨syed|Parachain|verify⟩ validators need to keep track of candidate receipts (see Definition 7.13) and validation failure messages of their peers. In case, there is a disagreement among the parachain validators about $\bar{B}$, all parachain validators must invoke Algorithm 7.6

ALGORITHM 7.6. PRIMARYVALIDATIONDISAGREEMENT

1: TBS

7.7. Availability

Backed candidates must be widely available for the entire, elected validators set without requiring each of those to maintain a full copy. PoV blocks get broken up into erasure-encoded chunks and each validators keep track of how those chunks are distributed among the validator set. When a validator has to verify a PoV block, it can request the chunk for one of its peers.

Definition 7.18. *The **erasure encoder/decoder** $\mathbf{encode}_{k,n}/\mathbf{decoder}_{k,n}$ is defined to be the Reed-Solomon encoder defined in [?].*

Algorithm 7.7. Erasure-Encode($\bar{B}$: blob defined in Definition 7.10)

TBS
Init Shards ← Make-Shards($\mathcal{V}_\rho, v_B$)
⟨statex|// Create a trie from the shards in order generate the trie nodes⟩
⟨statex|// which are required to verify each chunk with a Merkle root⟩
Init Trie
Init index = 0
for shard ∈ Shards **do**
    Insert(Trie, index, Blake2(shard))
    index = index + 1
end for
⟨statex|// Insert individual chunks into collection (Definition 7.19).⟩
Init Er_B
Init index = 0
for shard ∈ Shards **do**
    Init nodes ← Get-Nodes(Trie, index)
    Add(Er_B, (shard, index, nodes))
    index = index + 1
end for
return Er_B

- Make-Shards(..): return shards for each validator as described in algorithm 7.8. Return value is defined as $(\$_0, ..., \$_n)$ where $\$:= (b_0, ..., b_n)$
- Insert(trie, key, val): insert the given key and value into the trie.
- Get-Nodes(trie, key): based on the key, return all required trie nodes in order to verify the corresponding value for a (unspecified) Merkle root. Return value is defined as $(\mathbb{N}_0, ..., \mathbb{N}_n)$ where $\mathbb{N} := (b_0, ..., b_n)$.
- Add(sequence, item): add the given item to the sequence.

Algorithm 7.8. Make-Shards($\mathcal{V}_\rho, v_B$)

⟨statex| // Calculate the required values for Reed-Solomon.⟩
⟨statex| // Calculate the required lengths.⟩
Init $\text{Shard}_{\text{data}} = \frac{(|\mathcal{V}_\rho| - 1)}{3} + 1$
Init $\text{Shard}_{\text{parity}} = |\mathcal{V}_\rho| - \frac{(|\mathcal{V}_\rho| - 1)}{3} - 1$
Init $\text{base}_{\text{len}} = \begin{cases} 0 & \text{if} |\mathcal{V}_\rho| \bmod \text{Shard}_{\text{data}} = 0 \\ 1 & \text{if} |\mathcal{V}_\rho| \bmod \text{Shard}_{\text{data}} \neq 0 \end{cases}$

4: **Init** $\text{Shard}_{\text{len}} = \text{base}_{\text{len}} + (\text{base}_{\text{len}} \bmod 2)$
⟨statex|// Prepare shards, each padded with zeroes.⟩
⟨statex|// Shards $:= (\$_0, ..., \$_n)$ where $\$:= (b_0, ..., b_n)$⟩
5: **Init** Shards
for $n \in (\text{Shard}_{\text{data}} + \text{Shard}_{\text{partiy}})$ **do**
6: ADD(Shards, $(0_0, ..0_{\text{Shard}_{\text{len}}})$)
end for
⟨statex|// Copy shards of v_b into each shard.⟩
for (chunk, shard) $\in$ (TAKE($\text{Enc}_{\text{SC}}(v_B)$, $\text{Shard}_{\text{len}}$), Shards) **do**
7: **Init** len $\leftarrow$ MIN($\text{Shard}_{\text{len}}$, |chunk|)
8: shard $\leftarrow$ COPY-FROM(chunk, len)
end for
⟨statex|// Shards contains split shards of v_B.⟩
return Shards

- ADD(sequence, item): add the given item to the sequence.
- TAKE(sequence, len): iterate over len amount of bytes from sequence on each iteration. If the sequence does not provide len number of bytes, then it simply uses what's available.
- MIN(num1, num2): return the minimum value of num1 or num2.
- COPY-FROM(source, len): return len amount of bytes from source.

DEFINITION 7.19. *The **collection of erasure-encoded chunks** of $\bar{B}$, denoted by:*

$$\text{Er}_B := (e_1, ..., e_n)$$

is defined to be the output of the Algorithm 7.7. Each chunk is a tuple of the following format:

$$e := (\$, I, (\mathbb{N}_0, ..., \mathbb{N}_n))$$
$$\$:= (b_0, ..., b_n)$$
$$\mathbb{N} := (b_0, ..., b_n)$$

where each value represents:

- *$\$$: a byte array containing the erasure-encoded shard of data.*
- *I: the unsigned 32-bit integer representing the index of this erasure-encoded chunk of data.*
- *$(\mathbb{N}_0, ..., \mathbb{N}_n)$: an array of inner byte arrays, each containing the nodes of the Trie in order to verify the chunk based on the Merkle root.*

7.8. DISTRIBUTION OF CHUNKS

Following the computation of Er_B, v must construct the $\bar{B}$ Availability message defined in Definition 7.20. And distribute them to target validators designated by the Availability Networking Specification [?].

DEFINITION 7.20. ***PoV erasure chunk message*** $M_{\text{PoV}_{\bar{B}}}(i)$ *is TBS*

7.9. ANNOUNCING AVAILABILITY

When validator v receives its designated chunk for $\bar{B}$ it needs to broadcast Availability vote message as defined in Definition7.21

DEFINITION 7.21. ***Availability vote message*** $M_{\text{PoV}}^{\text{Avail},v_i}$ *TBS*

Some parachains have blocks that we need to vote on the availability of, that is decided by $>2/3$ of validators voting for availability. ⟨syed|For 100 parachain and 1000 validators this will involve putting 100k items of data and processing them on-chain for every relay chain block, hence we want to use bit operations that will be very efficient. We describe next what operations the relay chain runtime uses to process these availability votes.|this is not really relevant to the spec⟩

Definition 7.22. *An **availability bitfield** is signed by a particular validator about the availability of pending candidates. It's a tuple of the following format:*

$$(u32, ...)$$

[@fabio]

For each parachain, the relay chain stores the following data:
1) availability status, 2) candidate receipt, 3) candidate relay chain block number
where availability status is one of {no candidate, to be determined, unavailable, available} .
For each block, each validator v signs a message
Sign(bitfield b_v, block hash h_b)
where the ith bit of b_v is 1 if and only if

1. the availability status of the candidate receipt is "to be determined" on the relay chain at block hash h_b **and**
2. v has the erasure coded chunk of the corresponding parachain block to this candidate receipt.

These signatures go into a relay chain block.

7.9.1. Processing on-chain availability data

This section explains how the availability attestations stored on the relay chain, as described in Section ??, are processed as follows:

1. The relay chain stores the last vote from each validator on chain. For each new signature, the relay chain checks if it is for a block in this chain later than the last vote stored from this validator. If it is the relay chain updates the stored vote and updates the bitfield b_v and block number of the vote.
2. For each block within the last t blocks where t is some timeout period, the relay chain computes a bitmask bm_n (n is block number). This bitmask is a bitfield that represents whether the candidate considered in that block is still relevant. That is the ith bit of bm_n is 1 if and only if for the ith parachain, (a) the availability status is to be determined and (b) candidate block number $\leq n$
3. The relay chain initialises a vector of counts with one entry for each parachain to zero. After executing the following algorithm it ends up with a vector of counts of the number of validators who think the latest candidates is available.
    1. The relay chain computes b_v and bm_n where n is the block number of the validator's last vote
    2. For each bit in b_v and bm_n add the ith bit to the ith count.
4. For each count that is $>2/3$ of the number of validators, the relay chain sets the candidates status to "available". Otherwise, if the candidate is at least t blocks old, then it sets its status to "unavailable".
5. The relay chain acts on available candidates and discards unavailable ones, and then clears the record, setting the availability status to "no candidate". Then the relay chain accepts new candidate receipts for parachains that have "no candidate" status and once any such new candidate receipts is included on the relay chain it sets their availability status as "to be determined".

Algorithm 7.9. Relay chain's signature processing

1: TBD (from text above)

Based on the result of Algorithm 7.9 the validator node should mark a parachain block as either available or eventually unavailable according to definitions 7.23 and 7.24

DEFINITION 7.23. *Parachain blocks for which the corresponding blob is noted on the relay chain to be **available**, meaning that the candidate receipt has been voted to be available by 2/3 validators.*

After a certain time-out in blocks since we first put the candidate receipt on the relay chain if there is not enough votes of availability the relay chain logic decides that a parachain block is unavailable, see 7.9.

DEFINITION 7.24. *An **unavailabile** parachain block is TBS*

/syedSo to be clear we are not announcing unavailability we just keep it for grand pa vote

7.10. PUBLISHING ATTESTATIONS

⟨syed||this is out of place. We can mentioned that we have two type of (validity) attestations in the intro but we just need to spec each attestation in its relevant section (which we did with the candidate receipt). [move this to intro]⟩ We have two type of attestations, primary and secondary. Primary attestations are signed by the parachain validators and secondary attestations are signed by secondary checkers and include the VRF that assigned them as a secondary checker into the attestation. Both types of attestations are included in the relay chain block as a transaction. For each parachain block candidate the relay chain keeps track of which validators have attested to its validity or invalidity.

7.11. SECONDARY APPROVAL CHECKING

Once a parachain block is acted on we carry the secondary validity/availability checks as follows. A scheme assigns every validator to one or more PoV blocks to check its validity, see Section 7.11.3 for details. An assigned validator acquires the PoV block (see Section 7.12.0.1) and checks its validity by comparing it to the candidate receipt. If validators notices that an equivocation has happened an additional validity/availability assignments will be made that is described in Section7.11.5.

7.11.1. Approval Checker Assignment

Validators assign themselves to parachain block proposals as defined in Definition 7.17. The assignment needs to be random. Validators use their own VRF to sign the VRF output from the current relay chain block as described in Section 7.11.2. Each validator uses the output of the VRF to decide the block(s) they are revalidating as a secondary checker. See Section 7.11.3 for the detail.

In addition to this assignment some extra validators are assigned to every PoV block which is descirbed in Section 7.11.4.

7.11.2. VRF computation

Every validator needs to run Algorithm 7.10 for every Parachain ρ to determines assignments. [Fix this. It is incorrect so far.]

ALGORITHM 7.10. VRF-FOR-APPROVAL(B,z, s_k)

Require: B: the block to be approved **Require:** z: randomness for approval assignment
Require: s_k: session secret key of validator planning to participate in approval

1: $(\pi, d) \leftarrow \text{VRF}(H_h(B), \text{sk}(z))$

return (π, d)

Where VRF function is defined in [?].

7.11.3. One-Shot Approval Checker Assignment

Every validator v takes the output of this VRF computed by 7.10 mod the number of parachain blocks that we were decided to be available in this relay chain block according to Definition 7.23 and executed. This will give them the index of the PoV block they are assigned to and need to check. The procedure is formalised in 7.11.

Algorithm 7.11. OneShotAssignment

1: TBS

7.11.4. Extra Approval Checker Assigment

Now for each parachain block, let us assume we want #VCheck validators to check every PoV block during the secondary checking. Note that #VCheck is not a fixed number but depends on reports from collators or fishermen. Lets us #VDefault be the minimum number of validator we want to check the block, which should be the number of parachain validators plus some constant like 2. We set

$$\#\text{VCheck} = \#\text{VDefault} + c_f * \text{total fishermen stake}$$

where c_f is some factor we use to weight fishermen reports. Reports from fishermen about this

Now each validator computes for each PoV block a VRF with the input being the relay chain block VRF concatenated with the parachain index.

For every PoV bock, every validator compares #VCheck − #VDefault to the output of this VRF and if the VRF output is small enough than the validator checks this PoV blocks immediately otherwise depending on their difference waits for some time and only perform a check if it has not seen #VCheck checks from validators who either 1) parachain validators of this PoV block 2) or assigned during the assignment procedure or 3) had a smaller VRF output than us during this time.

More fisherman reports can increase #VCheck and require new checks. We should carry on doing secondary checks for the entire fishing period if more are required. A validator need to keep track of which blocks have #VCheck smaller than the number of higher priority checks performed. A new report can make us check straight away, no matter the number of current checks, or mean that we need to put this block back into this set. If we later decide to prune some of this data, such as who has checked the block, then we'll need a new approach here.

Algorithm 7.12. OneShotAssignment

1: TBS

⟨syed||[so assignees are not announcing their assignment just the result of the approval check I assume]⟩

7.11.5. Additional Checking in Case of Equivocation

In the case of a relay chain equivocation, i.e. a validator produces two blocks with the same VRF, we do not want the secondary checkers for the second block to be predictable. To this end we use the block hash as well as the VRF as input for secondary checkers VRF. So each secondary checker is going to produce twice as many VRFs for each relay chain block that was equivocated. If either of these VRFs is small enough then the validator is assigned to perform a secondary check on the PoV block. The process is formalized in Algorithm 7.13

Algorithm 7.13. EquivocatedAssignment

1: TBS

7.12. THE APPROVAL CHECK

Once a validator has a VRF which tells them to check a block, they announce this VRF and attempt to obtain the block. It is unclear yet whether this is best done by requesting the PoV block from parachain validators or by announcing that they want erasure-encoded chunks.

7.12.0.1. Retrieval

There are two fundamental ways to retrieve a parachain block for checking validity. One is to request the whole block from any validator who has attested to its validity or invalidity. Assigned appoval checker v sends RequestWholeBlock message specified in Definition 7.25 to ⟨syed | | any/all⟩ parachain validator in order to receive the specific parachain block. Any parachain validator receiving must reply with PoVBlockRespose message defined in Definition 7.26

DEFINITION 7.25. *Request Whole Block Message TBS*

DEFINITION 7.26. ***PoV Block Respose Message*** *TBS*

The second method is to retrieve enough erasure-encoded chunks to reconstruct the block from them. In the latter cases an announcement of the form specified in Definition has to be gossiped to all validators indicating that one needs the erasure-encoded chunks.

DEFINITION 7.27. ***Erasuree-coded chunks request message*** *TBS*

On their part, when a validator receive a erasuree-coded chunks request message it response with the message specified in Definition 7.28.

DEFINITION 7.28. ***Erasuree-coded chunks response message*** *TBS*

Assigned appoval checker v must retrieve enough erasure-encoded chunks of the block they are verifying to be able to reconstruct the block and the erasure chunks tree.

7.12.0.2. Reconstruction

After receiving $2f+1$ of erasure chunks every assigned approval checker v needs to recreate the entirety of the erasure code, hence every v will run Algorithm 7.14 to make sure that the code is complete and the subsequently recover the original $\bar{B}$.

ALGORITHM 7.14. RECONSTRUCT-POV-ERASURE(S_{Er_B})

Require: $S_{\mathrm{Er}_B} := (e_{j_1}, m_{j_1}), \cdot, (e_{j_k}, m_{j_k}))$ such that $k > 2f$

1: $\bar{B} \rightarrow$ ERASURE-DECODER$(e_{j_1}, \cdots, e_{j_k})$

if ERASURE-DECODER **failed then**

2: ANNOUNCE-FAILURE

return

end if

5: $\mathrm{Er}_B \rightarrow$ ERASURE-ENCODER$(\bar{B})$

if VERIFY-MERKLE-PROOF(S_{Er_B},Er_B) **failed then**

6: ANNOUNCE-FAILURE

return

end if

return $\bar{B}$

7.12.1. Verification

Once the parachain bock has been obtained or reconstructed the secondary checker needs to execute the PoV block. We declare a the candidate receipt as invalid if one one the following three conditions hold: 1) While reconstructing if the erasure code does not have the claimed Merkle root, 2) the validation function says that the PoV block is invalid, or 3) the result of executing the block is inconsistent with the candidate receipt on the relay chain.

The procedure is formalized in Algorithm

Algorithm 7.15. RevalidatingReconstructedPoV

1: TBS

If everything checks out correctly, we declare the block is valid. This means gossiping an attestation, including a reference that identifies candidate receipt and our VRF as specified in Definition 7.29.

Definition 7.29. ***Secondary approval attetstion message*** *TBS*

7.12.2. Process validity and invalidity messages

When a Black produced receive a Secondary approval attetstion message, it execute Algorithm 7.16 to verify the VRF and may need to judge when enough time has passed.

Algorithm 7.16. VerifyApprovalAttestation

1: TBS

These attestations are included in the relay chain as a transaction specified in

Definition 7.30. ***Approval Attestation Transaction*** *TBS*

Collators reports of unavailability and invalidty specified in Definition [Define these messages] also go onto the relay chain as well in the format specified in Definition

Definition 7.31. ***Collator Invalidity Transaction*** *TBS*

Definition 7.32. ***Collator unavailability Transaction*** *TBS*

7.12.3. Invalidity Escalation

When for any candidate receipt, there are attestations for both its validity and invalidity, then all validators acquire and validate the blob, irrespective of the assignments from section by executing Algorithm 7.14 and 7.15.

We do not vote in GRANDPA for a chain were the candidate receipt is executed until its vote is resolved. If we have n validators, we wait for $>2n/3$ of them to attest to the blob and then the outcome of this vote is one of the following:

If $>n/3$ validators attest to the validity of the blob and $\leq n/3$ attest to its invalidity, then we can vote on the chain in GRANDPA again and slash validators who attested to its invalidity.

If $>n/3$ validators attest to the invalidity of the blob and $\leq n/3$ attest to its validity, then we consider the blob as invalid. If the rely chain block where the corresponding candidate receipt was executed was not finalised, then we never vote on it or build on it. We slash the validators who attested to its validity.

If $>n/3$ validators attest to the validity of the blob and $>n/3$ attest to its invalidity then we consider the blob to be invalid as above but we do not slash validators who attest either way. We want to leave a reasonable length of time in the first two cases to slash anyone to see if this happens.

□

Chapter 8

Message Passing

Disclaimer: this document is work-in-progress and will change a lot until finalization.

8.1. Overview

Polkadot implements two types of message passing mechanisms; vertical passing and horizontal passing.

- Vertical message passing refers to the communication between the parachains and the relay chain. More precisely, when the relay chain sends messages to a parachain, it's "downward message passing". When a parachain sends messages to the relay chain, it's "upward message passing".
- Horizontal message passing refers to the communication between the parachains, only requiring minimal involvement of the relay chain. The relay chain essentially only stores proofs that message where sent and whether the recipient has read those messages.

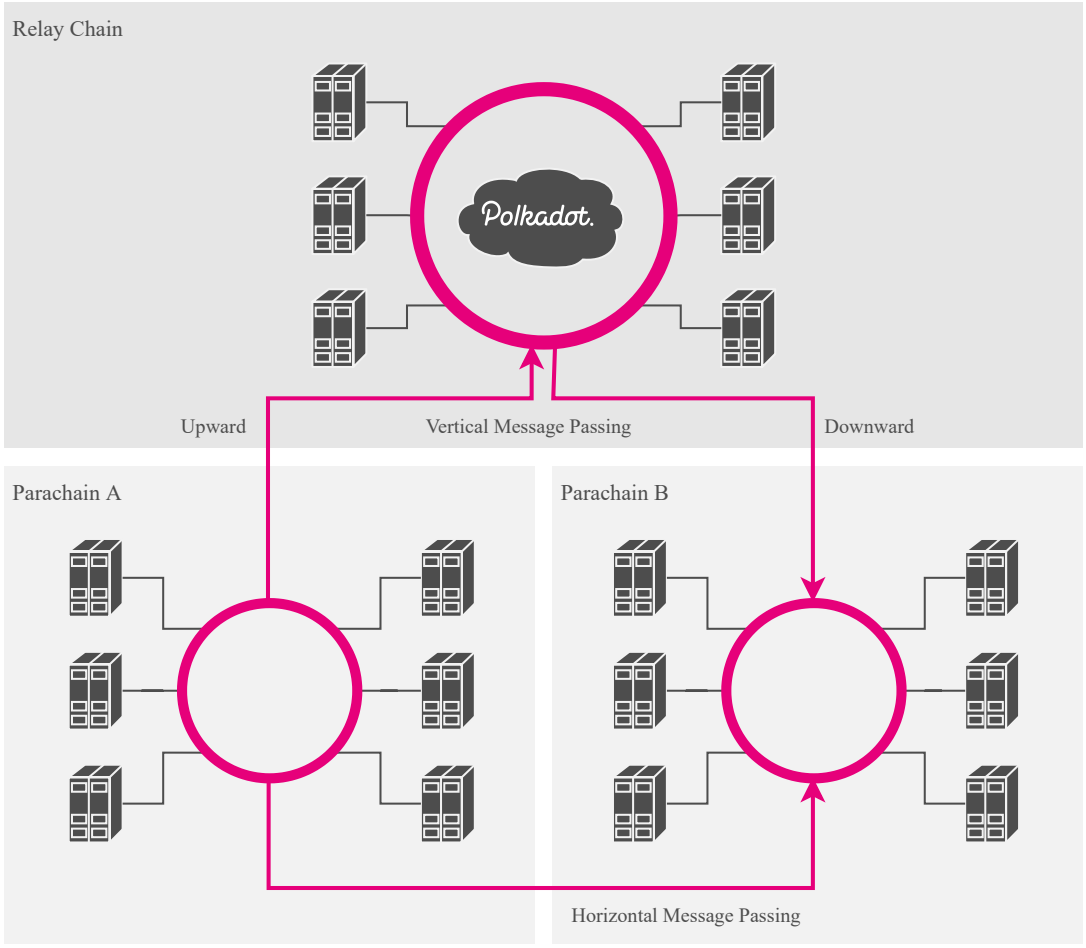

Figure 8.1. Parachain Message Passing Overview

8.2. Message Queue Chain (MQC)

The Message Queue Chain (MQC) is a general hash chain construct created by validators which keeps track of any messages and their order as sent from a sender to an individual recipient. The MQC is used by both HRMP and XCMP.

Each block within the MQC is a triple containing the following fields:
- `parent_hash`: The hash of the previous triple.
- `message_hash`: The hash of the message itself.
- `number`: The relay block number at which the message was sent.

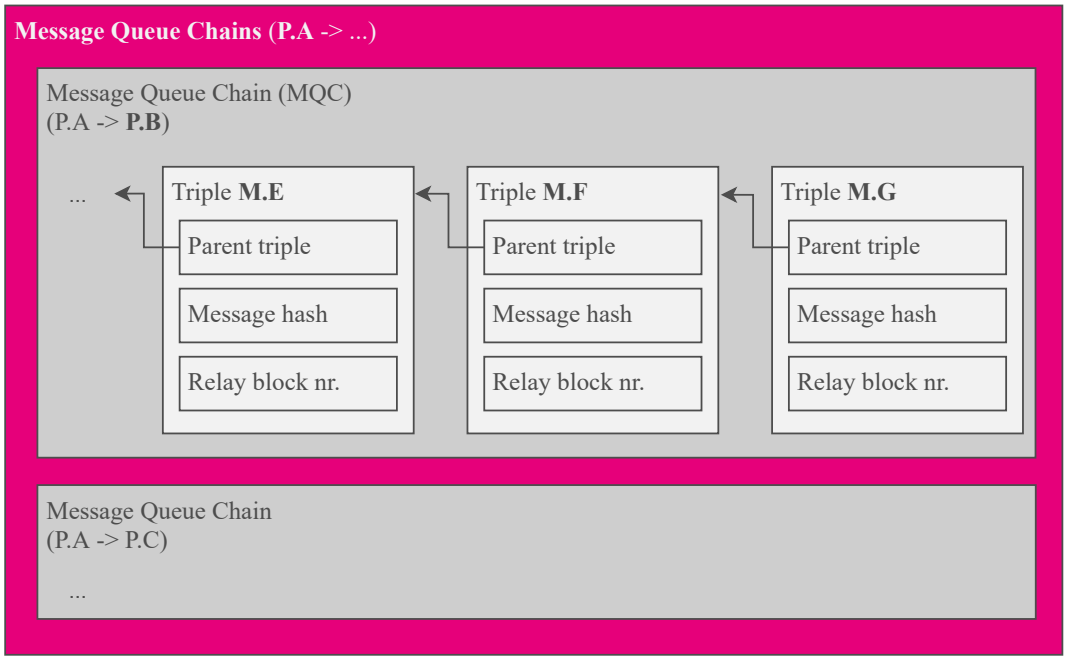

Figure 8.2. Message Queue Chain Overview

A MQC is always specific to one channel. Additional channels require its own, individual MQC. The MQC itself is not saved anywhere, but only provides a final proof of all the received messages. When a validators receives a candidate, it generates the MQC from the messages placed withing `upward_messages`, in ascending order.

8.3. HRMP

Polkadot currently implements the mechanism known as Horizontal Relay-routed Message Passing (HRMP), which fully relies on vertical message passing in order to communicate between parachains. Consequently, this goes against the entire idea of horizontal passing in the first place, since now every message has to be inserted into the relay chain itself, therefore heavily increasing footprint and resource requirements. However, HRMP currently serves as a fast track to implementing cross-chain interoperability. The upcoming replacement of HRMP is Cross-Chain Message Passing (XCMP), which exchanges messages directly between parachains and only updates proofs and read-confirmations on chain. With XCMP, vertical message processing is only used for opening and closing channels.

8.3.1. Channels

A channel is a construct on the relay chain indicating an open, one-directional communication line between a sender and a recipient, including information about how the channel is being used. The channel itself does not contain any messages. A channel construct is created for each, individual communication line.

A channel contains the following fields:
`HrmpChannel`:

- `sender_deposit: int`: staked balances of sender.
- `recipient_deposit: int`: staked balances of recipient.
- `limit_used_places: int`: the maximum number of messages that can be pending in the channel at once.
- `limit_used_bytes: int`: the maximum total size of the messages that can be pending in the channel at once.
- `limit_message_size`: the maximum message size that could be put into the channel.
- `used_places: int`: number of messages used by the sender in this channel.
- `used_bytes: int`: total number of bytes used by the sender in this channel.
- `sealed: bool`: (TOOD: this is not defined in the Impl-Guide) indicator wether the channel is sealed. If it is, then the recipient will no longer accept any new messages.
- `mqc_head`: a head of the MQC for this channel.

This structure is created or overwritten on every start of each session. Individual fields of this construct are updated for every message sent, such as `used_places`, `used_bytes` and `mqc_head`. If the channel is sealed and `used_places` reaches `0` (occurs when a new session begins), this construct is be removed on the *next* session start.

The Runtime maintains a structure of the current, open channels in a map. The key is a tuple of the sender ParaId and the recipient ParaId, where the value is the corresponding `HrmpChannel` structure.

```
channels: map(ParaId, ParaId) => Channel
```

8.3.2. Opening Channels

Polkadot places a certain limit on the amount of channels that can be opened between parachains. Only the the sender can open a channel.

In order to open a channel, the sender must send an opening request to the relay chain. The request is a construct containing the following fields:
`ChOpenRequest`:

- `sender: ParaId`: the ParaId of the sender.
- `recipient: ParaId`: the ParaId of the recipient.
- `confirmed: bool`: indicated whether the recipient has accepted the channel. On request creation, this value is `false`.
- `age: int`: the age of this request, which start at `0` and is incremented by 1 on every session start.

TODO: Shouldn't `ChOpenRequest` also have an `initiator` field? Or can only the sender open an channel?

8.3.2.1. Workflow

Before execution, the following conditions must be valid, otherwise the candidate will be rejected.

- The `sender` and the `recipient` exist.
- `sender` is not the `recipient`.
- There's currently not a active channel established, either seal or unsealed (TODO: what if there's an active closing request pending?).
- There's not already an open channel request for `sender` and `recipient` pending.

- The caller of this function (`sender`) has capacity for a new channel. An open request counts towards the capacity (TODO: where is this defined?).
- The caller of this function (`sender`) has enough funds to cover the deposit.

The PVF executes the following steps:

- Create a `ChOpenRequest` message and inserts it into the `upward_messages` list of the candidate commitments.

Once the candidate is included in the relay chain, the runtime reads the message from `upward_messages` and executes the following steps:

- Reads the message from `upward_messages` of the candidate commitments.
- Reserves a deposit for the caller of this function (`sender`) (TODO: how much?).
- Appends the `ChOpenRequest` request to the pending open request queue.

8.3.3. Accepting Channels

Open channel requests must be accepted by the other parachain.

TODO: How does a Parachain decide which channels should be accepted? Probably off-chain consensus/agreement?

The accept message contains the following fields:

`ChAccept`:

- `index: int`: the index of the open request list.

8.3.3.1. Workflow

Before execution, the following conditions must be valid, otherwise the candidate will be rejected.

- The `index` is valid (the value is within range of the list).
- The `recipient` ParaId corresponds to the ParaId of the caller of this function.
- The caller of this function (`recipient`) has enough funds to cover the deposit.

The PVF executes the following steps:

- Generates a `ChAccept` message and inserts it into the `upward_messages` list of the candidate commitments.

Once the candidate is included in the relay chain, the relay runtime reads the message from `upward_messages` and executes the following steps:

- Reserve a deposit for the caller of this function (`recipient`).
- Confirm the open channel request in the request list by setting the `confirmed` field to `true`.

8.3.4. Closing Channels

Any open channel can be closed by the corresponding sender or receiver. No mutual agreement is required. A close channel request is a construct containing the following fields:

`ChCloseRequest`:

- `initiator: int`: the ParaId of the parachain which initiated this request, either the sender or the receiver.
- `sender: ParaId`: the ParaId of the sender.
- `recipient: ParaId`: the ParaId of the recipient.

8.3.5. Workflow

Before execution, the following conditions must be valid, otherwise the candidate will be rejected.

- There's currently and open channel or a pending open channel request between `sender` and `recipient`.

- The channel is not sealed.
- The caller of the Runtime function is either the `sender` or `recipient`.
- There is not existing close channel request.

The PVF executes the following steps:

- Generates a `ChCloseRequest` message and inserts it into the `upward_messages` list of the candidate commitments.

Once a candidate block is inserted into the relay chain, the relay runtime:

- Reads the message from `upward_message` of the candidate commitments.
- Appends the request `ChCloseRequest` to the pending close request queue.

8.3.6. Sending messages

The Runtime treats messages as SCALE encoded byte arrays and has no concept or understanding of the message type or format itself. Consensus on message format must be established between the two communicating parachains (TODO: SPREE will handle this).

Messages intended to be read by other Parachains are inserted into `horizontal_messages` of the candidate commitments (`CandidateCommitments`), while message which are only intended to be read by the relay chain (such as when opening, accepting or closing channels) are inserted into `upward_messages`.

The messages are included by collators into the committed candidate receipt (), which contains the following fields:

TODO: This should be defined somewhere else, ideally in a backing/validation section (once this document is merged with AnV).

`CommittedCandidateReceipt`:

- `descriptor: CandidateDescriptor`: the descriptor of the candidate.
- `commitments: CandidateCommitments`: the commitments of the candidate receipt.

The candidate descriptor contains the following fields:

`CandidateDescriptor`:

- `para_id: ParaId`: the ID of the para this is a candidate for.
- `relay_parent: Hash`: the hash of the relay chain block this is executed in the context of.
- `collator: CollatorId`: the collator's SR25519 public key.
- `persisted_validation_data_hash: Hash`: the hash of the persisted valdation data. This is extra data derived from the relay chain state which may vary based on bitfields included before the candidate. Therefore, it cannot be derived entirely from the relay parent.
- `pov_hash: Hash`: the how of the PoV block.
- `signature: Signature`: the signature on the Blake2 256-bit hash of the following components of this receipt:
  - `para_id`
  - `relay_parent`
  - `persisted_validation_data_hash`
  - `pov_hash`

The candidate commitments contains the following fields:

`CandidateCommitments`:

- `fees: int`: fees paid from the chain to the relay chian validators
- `horizontal_message: [Message]`: a SCALE encoded arrary containing the messages intended to be received by the recipient parachain.
- `upward_messages: [Message]`: message destined to be interpreted by the relay chain itself.

- `erasure_root: Hash`: the root of a block's erasure encoding Merkle tree.
- `new_validation_code: Option<ValidationCode>`: new validation code for the parachain.
- `head_data: HeadData`: the head-data produced as a result of execution.
- `processed_downward_messages: u32`: the number of messages processed from the DMQ.
- `hrmp_watermark: BlockNumber`: the mark which specifies the block number up to which all inbound HRMP messages are processed.

8.3.7. Receiving Messages

A recipient can check for unread messages by calling into the `downward_messages` function of the relay runtime (TODO: currently it's not really clear how a recipient will check for new messages).
Params:

- `id: ParaId`: the ParaId of the sender.

On success, it returns a SCALE encoded array of messages.

8.4. XCMP

XCMP is a horizontal message passing mechanism of Polkadot which allows Parachains to communicate with each other and to prove that messages have been sent. A core principle is that the relay chain remains as thin as possible in regards to messaging and only contains the required information for the validity of message processing.

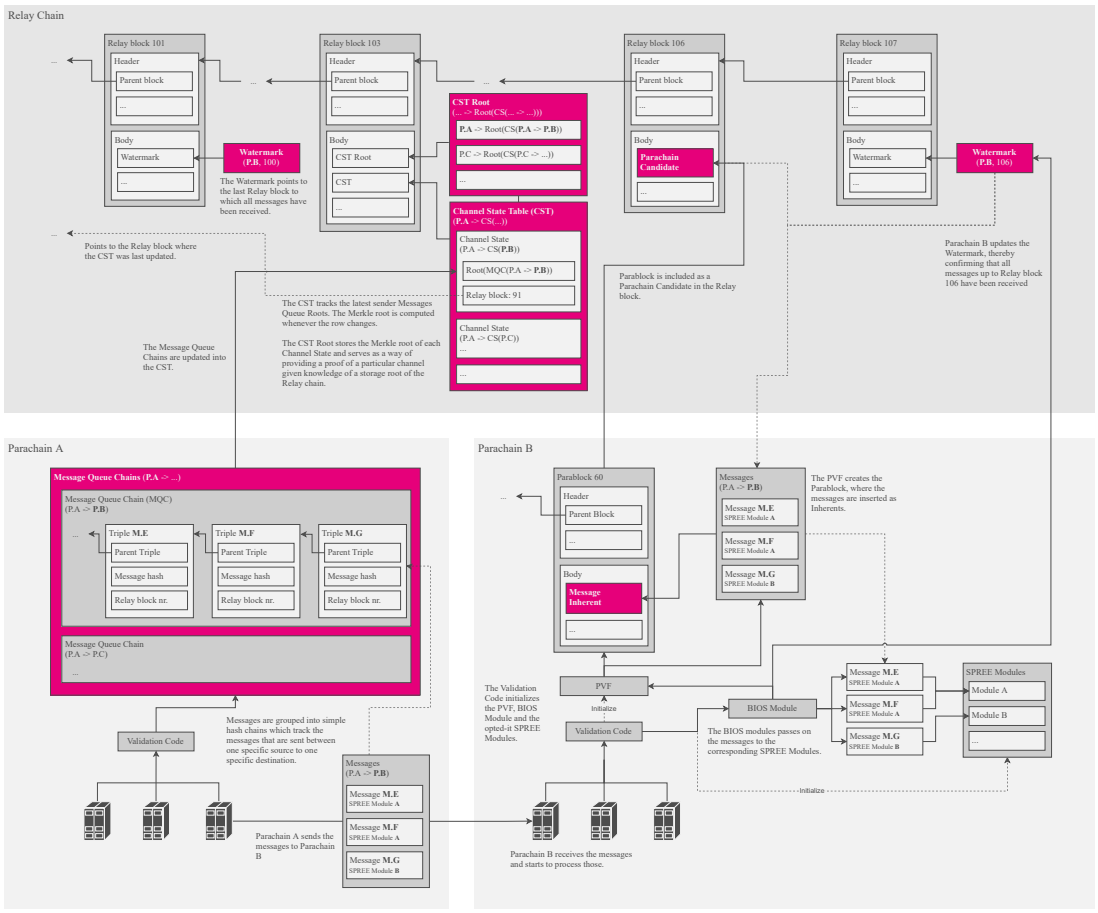

Figure 8.3. Parachain XCMP Overview

The entire XCMP process requires a couple of steps:

- The sender creates a local Message Queue Chain (MQC) of the messages it wants to send and inserts the Merkle root into a structure on the relay chain, known as the Channel State Table (CST).
- The messages are sent to the recipient and contain the necessary data in order to reproduce the MQC.
- The BIOS module of the recipient process those messages. The messages are then inserted into the next parablock body as inherent extrinsics.
- Once that parablock is inserted into the relay chain, the recipient then updates the Watermark, which points to the relay block number which includes the parablock. This serves as an indicator that the receiving parachain has processed messages up to that relay block.

Availabilty

- The messages created by the sender must be kept available for at least one day. When AnV assigns validators to check the validity of the sending parachains parablocks, it can load the data from the CST, which includes the information required in order to regenerate the MQC.
- ...

8.4.1. CST: Channel State Table

The Channel State Table (CST) is a map construct on the relay chain which keeps track of every MQC generated by a single sender. The corresponding value is a list of pairs, where each pair contains the ParaId of the recipient, the Merkle root of MQC heads and the relay block number where that item was last updated in the CST. This provides a mechanism for receiving parachains to easily verify messages sent from a specific source.

```
cst: map ParaId => [ChannelState]
```

`ChannelState`:

- `last_updated: BlockNumber`: the relay block number where the CST was last updated.
- `mqc_root: Option<Hash>`: The Merkle root of all MQC heads where the parachain is the sender. This item is `None` in case there is no prior message.

Besides the CST, there's also a CST Root, which is an additional map construct and contains an entry for every sender and the corresponding Merkle root of each `ChannelState` in the CST.

```
cst_roots: map ParaId => Hash
```

When a PoV block on the recipient is created, the collator which builds that block fetches the pairs of the sender from the CST and creates its own Merkle root. When that PoV block is sent to the validator, the validator can just fetch the Merkle root from the CST Root and verify the PoV block without requiring the full list of pairs.

8.4.2. Message content

All messages sent to the recipient must contain enough information in order for the recipient to verify those messages with the CST. This includes the necessary Merkle trie nodes, the parent triple of each individual MQC block and the messages themselves. The recipient then recreates the MQC and verifies it against the CST.

8.4.3. Watermark

Collators of the recipient insert the messages into their parablock as Inherents and publish the parablock to the relay chain. Once included, the watermark is updated and points to the relay chain block number where the inclusion ocurred.

```
watermark: map ParaId => (BlockNumber, ParaId)
```

8.5. SPREE

...

□

Appendix A

Cryptographic Algorithms

A.1. Hash Functions

A.2. BLAKE2

BLAKE2 is a collection of cryptographic hash functions known for their high speed. Their design closely resembles BLAKE which has been a finalist in the SHA-3 competition.

Polkadot is using the Blake2b variant which is optimized for 64-bit platforms. Unless otherwise specified, the Blake2b hash function with a 256-bit output is used whenever Blake2b is invoked in this document. The detailed specification and sample implementations of all variants of Blake2 hash functions can be found in RFC 7693 [?].

A.3. Randomness

A.4. VRF

A.5. Cryptographic Keys

Various types of keys are used in Polkadot to prove the identity of the actors involved in the Polkadot Protocols. To improve the security of the users, each key type has its own unique function and must be treated differently, as described by this Section.

Definition A.1. ***Account key*** $(\mathbf{sk}^a, \mathbf{pk}^a)$ *is a key pair of type of either of schemes listed in Table A.1:*

Key scheme	*Description*
SR25519	*Schnorr signature on Ristretto compressed Ed25519 points as implemented in [?]*
ED25519	*The standard ED25519 signature complying with [?]*
secp256k1	*Only for outgoing transfer transactions*

Table A.1. *List of the public key scheme which can be used for an account key*

An account key can be used to sign transactions among other accounts and blance-related functions.

There are two prominent subcategories of account keys namely "stash keys" and "controller keys", each being used for a different function, as described below.

Definition A.2. *The* ***Stash key*** *is a type of account key that holds funds bonded for staking (described in Section A.5.1) to a particular controller key (defined in Definition A.3). As a result, one may actively participate with a stash key keeping the stash key offline in a secure location. It can also be used to designate a Proxy account to vote in governance proposals, as described in A.5.2. The Stash key holds the majority of the users' funds and should neither be shared with anyone, saved on an online device, nor used to submit extrinsics.*

DEFINITION A.3. *The **Controller key** is a type of account key that acts on behalf of the Stash account. It signs transactions that make decisions regarding the nomination and the validation of the other keys. It is a key that will be in direct control of a user and should mostly be kept offline, used to submit manual extrinsics. It sets preferences like payout account and commission, as described in A.5.4. If used for a validator, it certifies the session keys, as described in A.5.5. It only needs the required funds to pay transaction fees [key needing fund needs to be defined].*

Keys defined in Definitions A.1, A.2 and A.3 are created and managed by the user independent of the Polkadot implementation. The user notifies the network about the used keys by submitting a transaction, as defined in A.5.2 and A.5.5 respectively.

DEFINITION A.4. ***Session keys** are short-lived keys that are used to authenticate validator operations. Session keys are generated by the Polkadot Host and should be changed regularly due to security reasons. Nonetheless, no validity period is enforced by the Polkadot protocol on session keys. Various types of keys used by the Polkadot Host are presented in Table A.2:*

Protocol	*Key scheme*
GRANDPA	*ED25519*
BABE	*SR25519*
I'm Online	*SR25519*
Parachain	*SR25519*

***Table A.2.** List of key schemes which are used for session keys depending on the protocol*

Session keys must be accessible by certain Polkadot Host APIs defined in Appendix D. Session keys are *not* meant to control the majority of the users' funds and should only be used for their intended purpose. [key managing fund need to be defined]

A.5.1. Holding and staking funds

To be specced

A.5.2. Creating a Controller key

To be specced

A.5.3. Designating a proxy for voting

To be specced

A.5.4. Controller settings

To be specced

A.5.5. Certifying keys

Due to security considerations and Runtime upgrades, the session keys are supposed to be changed regularly. As such, the new session keys need to be certified by a controller key before putting them in use. The controller only needs to create a certificate by signing a session public key and broadcasting this certificate via an extrinsic. [spec the detail of the data structure of the certificate etc.]

□

Appendix B

Auxiliary Encodings

B.1. SCALE Codec

The Polkadot Host uses *Simple Concatenated Aggregate Little-Endian" (SCALE) codec* to encode byte arrays as well as other data structures. SCALE provides a canonical encoding to produce consistent hash values across their implementation, including the Merkle hash proof for the State Storage.

Definition B.1. *The **SCALE codec** for **Byte array** A such that*

$$A := (b_0, b_1, ..., b_{n-1})$$

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

$$\text{Enc}_{\text{SC}}(A) := \text{Enc}^{\text{Len}}_{\text{SC}}(\|A\|)||A$$

where $\text{Enc}^{\text{Len}}_{\text{SC}}$ is defined in Definition B.13.

Definition B.2. *The **SCALE codec** for **Tuple** T such that:*

$$T := (A_1, \ldots, A_n)$$

*Where A_i's are values of **different types**. It is formally referred to as $\text{Enc}_{\text{SC}}(T)$ and defined as:*

$$\text{Enc}_{\text{SC}}(T) := \text{Enc}_{\text{SC}}(A_1)||\text{Enc}_{\text{SC}}(A_2)||\ldots||\,\text{Enc}_{\text{SC}}(A_n)$$

In case of a tuple (or struct), the knowledge of the shape of data is not encoded even though it is necessary for decoding. The decoder needs to derive that information from the context where the encoding/decoding is happenning.

Definition B.3. ***Dec$_{SC}$(d)** refers to the decoding of a blob of data. Since the SCALE codec is not self-describing, it's up to the decoder to validate whether the blob of data can be deserialized into the given type or datastructure.*

Definition B.4. *We define **varying data** types, referred to formally as $\mathcal{T}$, to be an ordered set of data types*

$$\mathcal{T} = \{T_1, \ldots, T_n\}$$

*A value **A** of varying date type is a pair $(A_{\text{Type}}, A_{\text{Value}})$ where $A_{\text{Type}} = T_i$ for some $T_i \in \mathcal{T}$ and A_{Value} is its value of type T_i, which can be empty. We define $\text{idx}(T_i) = i - 1$, unless it is explicitly defined as another value in the definition of a particular varying data type.*

In particular, we define two specific varying data which are frequently used in various part of Polkadot Protocol.

Definition B.5. *The **Option** type is a varying data type of $\{\text{None}, T_2\}$ which indicates if data of T_2 type is available (referred to as "some" state) or not (referred to as "empty", "none" or "null" state). The presence of type None, indicated by $\text{idx}(T_{\text{None}}) = 0$, implies that the data corresponding to T_2 type is not available and contains no additional data. Where as the presence of type T_2 indicated by $\text{idx}(T_2) = 1$ implies that the data is available.*

Definition B.6. *The **Result** type is a varying data type of $\{T_1, T_2\}$ which is used to indicate if a certain operation or function was executed successfully (referred to as "ok" state) or not (referred to as "err" state). T_1 implies success, T_2 implies failure. Both types can either contain additional data or are defined as empty type otherwise.*

DEFINITION B.7. *Scale coded for value* $\boldsymbol{A = (A_{\text{Type}}, A_{\text{Value}})}$ *of varying data type* $\mathcal{T} = \{T_1, \ldots, T_n\}$*, formally referred to as* $\text{Enc}_{\text{SC}}(A)$ *is defined as follows:*

$$\text{Enc}_{\text{SC}}(A) := \text{Enc}_{\text{SC}}(\text{Idx}(A_{\text{Type}}))||\text{Enc}_{\text{SC}}(A_{\text{Value}})$$

Where Idx *is encoded in a fixed length integer determining the type of* A*.*

In particular, for the optional type defined in Definition B.4, we have:

$$\text{Enc}_{\text{SC}}((\text{None}, \phi)) := 0_{\mathbb{B}_1}$$

SCALE codec does not encode the correspondence between the value of Idx defined in Definition B.7 and the data type it represents; the decoder needs prior knowledge of such correspondence to decode the data.

DEFINITION B.8. *The* ***SCALE codec*** *for* ***sequence*** S *such that:*

$$S := A_1, \ldots, A_n$$

where A_i*'s are values of* ***the same type*** *(and the decoder is unable to infer value of* n *from the context) is formally referred to as* $\text{Enc}_{\text{SC}}(S)$ *and defined as:*

$$\text{Enc}_{\text{SC}}(S) := \text{Enc}_{\text{SC}}^{\text{Len}}(\|S\|)|\text{Enc}_{\text{SC}}(A_1)|\text{Enc}_{\text{SC}}(A_2)|\ldots|\,\text{Enc}_{\text{SC}}(A_n)$$

where $\text{Enc}_{\text{SC}}^{\text{Len}}$ *is defined in Definition B.13.*

DEFINITION B.9. *SCALE codec for* ***dictionary*** *or* ***hashtable*** D *with key-value pairs* (k_i, v_i)*s such that:*

$$D := \{(k_1, v_1), \ldots, (k_1, v_n)\}$$

is defined the SCALE codec of D *as a sequence of key value pairs (as tuples):*

$$\text{Enc}_{\text{SC}}(D) := \text{Enc}_{\text{SC}}^{\text{Size}}(||D||)|\text{Enc}_{\text{SC}}((k_1, v_1))|\text{Enc}_{\text{SC}}((k_2, v_2))|\ldots|\,\text{Enc}_{\text{SC}}((k_n, v_n))$$

$\text{Enc}_{\text{SC}}^{\text{Size}}$ *is encoded the same way as* $\text{Enc}_{\text{SC}}^{\text{Len}}$ *but argument* size *refers to the number of key-value pairs rather than the length.*

DEFINITION B.10. *The* ***SCALE codec*** *for* ***boolean value*** b *defined as a byte as follows:*

$$\begin{aligned} \text{Enc}_{\text{SC}}:\ & \{\text{False}, \text{True}\} \to \mathbb{B}_1 \\ & b \to \begin{cases} 0 & b = \text{False} \\ 1 & b = \text{True} \end{cases} \end{aligned}$$

DEFINITION B.11. *The* ***SCALE codec,*** $\mathbf{Enc_{SC}}$ *for other types such as fixed length integers not defined here otherwise, is equal to little endian encoding of those values defined in Definition 1.7.*

DEFINITION B.12. *The* ***SCALE codec,*** $\mathbf{Enc_{SC}}$ *for an empty type is defined to a byte array of zero length and depicted as* $\boldsymbol{\phi}$*.*

B.1.1. Length and Compact Encoding

SCALE Length encoding is used to encode integer numbers of variying sizes prominently in an encoding length of arrays:

DEFINITION B.13. ***SCALE Length Encoding,*** $\mathbf{Enc_{SC}^{Len}}$ *also known as compact encoding of a non-negative integer number* n *is defined as follows:*

$$\begin{aligned} \text{Enc}_{\text{SC}}^{\text{Len}}:\ & \mathbb{N} \to \mathbb{B} \\ & n \to b := \begin{cases} l_1 & 0 \leqslant n < 2^6 \\ i_1\, i_2 & 2^6 \leqslant n < 2^{14} \\ j_1\, j_2\, j_3 & 2^{14} \leqslant n < 2^{30} \\ k_1\, k_2 \ldots k_m & 2^{30} \leqslant n \end{cases} \end{aligned}$$

in where the least significant bits of the first byte of byte array b are defined as follows:

$$\begin{array}{rcl} l_1^1\, l_1^0 & = & 00 \\ i_1^1\, i_1^0 & = & 01 \\ j_1^1\, j_1^0 & = & 10 \\ k_1^1\, k_1^0 & = & 11 \end{array}$$

and the rest of the bits of b store the value of n in little-endian format in base-2 as follows:

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

such that:

$$k_1^7 \dots k_1^3\, k_1^2 := m - 4$$

B.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, the Trie keys are broken into 4-bits nibbles. Accordingly, we need a method to encode sequences of 4-bits nibbles into byte arrays canonically. To this aim, we define *hex encoding* function $\text{Enc}_{\text{HE}}(\text{PK})$ as follows:

Definition B.14. *Suppose that* $\text{PK} = (k_1, \dots, k_n)$ *is a sequence of nibbles, then the* ***hex encoding*** *of* PK *is defined as:*

$\text{Enc}_{\text{HE}}(\text{PK}) :=$

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

□

Appendix C

Genesis State Specification

The genesis state is a set of key-value pairs representing the intial state of the Polkadot state storage. It can be retrieved from [?]. While each of those key-value pairs offers important identifyable information to the Runtime, to the Polkadot Host they are a transparent set of arbitrary chain- and network-dependent keys and values. The only exception to this are the `:code` and `:heappages` keys as described in Section 3.1.1 and 3.1.2.1, which are used by the Polkadot Host to initialize the WASM environment and its Runtime. The other keys and values are unspecifed and soley depend on the chain and respectively its corresponding Runtime. On initialization the data should be inserted into the state storage with the `set_storage` Host API, as defined in Section D.1.1.

As such, Polkadot does not defined a formal genesis block. Nonetheless for the compatibility reasons in several algorithms, the Polkadot Host defines the *genesis header* according to Definition C.1. By the abuse of terminalogy, "*genesis block*" refers to the hypothetical parent of block number 1 which holds genisis header as its header.

Definition C.1. *The Polkadot genesis header is a data structure conforming to block header format described in section 3.6. It contains the values depicted in Table C.1:*

Block header field	*Genesis Header Value*
parent_hash	*0*
number	*0*
`state_root`	*Merkle hash of the state storage trie as defined in Definition 2.10 after inserting the genesis state in it.*
extrinsics_root	*0*
digest	*0*

Table C.1. *Genesis header values*

□

Appendix D

Polkadot Host API

The Polkadot Host API is a set of functions that the Polkadot Host exposes to Runtime to access external functions needed for various reasons, such as the Storage of the content, access and manipulation, memory allocation, and also efficiency. The encoding of each data type is specified or referenced in this section. If the encoding is not mentioned, then the default Wasm encoding is used, such as little-endian byte ordering for integers.

Notation D.1. *By $\mathcal{RE}_B$ we refer to the API exposed by the Polkadot Host which interact, manipulate and response based on the state storage whose state is set at the end of the execution of block B.*

Definition D.2. *The **Runtime pointer** type is a `i32` integer representing a pointer to data in memory. This pointer is the primary way to exchange data of fixed/known size between the Runtime and Polkadot Host.*

Definition D.3. *The **Runtime pointer-size** type is an `i64` integer, representing two consecutive `i32` integers. The least significant is a pointer to the data in memory. The most significant provides the size of the data in bytes. This representation is the primary way to exchange data of arbitrary/dynamic sizes between the Runtime and the Polkadot Host.*

Definition D.4. ***Lexicographic ordering** refers to the ascending ordering of bytes or byte arrays, such as:*

$$[0,0,2] < [0,1,1] < [0,2,0] < [1] < [1,1,0] < [2] < [\ldots]$$

The functions are specified in each subsequent subsection for each category of those functions.

D.1. Storage

Interface for accessing the storage from within the runtime.

D.1.1. `ext_storage_set`

Sets the value under a given key into storage.

D.1.1.1. Version 1 - Prototype

```
(func $ext_storage_set_version_1
   (param $key i64) (param $value i64))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 containing the key.
- `value`: a pointer-size as defined in Definition D.3 containing the value.

D.1.2. `ext_storage_get`

Retrieves the value associated with the given key from storage.

D.1.2.1. Version 1 - Prototype

```
(func $ext_storage_get_version_1
```

```
(param $key i64) (result i64))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 containing the key.
- `result`: a pointer-size as defined in Definition D.3 returning the SCALE encoded `Option` as defined in Definition B.5 containing the value.

D.1.3. `ext_storage_read`

Gets the given key from storage, placing the value into a buffer and returning the number of bytes that the entry in storage has beyond the offset.

D.1.3.1. Version 1 - Prototype

```
(func $ext_storage_read_version_1
     (param $key i64) (param $value_out i64) (param $offset u32) (result i64))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 containing the key.
- `value_out`: a pointer-size as defined in Definition D.3 containing the buffer to which the value will be written to. This function will never write more then the length of the buffer, even if the value's length is bigger.
- `offset`: an i32 integer containing the offset beyond the value should be read from.
- `result`: a pointer-size (Definition D.3) pointing to a SCALE encoded `Option` (Definition B.5) containing an unsinged 32-bit interger representing the number of bytes left at supplied `offset`. Returns `None` if the entry does not exists.

D.1.4. `ext_storage_clear`

Clears the storage of the given key and its value.

D.1.4.1. Version 1 - Prototype

```
(func $ext_storage_clear_version_1
   (param $key_data i64))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 containing the key.

D.1.5. `ext_storage_exists`

Checks whether the given key exists in storage.

D.1.5.1. Version 1 - Prototype

```
(func $ext_storage_exists_version_1
   (param $key_data i64) (return i8))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 containing the key.
- `return`:a boolean equal to `true` if the key does exist, `false` if otherwise.

D.1.6. `ext_storage_clear_prefix`

Clear the storage of each key/value pair where the key starts with the given prefix.

D.1.6.1. Version 1 - Prototype

```
(func $ext_storage_clear_prefix_version_1
   (param $prefix i64))
```

Arguments:

- `prefix`: a pointer-size as defined in Definition D.3 containing the prefix.

D.1.7. `ext_storage_append`

Append the SCALE encoded value to a SCALE encoded sequence (Definition B.8) at the given key. This function assumes that the existing storage item is either empty or a SCALE encoded sequence and that the value to append is also SCALE encoded and of the same type as the items in the existing sequence.

To improve performance, this function is allowed to skip decoding the entire SCALE encoded sequence and instead can just append the new item to the end of the existing data and increment the length prefix Enc_{SC}^{Len}.

Warning: If the storage item does not exist or is not SCALE encoded, the storage item will be set to the specified value, represented as a SCALE encoded byte array.

D.1.7.1. Version 1 - Prototype

```
(func $ext_storage_append_version_1
   (param $key i64) (param $value i64))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 containing the key.
- `value`: a pointer-size as defined in Definition D.3 containing the value to be appended.

D.1.8. `ext_storage_root`

Compute the storage root.

D.1.8.1. Version 1 - Prototype

```
(func $ext_storage_root_version_1
   (return i32))
```

Arguments:

- `return`: a 32-bit pointer to the buffer containing the 256-bit Blake2 storage root.

D.1.9. `ext_storage_changes_root`

Compute the root of the Changes Trie as described in Section 3.3.4. The parent hash is a SCALE encoded block hash.

D.1.9.1. Version 1 - Prototype

```
(func $ext_storage_changes_root_version_1
   (param $parent_hash i64) (return i32))
```

Arguments:

- `parent_hash`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded block hash.
- `return`: a 32-bit pointer to the buffer containing the 256-bit Blake2 changes root.

D.1.10. ext_storage_next_key

Get the next key in storage after the given one in lexicographic order (Definition D.4). The key provided to this function may or may not exist in storage.

D.1.10.1. Version 1 - Prototype

```
(func $ext_storage_next_key_version_1
   (param $key i64) (return i64))
```

Arguments:

- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the next key in lexicographic order.

D.1.11. ext_storage_start_transaction

Start a new nested transaction. This allows to either commit or roll back all changes that are made after this call. For every transaction there must be a matching call to either `ext_storage_rollback_transaction` (D.1.12) or `ext_storage_commit_transaction` (D.1.13). This is also effective for all values manipulated using the child storage API (D.2).

Warning: This is a low level API that is potentially dangerous as it can easily result in unbalanced transactions. Runtimes should use high level storage abstractions.

D.1.11.1. Version 1 - Prototype

```
(func $ext_storage_start_transaction_version_1)
```

Arguments:

- None.

D.1.12. ext_storage_rollback_transaction

Rollback the last transaction started by `ext_storage_start_transaction` (D.1.11). Any changes made during that transaction are discarded.

Warning: Panics if there is no open transaction (`ext_storage_start_transaction` (D.1.11) was not called)

D.1.12.1. Version 1 - Prototype

```
(func $ext_storage_rollback_transaction_version_1)
```

Arguments:

- None.

D.1.13. ext_storage_commit_transaction

Commit the last transaction started by `ext_storage_start_transaction` (D.1.11). Any changes made during that transaction are committed to the main state.

Warning: Panics if there is no open transaction (`ext_storage_start_transaction` (D.1.11) was not called)

D.1.13.1. Version 1 - Prototype

```
(func $ext_storage_commit_transaction_version_1)
```

Arguments:

- None.

D.2. Child Storage

Interface for accessing the child storage from within the runtime.

Definition D.5. ***Child storage*** *key is a unprefixed location of the child trie in the main trie.*

D.2.1. `ext_default_child_storage_set`

Sets the value under a given key into the child storage.

D.2.1.1. Version 1 - Prototype

```
(func $ext_default_child_storage_set_version_1
   (param $child_storage_key i64) (param $key i64) (param $value i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `value`: a pointer-size as defined in Definition D.3 indicating the value.

D.2.2. `ext_default_child_storage_get`

Retrieves the value associated with the given key from the child storage.

D.2.2.1. Version 1 - Prototype

```
(func $ext_default_child_storage_get_version_1
   (param $child_storage_key i64) (param $key i64) (result i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the value.

D.2.3. `ext_default_child_storage_read`

Gets the given key from storage, placing the value into a buffer and returning the number of bytes that the entry in storage has beyond the offset.

D.2.3.1. Version 1 - Prototype

```
(func $ext_default_child_storage_read_version_1
   (param $child_storage_key i64) (param $key i64) (param $value_out i64)
   (param $offset u32) (result i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.

- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `value_out`: a pointer-size as defined in Definition D.3 indicating the buffer to which the value will be written to. This function will never write more then the length of the buffer, even if the value's length is bigger.
- `offset`: an i32 integer containing the offset beyond the value should be read from.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the number of bytes written into the **value_out** buffer. Returns `None` if the entry does not exists.

D.2.4. ext_default_child_storage_clear

Clears the storage of the given key and its value from the child storage.

D.2.4.1. Version 1 - Prototype

```
(func $ext_default_child_storage_clear_version_1
     (param $child_storage_key i64) (param $key i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.

D.2.5. ext_default_child_storage_storage_kill

Clears an entire child storage.

D.2.5.1. Version 1 - Prototype

```
(func $ext_default_child_storage_storage_kill_version_1
   (param $child_storage_key i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.

D.2.5.2. Version 2

```
(func $ext_default_child_storage_storage_kill_version_2
   (param $child_storage_key i64) (param $limit u32) (return i8)
```

Arguments

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `limit`: a SCALE encoded `Option` as defined in Definition B.5 containing the `u32` intiger indicationg the limit of child storage entries to delete. This function call wipes **all** pending (non-finalized) changes which should be committed to the specified child storage keys, including deleting up to `limit` number of database entries in lexicographic order. No limit is applied when this value is `None`.
- `result`: a SCALE encoded boolean equal to `false` if there are some keys remaining in the child trie or `true` if otherwise.

D.2.5.3. Version 3

```
(func $ext_default_child_storage_storage_kill_version3
   (param $child_storage_key i64) (param $limit u32) (return i32)
```

Arguments

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `limit`: a SCALE encoded `Option` as defined in Definition B.5 containing the `u32` intiger indicationg the limit of child storage entries to delete. This function call wipes **all** pending (non-finalized) changes which should be committed to the specified child storage keys, including deleting up to `limit` number of database entries in lexicographic order. No limit is applied when this value is `None`.
- `result`: a pointer to the following SCALE encoded varying type:

$$\begin{cases} 0 & \text{No keys remain in the child trie. Followed by } u32. \\ 1 & \text{At least one key still resides. Followed by } u32. \end{cases}$$

  The additional, following integer indicates the number of entries that were deleted by the function call. This must consider the specificed `limit`.

D.2.6. `ext_default_child_storage_exists`

Checks whether the given key exists in the child storage.

D.2.6.1. Version 1 - Prototype

```
(func $ext_default_child_storage_exists_version_1
     (param $child_storage_key i64) (param $key i64) (return i8))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Defintion D.5.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `return`: a boolean equal to `true` if the key does exist, `false` if otherwise..

D.2.7. `ext_default_child_storage_clear_prefix`

Clears the child storage of each key/value pair where the key starts with the given prefix.

D.2.7.1. Version 1 - Prototype

```
(func $ext_default_child_storage_clear_prefix_version_1
     (param $child_storage_key i64) (param $prefix i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `prefix`: a pointer-size as defined in Definition D.3 indicating the prefix.

D.2.8. `ext_default_child_storage_root`

Commits all existing operations and computes the resulting child storage root.

D.2.8.1. Version 1 - Prototype

```
(func $ext_default_child_storage_root_version_1
     (param $child_storage_key i64) (return i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded storage root.

D.2.9. `ext_default_child_storage_next_key`

Gets the next key in storage after the given one in lexicographic order (Definition D.4). The key provided to this function may or may not exist in storage.

D.2.9.1. Version 1 - Prototype

```
(func $ext_default_child_storage_next_key_version_1
    (param $child_storage_key i64) (param $key i64) (return i64))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition D.3 indicating the child storage key as defined in Definition D.5.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the next key in lexicographic order. Returns `None` if the entry cannot be found.

D.3. CRYPTO

Interfaces for working with crypto related types from within the runtime.

DEFINITION D.6. *Cryptographic keys are stored in seperate key stores based on their intended use case. The seperate key stores are identified by a 4-byte ASCII* ***key type identifier****. The following known types are available:*

Id	*Description*
acco	*Key type for the controlling accounts*
babe	*Key type for the Babe module*
gran	*Key type for the Grandpa module*
imon	*Key type for the ImOnline module*
audi	*Key type for the AuthorityDiscovery module*
para	*Key type for the Parachain Validator Key*
asgn	*Key type for the Parachain Assignment Key*

Table D.1. *Table of known key type identifiers*

DEFINITION D.7. ***EcdsaVerifyError*** *is a varying data type as defined in Definition B.4 and specifies the error type when using ECDSA recovery functionality. Following values are possible:*

Id	*Description*
0	*Incorrect value of R or S*
1	*Incorrect value of V*
2	*Invalid signature*

Table D.2. *Table of error types in ECDSA recovery*

D.3.1. `ext_crypto_ed25519_public_keys`

Returns all `ed25519` public keys for the given key identifier from the keystore.

D.3.1.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_public_keys_version_1
   (param $key_type_id i32) (return i64))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key type identifier as defined in Defintion D.6.
- `return`: a pointer-size as defined in Definition D.3 to an SCALE encoded 256-bit public keys.

D.3.2. `ext_crypto_ed25519_generate`

Generates an `ed25519` key for the given key type using an optional BIP-39 seed and stores it in the keystore.

Warning: Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

D.3.2.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_generate_version_1
     (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key type identifier as defined in Definition D.6.
- `seed`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the BIP-39 seed which must be valid UTF8.
- `return`: a 32-bit pointer to the buffer containing the 256-bit public key.

D.3.3. `ext_crypto_ed25519_sign`

Signs the given message with the `ed25519` key that corresponds to the given public key and key type in the keystore.

D.3.3.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_sign_version_1
     (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key type identifier as defined in Definition D.6.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be signed.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the 64-byte signature. This function returns `None` if the public key cannot be found in the key store.

D.3.4. `ext_crypto_ed25519_verify`

Verifies a `ed25519` signature.

D.3.4.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_verify_version_1
     (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.
- `return`: a boolean equal to `true` if the signature is valid or `false` if otherwise.

D.3.5. `ext_crypto_ed25519_batch_verify`

Registers a `ed25519` signature for batch verification. Batch verification must be enabled by calling `ext_crypto_start_batch_verify` as described in Section D.3.18. If batch verification is not enabled, then the signature is verified immediately. To get the result of the verification batch, `ext_crypto_finish_batch_verify` as described in Section D.3.19 must be called.

D.3.5.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_batch_verify_version_1
     (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments

- `sig`: a 32-bit pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.
- `return`: a boolean equal to `true` if the signature is batched or valid, `false` if otherwise.

D.3.6. `ext_crypto_sr25519_public_keys`

Returns all `sr25519` public keys for the given key id from the keystore.

D.3.6.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_public_keys_version_1
   (param $key_type_id i32) (return i64))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key type identifier as defined in D.6.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded 256-bit public keys.

D.3.7. `ext_crypto_sr25519_generate`

Generates an `sr25519` key for the given key type using an optional BIP-39 seed and stores it in the keystore.

Warning: Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

D.3.7.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_generate_version_1
     (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key identifier as defined in Definition D.6.
- `seed`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the BIP-39 seed which must be valid UTF8.

- `return`: a 32-bit pointer to the buffer containing the 256-bit public key.

D.3.8. `ext_crypto_sr25519_sign`

Signs the given message with the `sr25519` key that corresponds to the given public key and key type in the keystore.

D.3.8.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_sign_version_1
    (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key identifier as defined in Definition D.6
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be signed.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the 64-byte signature. This function returns `None` if the public key cannot be found in the key store.

D.3.9. `ext_crypto_sr25519_verify`

Verifies an `sr25519` signature. Only version 1 of this function supports deprecated Schnorr signatures introduced by the *schnorrkel* Rust library version 0.1.1 and should only be used for backward compatibility.

Returns `true` when the verification is either successful or batched. If no batching verification extension is registered, this function will fully verify the signature and return the result. If batching verification is registered, this function will push the data to the batch and return immediately. The caller can then get the result by calling `ext_crypto_finish_batch_verify` (D.3.19).

The verification extension is explained more in detail in `ext_crypto_start_batch_verify` (D.3.18).

D.3.9.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.
- `return`: a boolean equal to `true` if the signature is valid, `false` if otherwise.

D.3.9.2. Version 2 - Prototype

```
(func $ext_crypto_sr25519_verify_version_2
    (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.

- `return`: a boolean equal to `true` if the signature is valid, `false` if otherwise.

D.3.10. ext_crypto_sr25519_batch_verify

Registers a `sr25519` signature for batch verification. Batch verification must be enabled by calling `ext_crypto_start_batch_verify` as described in Section D.3.18. If batch verification is not enabled, then the signature is verified immediately. To get the result of the verification batch, `ext_crypto_finish_batch_verify` as described in Section D.3.19 must be called.

D.3.10.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_batch_verify_version_1
     (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.
- `return`: a boolean equal to `true` if the signature is batched or valid, `false` if otherwise.

D.3.11. ext_crypto_ecdsa_public_keys

Returns all `ecdsa` public keys for the given key id from the keystore.

D.3.11.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_public_keys_version_1
   (param $key_type_id i64) (return i64))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key type identifier as defined in D.6.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded 33-byte compressed public keys.

D.3.12. ext_crypto_ecdsa_generate

Generates an `ecdsa` key for the given key type using an optional BIP-39 seed and stores it in the keystore.

Warning: Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

D.3.12.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_generate_version_1
     (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key identifier as defined in Definition D.6.
- `seed`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the BIP-39 seed which must be valid UTF8.
- `return`: a 32-bit pointer to the buffer containing the 33-byte compressed public key.

D.3.13. ext_crypto_ecdsa_sign

Signs the given message with the `ecdsa` key that corresponds to the given public key and key type in the keystore.

D.3.13.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_sign_version_1
    (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments:

- `key_type_id`: a 32-bit pointer to the key identifier as defined in Definition D.6
- `key`: a 32-bit pointer to the buffer containing the 33-byte compressed public key.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be signed.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID. This function returns `None` if the public key cannot be found in the key store.

D.3.14. `ext_crypto_ecdsa_verify`

Verifies an `ecdsa` signature. Returns `true` when the verification is either successful or batched. If no batching verification extension is registered, this function will fully verify the signature and return the result. If batching verification is registered, this function will push the data to the batch and return immediately. The caller can then get the result by calling `ext_crypto_finish_batch_verify` (D.3.19).

The verification extension is explained more in detail in `ext_crypto_start_batch_verify` (D.3.18).

D.3.14.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 65-byte signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 33-byte compressed public key.
- `return`: a boolean equal to `true` if the signature is valid, `false` if otherwise.

D.3.15. `ext_ecdsa_batch_verify`

Registers a `ecdsa` signature for batch verification. Batch verification must be enabled by calling `ext_crypto_start_batch_verify` as described in Section D.3.18. If batch verification is not enabled, then the signature is verified immediately. To get the result of the verification batch, `ext_crypto_finish_batch_verify` as described in Section D.3.19 must be called.

D.3.15.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_batch_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i8))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition D.3 indicating the message that is to be verified.
- `key`: a 32-bit pointer to the buffer containing the 256-bit public key.

- `return`: a boolean equal to `true` if the signature is batched or valid, `false` if otherwise.

D.3.16. ext_crypto_secp256k1_ecdsa_recover

Verify and recover a `secp256k1` ECDSA signature.

D.3.16.1. Version 1 - Prototype

```
(func $ext_crypto_secp256k1_ecdsa_recover_version_1
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a 32-bit pointer to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definition B.6. On success it contains the 64-byte recovered public key or an error type as defined in Definition D.7 on failure.

D.3.17. ext_crypto_secp256k1_ecdsa_recover_compressed

Verify and recover a `secp256k1` ECDSA signature.

D.3.17.1. Version 1 - Prototype

```
(func $ext_crypto_secp256k1_ecdsa_recover_compressed_version_1
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments:

- `sig`: a 32-bit pointer to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a 32-bit pointer to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definiton B.6. On success it contains the 33-byte recovered public key in compressed form on success or an error type as defined in Definition D.7 on failure.

D.3.18. ext_crypto_start_batch_verify

Starts the verification extension. The extension is a separate background process and is used to parallel-verify signatures which are pushed to the batch with `ext_crypto_ed25519_verify` (D.3.4), `ext_crypto_sr25519_verify` (D.3.9) or `ext_crypto_ecdsa_verify` (D.3.14). Verification will start immediatly and the Runtime can retrieve the result when calling `ext_crypto_finish_batch_verify` (D.3.19).

D.3.18.1. Version 1 - Prototype

```
(func $ext_crypto_start_batch_verify_version_1)
```

Arguments:

- None.

D.3.19. ext_crypto_finish_batch_verify

Finish verifying the batch of signatures since calling `ext_crypto_start_batch_verify` (D.3.18). Blocks until all the signatures are verified. If the batch is empty, this function just returns `true`.

Warning: Panics if no verification extension is registered (`ext_crypto_start_batch_verify` (D.3.18) was not called.)

D.3.19.1. Version 1 - Prototype

```
(func $ext_crypto_finish_batch_verify_version_1
   (return i8))
```

Arguments:

- `return`: a boolean equal to `true` if all signatures are valid or the batch is empty, `false` if otherwise.

D.4. Hashing

Interface that provides functions for hashing with different algorithms.

D.4.1. `ext_hashing_keccak_256`

Conducts a 256-bit Keccak hash.

D.4.1.1. Version 1 - Prototype

```
(func $ext_hashing_keccak_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 256-bit hash result.

D.4.2. `ext_hashing_keccak_512`

Conducts a 512-bit Keccak hash.

D.4.2.1. Version 1 - Prototype

```
(func $ext_hashing_keccak_512_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 512-bit hash result.

D.4.3. `ext_hashing_sha2_256`

Conducts a 256-bit Sha2 hash.

D.4.3.1. Version 1 - Prototype

```
(func $ext_hashing_sha2_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 256-bit hash result.

D.4.4. `ext_hashing_blake2_128`

Conducts a 128-bit Blake2 hash.

D.4.4.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_128_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 128-bit hash result.

D.4.5. `ext_hashing_blake2_256`

Conducts a 256-bit Blake2 hash.

D.4.5.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 256-bit hash result.

D.4.6. `ext_hashing_twox_64`

Conducts a 64-bit xxHash hash.

D.4.6.1. Version 1 - Prototype

```
(func $ext_hashing_twox_64_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 64-bit hash result.

D.4.7. `ext_hashing_twox_128`

Conducts a 128-bit xxHash hash.

D.4.7.1. Version 1 - Prototype

```
(func $ext_hashing_twox_128_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 128-bit hash result.

D.4.8. `ext_hashing_twox_256`

Conducts a 256-bit xxHash hash.

D.4.8.1. Version 1 - Prototype

```
(func $ext_hashing_twox_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the data to be hashed.
- `return`: a 32-bit pointer to the buffer containing the 256-bit hash result.

D.5. Offchain

The offchain workers allow the execution of long-running and possibly non-deterministic tasks (e.g. web requests, encryption/decryption and signing of data, random number generation, CPU-intensive computations, enumeration/aggregation of on-chain data, etc.) which could otherwise require longer than the block execution time. Offchain workers have their own execution environment. This separation of concerns is to make sure that the block production is not impacted by the long-running tasks.

All data and results generated by offchain workers are unique per node and nondeterministic. Information can be propagated to other nodes by submitting a transaction that should be included in the next block. As offchain workers runs on their own execution environment they have access to their own separate storage. There are two different types of storage available which are defined in Definitions F.1 and F.2.

Definition D.8. ***Persistent storage*** *is non-revertible and not fork-aware. It means that any value set by the offchain worker is persisted even if that block (at which the worker is called) is reverted as non-canonical (meaning that the block was surpassed by a longer chain). The value is available for the worker that is re-run at the new (different block with the same block number) and future blocks. This storage can be used by offchain workers to handle forks and coordinate offchain workers running on different forks.*

Definition D.9. ***Local storage*** *is revertible and fork-aware. It means that any value set by the offchain worker triggered at a certain block is reverted if that block is reverted as non-canonical. The value is NOT available for the worker that is re-run at the next or any future blocks.*

Definition D.10. ***HTTP status codes*** *that can get returned by certain Offchain HTTP functions.*

- `0`: *the specified request identifier is invalid.*
- `10`: *the deadline for the started request was reached.*
- `20`: *an error has occurred during the request, e.g. a timeout or the remote server has closed the connection. On returning this error code, the request is considered destroyed and must be reconstructed again.*
- `100-999`: *the request has finished with the given HTTP status code.*

Definition D.11. ***HTTP error*** *is a varying data type as defined in Definition B.4 and specifies the error types of certain HTTP functions. Following values are possible:*

Id	*Description*
0	*The deadline was reached*
1	*There was an IO error while processing the request*
2	*The ID of the request is invalid*

Table D.3. *Table of possible HTTP error types*

D.5.1. `ext_offchain_is_validator`

Check whether the local node is a potential validator. Even if this function returns `1`, it does not mean that any keys are configured or that the validator is registered in the chain.

D.5.1.1. Version 1 - Prototype

```
(func $ext_offchain_is_validator_version_1 (return i8))
```

Arguments:

- `return`: a boolean equal to `true` if the node is a validator, `false` if otherwise.

D.5.2. `ext_offchain_submit_transaction`

Given a SCALE encoded extrinsic, this function submits the extrinsic to the Host's transaction pool, ready to be propagated to remote peers. This process is critical for issuing the `ImOnline` message [refer].

D.5.2.1. Version 1 - Prototype

```
(func $ext_offchain_submit_transaction_version_1 (param $data i64) (return i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the byte array storing the encoded extrinsic.
- `return`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definition B.6. Neither on success or failure is there any additional data provided. The cause of a failure is implementation specific.

D.5.3. `ext_offchain_network_state`

Returns the SCALE encoded, opaque information about the local node's network state.

DEFINITION D.12. *The `OpaqueNetworkState` structure, O_{NS}, is a SCALE encoded blob holding information about the the `libp2p PeerId`, P_{id}, of the local node and a list of `libp2p Multiaddresses`, $(M_0 \ldots M_n)$, the node knows it can be reached at:*

$$O_{\text{NS}} = (P_{\text{id}}, (M_0 \ldots M_n))$$

where:

$$\begin{aligned} P_{\text{id}} &= (b_0 \ldots b_n) \\ M &= (b_0 \ldots b_n) \end{aligned}$$

The information contained in this structure is naturally opaque to the caller of this function.

D.5.3.1. Version 1 - Prototype

```
(func $ext_offchain_network_state_version_1 (result i64))
```

Arguments:

- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definition B.6. On success it contains the `OpaqueNetworkState` structure as defined in Definition D.12. On failure, an empty value is yielded where its cause is implementation specific.

D.5.4. `ext_offchain_timestamp`

Returns current timestamp.

D.5.4.1. Version 1 - Prototype

```
(func $ext_offchain_timestamp_version_1 (result u64))
```

Arguments:

- `result`: an u64 integer indicating the current UNIX timestamp as defined in Definition 1.10.

D.5.5. ext_offchain_sleep_until

Pause the execution until 'deadline' is reached.

D.5.5.1. Version 1 - Prototype

```
(func $ext_offchain_sleep_until_version_1 (param $deadline u64))
```

Arguments:

- `deadline`: an u64 integer specifying the UNIX timestamp as defined in Definition 1.10.

D.5.6. ext_offchain_random_seed

Generates a random seed. This is a truly random non deterministic seed generated by the host environment.

D.5.6.1. Version 1 - Prototype

```
(func $ext_offchain_random_seed_version_1 (result i32))
```

Arguments:

- `result`: a 32-bit pointer to the buffer containing the 256-bit seed.

D.5.7. ext_offchain_local_storage_set

Sets a value in the local storage. This storage is not part of the consensus, it's only accessible by the offchain worker tasks running on the same machine and is persisted between runs.

D.5.7.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_set_version_1
     (param $kind i32) (param $key i64) (param $value i64))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to `1` is used for a persistent storage as defined in Definition D.8 and a value equal to `2` for local storage as defined in Definition D.9.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `value`: a pointer-size as defined in Definition D.3 indicating the value.

D.5.8. ext_offchain_local_storage_clear

Remove a value from the local storage.

D.5.8.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_clear_version_1
     (param $kind i32) (param $key i64))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to `1` is used for a persistent storage as defined in Definition D.8 and a value equal to `2` for local storage as defined in Definition D.9.
- key: a pointer-size as defined in Definition D.3 indicating the key.

D.5.9. ext_offchain_local_storage_compare_and_set

Sets a new value in the local storage if the condition matches the current value.

D.5.9.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_compare_and_set_version_1
     (param $kind i32) (param $key i64)
     (param $old_value i64) (param $new_value i64)
     (result i8))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to `1` is used for a persistent storage as defined in Definition D.8 and a value equal to `2` for local storage as defined in Definition D.9.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `old_value`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the old key.
- `new_value`: a pointer-size as defined in Definition D.3 indicating the new value.
- `result`: a boolean equal to `true` if the new value has been set, `false` if otherwise.

D.5.10. ext_offchain_local_storage_get

Gets a value from the local storage.

D.5.10.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_get_version_1
     (param $kind i32) (param $key i64) (result i64))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to `1` is used for a persistent storage as defined in Definition D.8 and a value equal to `2` for local storage as defined in Definition D.9.
- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the value or the corresponding key.

D.5.11. ext_offchain_http_request_start

Initiates a HTTP request given by the HTTP method and the URL. Returns the id of a newly started request.

D.5.11.1. Version 1 - Prototype

```
(func $ext_offchain_http_request_start_version_1
     (param $method i64) (param $uri i64) (param $meta i64) (result i64))
```

Arguments:

- `method`: a pointer-size as defined in Definition D.3 indicating the HTTP method. Possible values are `''GET''` and `''POST''`.
- `urli`: a pointer-size as defined in Definition D.3 indicating the URI.
- `meta`: a future-reserved field containing additional, SCALE encoded parameters. Currently, an empty array should be passed.

- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definition B.6 containing the i16 ID of the newly started request. On failure no additionally data is provided. The cause of failure is implementation specific.

D.5.12. `ext_offchain_http_request_add_header`

Append header to the request. Returns an error if the request identifier is invalid, `http_response_wait` has already been called on the specified request identifier, the deadline is reached or an I/O error has happened (e.g. the remote has closed the connection).

D.5.12.1. Version 1 - Prototype

```
(func $ext_offchain_http_request_add_header_version_1
     (param $request_id i32) (param $name i64) (param $value i64) (result i64))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `name`: a pointer-size as defined in Definition D.3 indicating the HTTP header name.
- `value`: a pointer-size as defined in Definition D.3 indicating the HTTP header value.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definition B.6. Neither on success or failure is there any additional data provided. The cause of failure is implemenation specific.

D.5.13. `ext_offchain_http_request_write_body`

Writes a chunk of the request body. Returns a non-zero value in case the deadline is reached or the chunk could not be written.

D.5.13.1. Version 1 - Prototype

```
(func $ext_offchain_http_request_write_body_version_1
     (param $request_id i32) (param $chunk i64) (param $deadline i64) (result
i64))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `chunk`: a pointer-size as defined in Definition D.3 indicating the chunk of bytes. Writing an empty chunk finalizes the request.
- `deadline`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the UNIX timestamp as defined in Definition 1.10. Passing `None` blocks indefinitely.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined Definition B.6. On success, no additional data is provided. On error it contains the HTTP error type as defined in Definition D.11.

D.5.14. `ext_offchain_http_response_wait`

Returns an array of request statuses (the length is the same as IDs). Note that if deadline is not provided the method will block indefinitely, otherwise unready responses will produce `DeadlineReached` status.

D.5.14.1. Version 1- Prototype

```
(func $ext_offchain_http_response_wait_version_1
     (param $ids i64) (param $deadline i64) (result i64))
```

Arguments:

- `ids`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded array of started request IDs.
- `deadline`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the UNIX timestamp as defined in Definition 1.10. Passing `None` blocks indefinitely.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded array of request statuses as defined in Definition D.10.

D.5.15. `ext_offchain_http_response_headers`

Read all HTTP response headers. Returns an array of key/value pairs. Response headers must be read before the response body.

D.5.15.1. Version 1 - Prototype

```
(func $ext_offchain_http_response_headers_version_1
     (param $request_id i32) (result i64))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `result`: a pointer-size as defined in Definition D.3 indicating a SCALE encoded array of key/value pairs.

D.5.16. `ext_offchain_http_response_read_body`

Reads a chunk of body response to the given buffer. Returns the number of bytes written or an error in case a deadline is reached or the server closed the connection. If `0` is returned it means that the response has been fully consumed and the `request_id` is now invalid. This implies that response headers must be read before draining the body.

D.5.16.1. Version 1 - Prototype

```
(func $ext_offchain_http_response_read_body_version_1
     (param $request_id i32) (param $buffer i64) (param $deadline i64) (result
i64))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `buffer`: a pointer-size as defined in Definition D.3 indicating the buffer where the body gets written to.
- `deadline`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the UNIX timestamp as defined in Definition 1.10. Passing `None` will block indefinitely.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Result` as defined in Definition B.6. On success it contains an i32 integer specifying the number of bytes written or a HTTP error type as defined in Definition D.11 on faiure.

D.5.17. `ext_offchain_set_authorized_nodes`

Set the authorized nodes which are allowed to connect to the local node. This function is offered by the Substrate codebase and is primarily used for custom, non-Polkadot/Kusama chains. It is not required for the public and open Polkadot protocol.

D.5.17.1. Version 1 - Prototype

```
(func $ext_offchain_set_authorized_nodes_version_1
```

```
(param $nodes i64) (param $authorized_only i32)
```

Arguments:

- `nodes`: a pointer-size as defined in Definition D.3 indicating the buffer of the SCALE encoded array of `libp2p PeerId`'s. Invalid `PeerId`'s are silently ignored.
- `authorized_only`: If set to `1`, then only the authorized nodes are allowed to connect to the local node (whitelist). All other nodes are rejected. If set to `0`, then no such restriction is placed.

D.6. Offchain Index

Interface that provides functions to access the Offchain database.

D.6.1. `ext_offchain_index_set`

Write a key value pair to the offchain database in a buffered fashion.

D.6.1.1. Version 1 - Prototype

```
(func $ext_offchain_index_set_version_1
     (param $key i64) (param $value i64))
```

Arguments

- `key`: a pointer-size as defined in Definition D.3 indicating the key.
- `value`: a pointer-size as defined in Definition D.3 indicating the value.

D.6.2. `ext_offchain_index_clear`

Remove a key and its associated value from the offchain database.

D.6.2.1. Version 1 - Prototype

```
(func $ext_offchain_index_clear_version_1
     (param $key i64))
```

Arguments

- `key`: a pointer-size as defined in Definition D.3 indicating the key.

D.7. Trie

Interface that provides trie related functionality.

D.7.1. `ext_trie_blake2_256_root`

Compute a 256-bit Blake2 trie root formed from the iterated items.

D.7.1.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_root_version_1
     (param $data i64) (result i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the iterated items from which the trie root gets formed. The items consist of a SCALE encoded array containing arbitrary key/value pairs.

- `result`: a 32-bit pointer to the buffer containing the 256-bit trie root.

D.7.2. `ext_trie_blake2_256_ordered_root`

Compute a 256-bit Blake2 trie root formed from the enumerated items.

D.7.2.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_ordered_root_version_1
     (param $data i64) (result i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the enumerated items from which the trie root gets formed. The items consist of a SCALE encoded array containing only values, where the corresponding key of each value is the index of the item in the array, starting at 0. The keys are compact encoded integers as described in Definition B.13.
- `result`: a 32-bit pointer to the buffer containing the 256-bit trie root result.

D.7.3. `ext_trie_keccak_256_root`

Compute a 256-bit Keccak trie root formed from the iterated items.

D.7.3.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_root_version_1
   (param $data i64) (result i32)
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the iterated items from which the trie root gets formed. The items consist of a SCALE encoded array containing arbitrary key/value pairs.
- result: a 32-bit pointer to the buffer containing the 256-bit trie root.

D.7.4. `ext_trie_keccak_256_ordered_root`

Compute a 256-bit Keccak trie root formed from the enumerated items.

D.7.4.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_ordered_root_version_1
     (param $data i64) (result i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the enumerated items from which the trie root gets formed. The items consist of a SCALE encoded array containing only values, where the corresponding key of each value is the index of the item in the array, starting at 0. The keys are compact encoded integers as described in Definition B.13.
- `result`: a 32-bit pointer to the buffer containing the 256-bit trie root result.

D.8. MISCELLANEOUS

Interface that provides miscellaneous functions for communicating between the runtime and the node.

D.8.1. `ext_misc_chain_id`

Returns the current relay chain identifier.

D.8.1.1. Version 1 - Prototype

```
(func $ext_misc_chain_id_version_1 (result i64))
```

Arguments:

- `result`: the current relay chain identifier.

D.8.2. `ext_misc_print_num`

Print a number.

D.8.2.1. Version 1 - Prototype

```
(func $ext_misc_print_num_version_1 (param $value i64))
```

Arguments:

- `value`: the number to be printed.

D.8.3. `ext_misc_print_utf8`

Print a valid `UTF8` buffer.

D.8.3.1. Version 1 - Prototype

```
(func $ext_misc_print_utf8_version_1 (param $data i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the valid `UTF8` buffer to be printed.

D.8.4. `ext_misc_print_hex`

Print any buffer in hexadecimal representation.

D.8.4.1. Version 1 - Prototype

```
(func $ext_misc_print_hex_version_1 (param $data i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the buffer to be printed.

D.8.5. `ext_misc_runtime_version`

Extract the Runtime version of the given Wasm blob by calling `Core_version` as defined in Definition E.3.1.1. Returns the SCALE encoded runtime version or `None` as defined in Definition B.5 if the call fails. This function gets primarily used when upgrading Runtimes.

Warning: Calling this function is very expensive and should only be done very occasionally. For getting the runtime version, it requires instantiating the Wasm blob as described in Section 3.1.1 and calling a function in this blob.

D.8.5.1. Version 1 - Prototype

```
(func $ext_misc_runtime_version_version_1 (param $data i64) (result i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition D.3 indicating the Wasm blob.
- `result`: a pointer-size as defined in Definition D.3 indicating the SCALE encoded `Option` as defined in Definition B.5 containing the Runtime version of the given Wasm blob.

D.9. ALLOCATOR

The Polkadot Runtime does not include a memory allocator and relies on the Host API for all heap allocations. The beginning of this heap is marked by the `__heap_base` symbol exported by the Polkadot Runtime. No memory should be allocated below that address, to avoid clashes with the stack and data section. The same allocator made accessible by this Host API should be used for any other WASM memory allocations and deallocations outside the runtime e.g. when passing the SCALE-encoded parameters to Runtime API calls.

D.9.1. ext_allocator_malloc

Allocates the given number of bytes and returns the pointer to that memory location.

D.9.1.1. Version 1 - Prototype

```
(func $ext_allocator_malloc_version_1 (param $size i32) (result i32))
```

Arguments:

- `size`: the size of the buffer to be allocated.
- `result`: a 32-bit pointer to the allocated buffer.

D.9.2. ext_allocator_free

Free the given pointer.

D.9.2.1. Version 1 - Prototype

```
(func $ext_allocator_free_version_1 (param $ptr i32))
```

Arguments:

- `ptr`: a 32-bit pointer to the memory buffer to be freed.

D.10. LOGGING

Interface that provides functions for logging from within the runtime.

DEFINITION D.13. ***Log Level*** *is a varying data type as defined in Definition B.4 and implies the emergency of the log. Possible levels and it's identifiers are defined in the following table.*

Id	*Level*
0	*Error = 1*
1	*Warn = 2*
2	*Info = 3*
3	*Debug = 4*
4	*Trace = 5*

Table D.4. *Log Levels for the logging interface*

D.10.1. ext_logging_log

Request to print a log message on the host. Note that this will be only displayed if the host is enabled to display log messages with given level and target.

D.10.1.1. Version 1 - Prototype

```
(func $ext_logging_log_version_1
     (param $level i32) (param $target i64) (param $message i64))
```

Arguments:

- `level`: the log level as defined in Definition D.13.
- `target`: a pointer-size as defined in Definition D.3 indicating the string which contains the path, module or location from where the log was executed.
- `message`: a pointer-size as defined in Definition D.3 indicating the log message.

□

Appendix E
Polkadot Runtime API

E.1. General Information

The Polkadot Host assumes that at least the constants and functions described in this Chapter are implemented in the Runtime Wasm blob.

It should be noted that the API can change through the Runtime updates. Therefore, a host should check the API versions of each module returned in the `api` field by `Core_version` (Section E.3.1.1) after every Runtime upgrade and warn if an updated API is encountered and that this might require an update of the host.

E.1.1. JSON-RPC API for external services

Polkadot Host implementers are encouraged to implement an API in order for external, third-party services to interact with the node. The JSON-RPC Interface for Polkadot Nodes (PSP Number 006) is a Polkadot Standard Proposal for such an API and makes it easier to integrate the node with existing tools available in the Polkadot ecosystem, such as polkadot.js.org. The Runtime API has a few modules designed specifically for use in the official RPC API.

E.2. Runtime Constants

E.2.1. `__heap_base`

This constant indicates the beginning of the heap in memory. The space below is reserved for the stack and the data section. For more details please refer to Section D.9.

E.3. Runtime Functions

In this section, we describe all Runtime API functions alongside their arguments and the return values. The functions are organized into modules with each being versioned independently.

Definition E.1. *The **Runtime API Call Convention** describes that all functions receive and return SCALE-encoded data and as a result have the following prototype signature:*

```
(func $generic_runtime_entry
  (param $ptr i32) (parm $len i32) (result i64))
```

where `ptr` *points to the SCALE encoded tuple of the parameters passed to the function and* `len` *is the length of this data, while* `result` *is a pointer-size (Definition D.3) to the SCALE-encoded return data.*

See Section 3.1.2 for more information about the behaviour of the Wasm Runtime. Do note that any state changes created by calling any of the Runtime functions are not necessarily to be persisted after the call is ended. See Section 3.1.2.4 for more information.

E.3.1. Core Module (Version 3)

E.3.1.1. `Core_version`

Returns the version identifiers of the Runtime. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section E.1.1.

Arguments:

- None

Return:

- A data structure of the following format:

Name	Type	Description
`spec_name`	String	Runtime identifier
`impl_name`	String	the name of the implementation (e.g. C++)
`authoring_version`	UINT32	the version of the authorship interface
`spec_version`	UINT32	the version of the Runtime specification
`impl_version`	UINT32	the version of the Runtime implementation
`apis`	ApisVec (E.2)	List of supported APIs along with their version
`transaction_version`	UINT32	the version of the transaction format

Table E.1. Details of the version that the data type returns from the Runtime `version` function.

DEFINITION E.2. ***ApisVec*** *is a specialized type for the* `Core_version` *(E.3.1.1) function entry. It represents an array of tuples, where the first value of the tuple is an array of 8-bytes containing the Blake2b hash of the API name. The second value of the tuple is the version number of the corresponding API.*

$$\begin{aligned} \text{ApiVec} &:= (T_0, \ldots, T_n) \\ T &:= ((b_0, \ldots, b_7), \text{UINT32}) \end{aligned}$$

Requires `Core_intialize_block` to be called beforehand.

E.3.1.2. `Core_execute_block`

This function executes a full block and all its exctrinsics and updates the state accordingly. Additionally, some integrity checks are executed such as validating if the parent hash is correct and that the transaction root represents the transactions. Internally, this function performs an operation similar to the process described in Algorithm 6.7, by calling `Core_initialize_block`, `BlockBuilder_apply_extrinsics` and `BlockBuilder_finalize_block`.

This function should be called when a fully complete block is available that is not actively being built on, such as blocks received from other peers. State changes resulted from calling this function are usually meant to persist when the block is imported successfully.

Additionally, the seal digest in the block header, as described in Section 3.7, must be removed by the Polkadot host before submitting the block.

Arguments:

- A block represented as a tuple consisting of a block header, as described in Section 3.6, and the block body, as described in Section 3.9.

Return:

- None.

E.3.1.3. `Core_initialize_block`

Sets up the environment required for building a new block as described in Algorithm 6.7.

Arguments:

- The header of the new block as defined in 3.6. The values H_r, H_e and H_d are left empty.

Return:

- None.

E.3.2. Metadata Module (Version 1)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called first.

E.3.2.1. `Metadata_metadata`

Returns native Runtime metadata in an opaque form. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section E.1.1. and returns all the information necessary to build valid transactions.

Arguments:

- None.

Return:

- A byte array of varying size containing the metadata in an opaque form.

E.3.3. BlockBuilder Module (Version 4)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.3.1. `BlockBuilder_apply_extrinsic`

Apply the extrinsic outside of the block execution function. This does not attempt to validate anything regarding the block, but it builds a list of transaction hashes.

Arguments:

- A byte array of varying size containing the extrinsic.

Return:

- Returns the varying datatype `ApplyExtrinsicResult` as defined in Definition E.3. This structure lets the block-builder know whether an extrinsic should be included into the block or rejected.

Definition E.3. *`ApplyExtrinsicResult` is the varying data type `Result` as defined in Definition B.6. This structure can contain multiple nested structures, indicating either module dispatch outcomes or transaction invalidity errors.*

Id	*Description*	*Type*
0	*Outcome of dispatching the extrinsic.*	*DispatchOutcome (E.5)*
1	*Possible errors while checking the validity of a transaction.*	*TransactionValidityError (E.9)*

Table E.2. *Possible values of varying data type* ***ApplyExtrinsicResult****.*

Note E.4. As long as a `DispatchOutcome` (E.5) is returned, the extrinsic is always included in the block, even if the outcome is a dispatch error. Dispatch errors do not invalidate the block and all state changes are persisted.

DEFINITION E.5. ***DispatchOutcome*** *is the varying data type* `Result` *as defined in Definition B.6.*

Id	*Description*	*Type*
0	*Extrinsic is valid and was submitted successfully.*	*None*
1	*Possible errors while dispatching the extrinsic.*	*DispatchError (E.6)*

Table E.3. *Possible values of varying data type* ***DispatchOutcome****.*

DEFINITION E.6. ***DispatchError*** *is a varying data type as defined in Definition B.4. Indicates various reasons why a dispatch call failed.*

Id	*Description*	*Type*
0	*Some unknown error occurred.*	*SCALE encoded byte array containing a valid UTF-8 sequence.*
1	*Failed to lookup some data.*	*None*
2	*A bad origin.*	*None*
3	*A custom error in a module.*	*CustomModuleError (E.7)*

Table E.4. *Possible values of varying data type* ***DispatchError****.*

DEFINITION E.7. ***CustomModuleError*** *is a tuple appended after a possible error in* `DispatchError` *as defined in Defintion E.6.*

Name	*Description*	*Type*
Index	*Module index matching the metadata module index.*	*Unsigned 8-bit integer.*
Error	*Module specific error value.*	*Unsigned 8-bit integer*
Message	*Optional error message.*	*Varying data type* ***Option*** *(B.5). The optional value is a SCALE encoded byte array containing a valid UTF-8 sequence.*

Table E.5. *Possible values of varying data type* ***CustomModuleError****.*

Note E.8. Whenever `TransactionValidityError` (E.9) is returned, the contained error type will indicate whether an extrinsic should be outright rejected or requested for a later block. This behaviour is clarified further in Definition E.10 respectively Definition E.11.

DEFINITION E.9. ***TransactionValidityError*** *is a varying data type as defined in Definition B.4. It indicates possible errors that can occur while checking the validity of a transaction.*

Id	*Description*	*Type*
0	*Transaction is invalid.*	*InvalidTransaction (E.10)*
1	*Transaction validity can't be determined.*	*UnknownTransaction (E.11)*

Table E.6. *Possible values of varying data type* ***TransactionValidityError****.*

DEFINITION E.10. ***InvalidTransaction*** *is a varying data type as defined in Definition B.4 and specifies the invalidity of the transaction in more detail.*

Id	*Description*	*Type*	*Reject*
0	*Call of the transaction is not expected.*	*None*	*Yes*
1	*General error to do with the inability to pay some fees (e.g. account balance too low).*	*None*	*Yes*
2	*General error to do with the transaction not yet being valid (e.g. nonce too high).*	*None*	*No*
3	*General error to do with the transaction being outdated (e.g. nonce too low).*	*None*	*Yes*
4	*General error to do with the transactions' proof (e.g. signature)*	*None*	*Yes*
5	*The transaction birth block is ancient.*	*None*	*Yes*
6	*The transaction would exhaust the resources of the current block.*	*None*	*No*
7	*Some unknown error occured.*	*Unsigned 8-bit integer*	*Yes*
8	*An extrinsic with mandatory dispatch resulted in an error.*	*None*	*Yes*
9	*A transaction with a mandatory dispatch (only inherents are allowed to have mandatory dispatch).*	*None*	*Yes*

Table E.7. *Possible values of varying data type* ***InvalidTransaction****.*

Definition E.11. ***UnknownTransaction*** *is a varying data type as defined in Definition B.4 and specifies the unknown invalidity of the transaction in more detail.*

Id	*Description*	*Type*	*Reject*
0	*Could not lookup some information that is required to validate the transaction.*	*None*	*Yes*
1	*No validator found for the given unsigned transaction.*	*None*	*Yes*
2	*Any other custom unknown validity that is not covered by this type.*	*Unsigned 8-bit integer*	*Yes*

Table E.8. *Possible values of varying data type* ***UnknownTransaction****.*

E.3.3.2. `BlockBuilder_finalize_block`

Finalize the block - it is up to the caller to ensure that all header fields are valid except for the state root. State changes resulting from calling this function are usually meant to persist upon successful execution of the function and appending of the block to the chain.

Arguments:

- None.

Return:

- The header of the new block as defined in 3.6.

E.3.3.3. `BlockBuilder_inherent_extrinsics`

Generates the inherent extrinsics, which are explained in more detail in Section 3.2.3. This function takes a SCALE-encoded hash table as defined in Section B.8 and returns an array of extrinsics. The Polkadot Host must submit each of those to the `BlockBuilder_apply_extrinsic`, described in Section E.3.3.1. This procedure is outlined in Algorithm 6.7.

Arguments:

- A Inherents-Data structure as defined in 3.5.

Return:

- A byte array of varying size containing extrinisics. Each extrinsic is a byte array of varying size.

E.3.3.4. `BlockBuilder_check_inherents`

Checks whether the provided inherent is valid. This function can be used by the Polkadot Host when deemed appropriate, e.g. during the block-building process.

Arguments:

- A block represented as a tuple consisting of a block header as described in Section 3.6 and the block body as described in Section 3.9.
- A INHERENTS-DATA structure as defined in 3.5.

Return:

- A datastructure of the following format:

$$(o, f_e, e)$$

where

- o is a boolean indicating whether the check was successful.
- f_e is a boolean indicating whether a fatal error was encountered.
- e is a INHERENTS-DATA structure as defined in 3.5 containing any errors created by this Runtime function.

E.3.3.5. `BlockBuilder_random_seed`

Generates a random seed. [there is currently no requirement for having to call this function.]

Arguments:

- None.

Return:

- A 32-byte array containing the random seed.

E.3.4. TaggedTransactionQueue (Version 2)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.4.1. `TaggedTransactionQueue_validate_transaction`

This entry is invoked against extrinsics submitted through a transaction network message 4.8.3 or by an offchain worker through the `ext_offchain_submit_transaction` Host API (Section D.5.2). It indicates if the submitted blob represents a valid extrinsics, the order in which it should be applied and if it should be gossiped to other peers. Furthermore this function gets called internally when executing blocks with the `Core_execute_block` runtime function as described in Section E.3.1.2.

Arguments:

- The source of the transaction as defined in Definition E.12.
- A byte array that contains the transaction.

Definition E.12. ***TransactionSource*** *is an enum describing the source of a transaction and can have one of the following values:*

Id	*Name*	*Description*
0	*InBlock*	*Transaction is already included in a block.*
1	*Local*	*Transaction is coming from a local source, e.g. off-chain worker.*
2	*External*	*Transaction has been received externally, e.g. over the network.*

Table E.9. *The* `TransactionSource` *enum*

Return: This function returns a `Result` as defined in Definition B.6 which contains the type `ValidTransaction` as defined in Definition E.13 on success and the type `TransactionValidityError` as defined in Definition E.9 on failure.

Definition E.13. ***ValidTransaction*** *is a tuple that contains information concerning a valid transaction.*

Name	*Description*	*Type*
Priority	*Determines the ordering of two transactions that have all their dependencies (required tags) are satisfied.*	*Unsigned 64bit integer*
Requires	*List of tags specifying extrinsics which should be applied before the current exrinsics can be applied.*	*Array containing inner arrays*
Provides	*Informs Runtime of the extrinsics depending on the tags in the list that can be applied after current extrinsics are being applied. Describes the minimum number of blocks for the validity to be correct*	*Array containing inner arrays*
Longevity	*After this period, the transaction should be removed from the pool or revalidated.*	*Unsigned 64bit integer*
Propagate	*A flag indicating if the transaction should be gossiped to other peers.*	*Boolean*

Table E.10. *The tuple provided by* `TaggedTransactionQueue_transaction_validity` *in the case the transaction is judged to be valid.*

Note: If *Propagate* is set to `false` the transaction will still be considered for inclusion in blocks that are authored on the current node, but should not be gossiped to other peers.

Note: If this function gets called by the Polkadot Host in order to validate a transaction received from peers, the Polkadot Host disregards and rewinds state changes resulting in such a call.

E.3.5. OffchainWorkerApi Module (Version 2)

Does not require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.5.1. `OffchainWorkerApi_offchain_worker`

Starts an off-chain worker and generates extrinsics. [when is this called?]

Arguments:

- The block header as defined in 3.6.

Return:

- None.

E.3.6. ParachainHost Module (Version 1)

E.3.6.1. `ParachainHost_validators`

[future-reserved]

E.3.6.2. `ParachainHost_validator_groups`

[future-reserved]

E.3.6.3. `ParachainHost_availability_cores`

[future-reserved]

E.3.6.4. `ParachainHost_persisted_validation_data`

[future-reserved]

E.3.6.5. `ParachainHost_check_validation_outputs`

[future-reserved]

E.3.6.6. `ParachainHost_session_index_for_child`

[future-reserved]

E.3.6.7. `ParachainHost_session_info`

[future-reserved]

E.3.6.8. `ParachainHost_validation_code`

[future-reserved]

E.3.6.9. `ParachainHost_historical_validation_code`

[future-reserved]

E.3.6.10. `ParachainHost_candidate_pending_availability`

[future-reserved]

E.3.6.11. `ParachainHost_candidate_events`

[future-reserved]

E.3.6.12. `ParachainHost_dmq_contents`

[future-reserved]

E.3.6.13. `ParachainHost_inbound_hrmp_channel_contents`

[future-reserved]

E.3.7. GrandpaApi Module (Version 2)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.7.1. `GrandpaApi_grandpa_authorities`

This entry fetches the list of GRANDPA authorities according to the genesis block and is used to initialize an authority list at genesis, defined in Definition 6.1. Any future authority changes get tracked via Runtime-to-consensus engine messages, as described in Section 6.1.2.

Arguments:

- None.

Return:

- An authority list as defined in Definition 6.1.

E.3.7.2. `GrandpaApi_submit_report_equivocation_unsigned_extrinsic`

A GRANDPA equivocation occurs when a validator votes for multiple blocks during one voting subround, as described further in Section 6.31. The Polkadot Host is expected to identify equivocators and report those to the Runtime by calling this function.

Arguments:

- The equivocation proof of the following format:

$$
\begin{aligned}
G_{\text{Ep}} &= (\text{id}_{\mathbb{V}}, e, r, A_{\text{id}}, B_h^1, B_n^1 A_{\text{sig}}^1, B_h^2, B_n^2, A_{\text{sig}}^2) \\
e &= \begin{cases} 0 & \textit{Equivocation at prevote stage.} \\ 1 & \textit{Equivocation at precommit stage} \end{cases}
\end{aligned}
$$

  where

  - $\text{id}_{\mathbb{V}}$ is the authority set as defined in Section 6.24.
  - e indicates the stage at which the equivocation occurred.
  - r is the round number the equivocation occurred.
  - A_{id} is the public key of the equivocator.
  - B_h^1 is the block hash of the first block the equivocator voted for.
  - B_n^1 is the block number of the first block the equivocator voted for.
  - A_{sig}^1 is the equivocators signature of the first vote.
  - B_h^2 is the block hash of the second block the equivocator voted for.
  - B_n^2 is the block number of the second block the equivocator voted for.
  - A_{sig}^2 is the equivocators signature of the second vote.
- A proof of the key owner in an opaque form as described in Section E.3.7.3.

Return:

- A SCALE encoded `Option` as defined in Definition B.5 containing an empty value on success.

E.3.7.3. `GrandpaApi_generate_key_ownership_proof`

Generates proof of the membership of a key owner in the specified block state. The returned value is used to report equivocations as described in Section E.3.7.2.

Arguments:

- The authority set Id as defined in Definition 6.24.
- The 256-bit public key of the authority.

Return:

- A SCALE encoded `Option` as defined in Definition B.5 containing the proof in an opaque form.

E.3.8. BabeApi Module (Version 2)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.8.1. `BabeApi_configuration`

This entry is called to obtain the current configuration of the BABE consensus protocol.

Arguments:

- None.

Return:

- A tuple containing configuration data used by the Babe consensus engine.

Name	Description	Type
SlotDuration	The slot duration in milliseconds. Currently, only the value provided by this type at genesis will be used. Dynamic slot duration may be supported in the future.	Unsigned 64bit integer
EpochLength	The duration of epochs in slots.	Unsigned 64bit integer
Constant	A constant value that is used in the threshold calculation formula as defined in definition 6.11.	Tuple containing two unsigned 64bit integers
Genesis Authorities	The authority list for the genesis epoch as defined in Definition 6.1.	Array of tuples containing a 256-bit byte array and a unsigned 64bit integer
Randomness	The randomness for the genesis epoch	32-byte array
SecondarySlot	Whether this chain should run with secondary slots and wether they are assigned in a round-robin manner or via a second VRF.	8bit enum

Table E.11. The tuple provided by **BabeApi_configuration**.

E.3.8.2. `BabeApi_current_epoch_start`

Finds the start slot of the current epoch.

Arguments:

- None.

Return:

- A unsigned 64-bit integer indicating the slot number.

E.3.8.3. `BabeApi_current_epoch`

Produces information about the current epoch.

Arguments:

- None.

Return:

- A datastructure of the following format:

$$(e_i, s_s, d, A, r)$$

where:

- e_i is a unsigned 64-bit integer representing the epoch index.
- s_s is a unsigned 64-bit integer representing the starting slot of the epoch.

- d is a unsigned 64-bit integer representing the duration of the epoch.
- A is an authority list as defined in Definition 6.1.
- r is an 256-bit array containing the randomness for the epoch as defined in Definition 6.22.

E.3.8.4. `BabeApi_next_epoch`

Produces information about the next epoch.

Arguments:

- None.

Return:

- Returns the same datastructure as described in Section E.3.8.3.

E.3.8.5. `BabeApi_generate_key_ownership_proof`

Generates a proof of the membership of a key owner in the specified block state. The returned value is used to report equivocations as described in Section E.3.8.6.

Arguments:

- The unsigned 64-bit integer indicating the slot number.
- The 256-bit public key of the authority.

Return:

- A SCALE encoded `Option` as defined in Definition B.5 containing the proof in an opaque form.

E.3.8.6. `BabeApi_submit_report_equivocation_unsigned_extrinsic`

A BABE equivocation occurs when a validator produces more than one block at the same slot. The proof of equivocation are the given distinct headers that were signed by the validator and which include the slot number. The Polkadot Host is expected to identify equivocators and report those to the Runtime using this function.

Note E.14. If there are more than two blocks which cause an equivocation, the equivocation only needs to be reported once i.e. no additional equivocations must be reported for the same slot.

Arguments:

- The equivocation proof of the following format:

$$B_{\text{Ep}} = (A_{\text{id}}, s, h_1, h_2)$$

  where

  - A_{id} is the public key of the equivocator.
  - s is the slot as described in Section 6.2 at which the equivocation occurred.
  - h_1 is the block header of the first block produced by the equivocator.
  - h_2 is the block header of the second block produced by the equivocator.

  Unlike during block execution, the Seal in both block headers is not removed before submission. The block headers are submitted in its full form.
- An proof of the key owner in an opaque form as described in Section E.3.8.5.

Return:

- A SCALE encoded `Option` as defined in Definition B.5 containing an empty value on success.

E.3.9. AuthorityDiscoveryApi Module (Version 1)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.9.1. `AuthorityDiscoveryApi_authorities`

A function which helps to discover authorities.

Arguments:

- None.

Return:

- A byte array of varying size containing 256-bit pulic keys of the authorities.

E.3.10. SessionKeys Module (Version 1)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.10.1. `SessionKeys_generate_session_keys`

Generates a set of session keys with an optional seed. The keys should be stored within the keystore exposed by the Host Api. The seed needs to be valid and UTF-8 encoded.

Arguments:

- A SCALE-encoded `Option` as defined in Definition B.5 containing an array of varying sizes indicating the seed.

Return:

- A byte array of varying size containg the encoded session keys.

E.3.10.2. `SessionKeys_decode_session_keys`

Decodes the given public session keys. Returns a list of raw public keys including their key type.

Arguments:

- An array of varying size containing the encoded public session keys.

Return:

- An array of varying size containing tuple pairs of the following format:

$$(k, k_{\text{id}})$$

where k is an array of varying sizes containg the raw public key and k_{id} is a 4-byte array indicating the key type.

E.3.11. AccountNonceApi Module (Version 1)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.11.1. `AccountNonceApi_account_nonce`

Get the current nonce of an account. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section E.1.1.

Arguments:

- The 256-bit public key of the account.

Return:

- A 32-bit unsigned integer indicating the nonce of the account.

E.3.12. TransactionPaymentApi Module (Version 1)

All calls in this module require `Core_intialize_block` (Section E.3.1.3) to be called beforehand.

E.3.12.1. `TransactionPaymentApi_query_info`

Returns information of a given extrinsic. This function is not aware of the internals of an extrinsic, but only interprets the extrinsic as some encoded value and accounts for its weight and length, the Runtime's extrinsic base weight and the current fee multiplier.

This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section E.1.1.

Arguments:

- A byte array of varying sizes containing the extrinsic.
- The length of the extrinsic. [why is this needed?]

Return:

- A data structure of the following format:

$$(w, c, f)$$

where:

- w is the weight of the extrinsic.
- c is the "class" of the extrinsic, where class is a varying data type defined as:

$$c = \begin{cases} 0 & \text{Normal extrinsic} \\ 1 & \text{Operational extrinsic} \\ 2 & \text{Mandatory extrinsic, which is always included} \end{cases}$$

- f is the inclusion fee of the extrinsic. This does not include a tip or anything else that depends on the signature.

E.3.12.2. `TransactionPaymentApi_query_fee_details`

Query the detailed fee of a given extrinsic. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section E.1.1.

Arguments:

- A byte array of varying sizes containing the extrinsic.
- The length of the extrinsic.

Return:

- A data structure of the following format:

$$(f, t)$$

where

- f is a SCALE encoded `Option` as defined in Definition B.5 containing the following datastructure:

$$(f_b, f_l, f_a)$$

where

- f_b is the minimum required fee for an extrinsic.
- f_l is the length fee, the amount paid for the encoded length (in bytes) of the extrinsic.
- f_a is the "adjusted weight fee", which is a multiplication of the fee multiplier and the weight fee. The fee multiplier varies depending on the usage of the network.

- t is the tip for the block author.

□

Glossary

P_n	A path graph or a path of n nodes. . . . 7
$(b_0, b_1, ..., b_{n-1})$	A sequence of bytes or byte array of length n . . . 8, 83
$\mathbb{B}_n$	A set of all byte arrays of length n . . . 8
$I = (B_n \dots B_0)_{256}$	A non-negative interger in base 256 . . . 8
$B = (b_0, b_1, \dots, b_n)$	The little-endian representation of a non-negative interger $I = (B_n \dots B_0)_{256}$ such that $b_i := B_i$ . . . 8
Enc_{LE}	The little-endian encoding function. . . . 8
C	A blockchain defined as a directed path graph. . . . 8
Block	A node of the directed path graph (blockchain) C . . . 8
Genesis Block	The unique sink of blockchain C . . . 8
Head	The source of blockchain C . . . 8
P(B)	The parent of block B . . . 8
UNIX time	The number of milliseconds that have elapsed since the Unix epoch as a 64-bit integer . 8
BT	The block tree of a blockchain . . . 9
PBT, Pruned BT	Pruned Block Tree. . . . 9
Pruning	The transformation of BT into PBT . . . 9
G	The genesis block, the root of the block tree BT . . . 9
CHAIN(B)	The path graph from G to B in (P)BT. . . . 9
$\textsc{Head}(C)$	The head of chain C. . . . 9
$\lvert C \rvert$	The length of chain C as a path graph . . . 9
$\textsc{SubChain}(B', B)$	The subgraph of $\textsc{Chain}(B)$ path graph containing both B and B'. . . . 9
$\mathbb{C}_{B'}((P)\text{BT})$	The set of all subchains of (P)BT rooted at block B'. . . . 9
$\mathbb{C}$, $\mathbb{C}$((P)BT)	$\mathbb{C}_G((P)\text{BT})$ i.e. the set of all chains of (P)BT rooted at genesis block . . . 9
$\textsc{Longest-Chain}$(BT)	The longest sub path graph of BT *i.e.* C: $\lvert C \rvert = \max\{\lvert C_i \rvert \lvert C_i \in \mathbb{C}\}$ . . . 9
$\textsc{Longest-Path}$(BT)	The longest sub path graph of (P)BT with earliest block arrival time . . . 9
$\textsc{Deepest-Leaf}$(BT)	$\textsc{Head}(\textsc{Longest-Path}(\text{BT}))$ i.e. the head of $\textsc{Longest-Path}$(BT) . . . 9
$B > B'$	B is a descendant of B' in the block tree . . . 9
StoredValue(k)	The function to retrieve the value stored under a specific key in the state storage. . . . 11
State Trie, Trie	*The Merkle radix-16 Tree which stores hashes of storage enteries.* . . . 11, 12
KeyEncode(k)	The function to encode keys for labeling branaches of the Trie. . . . 11
$\mathcal{N}$	The set of all nodes in the Polkadot state trie . . . 12
N	An individual node in the Trie . . . 12
$\mathcal{N}_{\text{b}}$	A branch node of the Trie which has at least one and at most 16 children . . . 12
$\mathcal{N}_l$	A childless leaf node of the Trie . . . 12
pk_N^{Agr}	The aggregated prefix key of node N . . . 12
pk_N	The (suffix) partial key of node N . . . 12
Index_N	A function returning an integer in range of $\{0,\dots,15\}$ represeting the index of a child node of node N among the children of N . . . 12
v_N	Node value containing the header of node N, its partial key and the digest of its childern values . . . 13
Head_N	The node header of Trie node N storing information about the node's type and kay . . 13
$H(N)$	The Merkle value of node N. . . . 14
ChildrenBitmap	The binary function indicating which child of a given node is present in the Trie. . . . 14
sv_N	The subvalue of a Trie node N. . . . 15
Child storage	A sub storage of the state storage which has the same structure although being stored seperately . . . 15
Child Trie	State trie of a child storage . . . 15
Transaction Message	. . . 19, 32
Transaction Pool	. . . 19
Transaction Queue	. . . 19
H_{p}	The 32-byte Blake2b hash of the header of the parent of the block. . . . 21
H_i, $H_i(B)$	Block number, the incremental interger index of the current block in the chain. . . 21, 24
H_r	The hash of the root of the Merkle trie of the state storage at a given block . . . 21
H_e	An auxuliray field in block header used by Runtime to validate the integrity of the extrinsics composing the block body . . . 21
H_d, $H_d(B)$	A block header used to store any chain-specific auxiliary data. . . . 22, 22
$H_h(B)$	The hash of the header of block B . . . 22
Body(B)	The body of block B consisting of a set of extrinsics . . . 22
$M_v^{r,\text{stage}}$	Vote message broadcasted by the voter v as part of the finality protocol . . . 33

$M_v^{r,\mathrm{Fin}}(B)$	The commit message broadcasted by voter v indicating that they have finalized bock B in round r . . . 34
v	GRANDPA voter node which casts vote in the finality protocol . . . 47
k_v^{pr}	The private key of voter v . . . 47
v_{id}	The public key of voter v . . . 47
$\mathbb{V}_B, \mathbb{V}$	The set of all GRANDPA voters for at block B . . . 47, 47
GS	GRANDPA protocol state consisting of the set of voters, number of times voters set has changed and the current round number. . . . 47
r	The voting round counter in the finality protocol . . . 47
$V(B)$	A GRANDPA vote casted in favor of block B . . . 48
$V_v^{r,\mathrm{pv}}$	A GRANDPA vote casted by voter v during the pre-vote stage of round r . . . 48
$V_v^{r,\mathrm{pc}}$	A GRANDPA vote casted by voter v during the pre-commit stage of round r . . . 48
$J^{r,\mathrm{stage}}(B)$	The justification for pre-commiting or comming to block B in round r of finality protocol . . . 48
$\mathrm{Sign}_{v_i}^{r,\mathrm{stage}}(B)$	The signature of voter v on their voteto block B, broadcasted during the specified stage of finality round r . . . 48
$\mathcal{E}^{r,\mathrm{stage}}$	The set of all equivocator voters in sub-round "stage" of round r . . . 49
$\mathcal{E}^{r,\mathrm{stage}}_{\mathrm{obs}(v)}$	The set of all equivocator voters in sub-round "stage" of round r observed by voter v . 49
$\mathrm{VD}^{r,\mathrm{stage}}_{\mathrm{obs}(v)}(B)$	The set of observed direct votes for block B in round r . . . 49
$V^{r,\mathrm{stage}}_{\mathrm{obs}(v)}$	The set of total votes observed by voter v in sub-round "stage" of round r . . . 49
$V^{r,\mathrm{stage}}_{\mathrm{obs}(v)}(B)$	The set of all observed votes by v in the sub-round "stage" of round r (directly or indirectly) for block B . . . 49
$B_v^{r,\mathrm{pv}}$	The currently pre-voted block in round r. The GRANDPA GHOST of round r . . . 49
Account key, $(\mathrm{sk}^a, \mathrm{pk}^a)$	A key pair of types accepted by the Polkadot protocol which can be used to sign transactions . . . 81
$\mathrm{Enc}_{\mathrm{SC}}(A)$	SCALE encoding of value A . . . 83, 83, 84
$T := (A_1, \ldots, A_n)$	A tuple of values A_i's each of different type . . . 83
Varying Data Types $\mathcal{T} = \{T_1, \ldots, T_n\}$	A data type representing any of varying types $T_1, \ldots, T_n$. . . . 83
$S := A_1, \ldots, A_n$	Sequence of values A_i of the same type . . . 84
$\mathrm{Enc}^{\mathrm{Len}}_{\mathrm{SC}}(n)$	SCALE length encoding aka. compact encoding of non-negative interger n of arbitrary size. . . . 84
$\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK})$	Hex encoding . . . 85

Bibliography

[Bur19] Jeff Burdges. Schnorr VRFs and signatures on the Ristretto group. Technical Report, 2019.

[DGKR18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[Fou21] Web3.0 Technologies Foundation. Polkadot protocol Specification. Technical Report, https://github.com/w3f/polkadot-spec, 2021.

[Gro19] W3F Research Group. Blind Assignment for Blockchain Extension. Technical ⟨keepcase | Specification⟩, Web 3.0 Foundation, http://research.web3.foundation/en/latest/polkadot/BABE/Babe/, 2019.

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). In *Internet Research Task Force, Crypto Forum Research Group, RFC*, volume 8032. 2017.

[SA15] Markku Juhani Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC). ⟨keepcase|RFC⟩ 7693, -, https://tools.ietf.org/html/rfc7693, 2015.

[Ste19] Alistair Stewart. GRANDPA: A Byzantine Finality Gadget. 2019.

[Tec20] Parity Tech. Polkadot Genisis State. Technical Report, https://github.com/paritytech/polkadot/tree/master/node/service/res/, 2020.