The Polkadot Host

Protocol Specification

November 25, 2020

Table of contents

1. Background 13

1.1. Introduction 13
1.2. Definitions and Conventions 13
1.2.1. Block Tree 14

2. State Specification 17

2.1. State Storage and Storage Trie 17
2.1.1. Accessing System Storage 17
2.1.2. The General Tree Structure 17
2.1.3. Trie Structure 18
2.1.4. Merkle Proof 20
2.2. Child Storage 21
2.2.1. Child Tries 21

3. State Transition 23

3.1. Interactions with Runtime 23
3.1.1. Loading the Runtime Code 23
3.1.2. Code Executor 24
3.1.2.1. Access to Runtime API 24
3.1.2.2. Sending Arguments to Runtime 24
3.1.2.3. The Return Value from a Runtime Entry 24
3.1.2.4. Handling Runtimes update to the State 25
3.2. Extrinsics 25
3.2.1. Preliminaries 25
3.2.2. Transactions 25
3.2.2.1. Transaction Submission 25
3.2.3. Transaction Queue 25
3.2.3.1. Inherents 27
3.3. State Replication 27
3.3.1. Block Format 27
3.3.1.1. Block Header 27
3.3.1.2. Justified Block Header 28
3.3.1.3. Block Body 28
3.3.2. Importing and Validating Block 29
3.3.3. Managaing Multiple Variants of State 29
3.3.4. Changes Trie 30
3.3.4.1. Key to extrinsics pairs 31
3.3.4.2. Key to block pairs 31
3.3.4.3. Key to Child Changes Trie pairs 32

4. Networking 33

4.1. Introduction 33
4.1.1. External Documentation 33
4.1.2. Discovery mechanism 33
4.1.3. Connection establishment 33
4.1.4. Substreams 34
4.2. Network Messages 34
4.2.1. API Package 34
4.2.1.1. BlockRequest 34

4.2.1.2. BlockResponse . . . 35
4.2.2. Light Package . . . 35
4.2.2.1. Request . . . 35
4.2.2.2. Response . . . 36
4.2.2.3. RemoteCallRequest . . . 36
4.2.2.4. RemoteCallResponse . . . 36
4.2.2.5. RemoteReadRequest . . . 36
4.2.2.6. RemoteReadResponse . . . 36
4.2.2.7. RemoteReadChildRequest . . . 37
4.2.2.8. RemoteHeaderRequest . . . 37
4.2.2.9. RemoteHeaderResponse . . . 37
4.2.2.10. RemoteChangesRequest . . . 37
4.2.2.11. RemoteChangesResponse . . . 38
4.2.3. Finality Package . . . 38
4.2.3.1. FinalityProofRequest . . . 38
4.2.3.2. FinalityProofResponse . . . 38
4.3. To be migrated . . . 39
4.3.1. Node Identities and Addresses . . . 39
4.3.2. Discovery Mechanisms . . . 39
4.3.3. Transport Protocol . . . 40
4.3.3.1. Encryption . . . 40
4.3.3.2. Multiplexing . . . 40
4.3.4. Substreams . . . 40
4.3.4.1. Periodic Ephemeral Substreams . . . 40
4.3.4.2. Polkadot Communication Substream . . . 40

5. Bootstrapping . . . 43

6. Consensus . . . 45

6.1. Common Consensus Structures . . . 45
6.1.1. Consensus Authority Set . . . 45
6.1.2. Runtime-to-Consensus Engine Message . . . 45
6.2. Block Production . . . 47
6.2.1. Preliminaries . . . 47
6.2.2. Block Production Lottery . . . 48
6.2.3. Slot Number Calculation . . . 48
6.2.4. Block Production . . . 50
6.2.5. Epoch Randomness . . . 51
6.2.6. Verifying Authorship Right . . . 51
6.2.7. Block Building Process . . . 52
6.3. Finality . . . 53
6.3.1. Preliminaries . . . 53
6.3.2. GRANDPA Messages Specification . . . 55
6.3.2.1. Vote Messages . . . 55
6.3.2.2. Finalizing Message . . . 56
6.3.2.3. Catch-up Messages . . . 56
6.3.3. Initiating the GRANDPA State . . . 57
6.3.3.1. Voter Set Changes . . . 57
6.3.4. Voting Process in Round r . . . 57
6.4. Block Finalization . . . 60
6.4.1. Catching up . . . 60
6.4.1.1. Sending catch-up requests . . . 60
6.4.1.2. Processing catch-up requests . . . 60
6.4.1.3. Processing catch-up responses . . . 61

7. Availability & Validity . . . 63

7.1. Introduction . . . 63
7.2. Preliminaries . . . 63
7.2.1. Extra Validation Data . . . 64
7.3. Overal process . . . 65
7.4. Candidate Selection . . . 67
7.5. Candidate Backing . . . 67
7.5.1. Inclusion of candidate receipt on the relay chain . . . 70
7.6. PoV Distribution . . . 70
7.6.1. Primary Validation Disagreement . . . 70
7.7. Availability . . . 70
7.8. Distribution of Chunks . . . 72
7.9. Announcing Availability . . . 72
7.9.1. Processing on-chain availability data . . . 73
7.10. Publishing Attestations . . . 73
7.11. Secondary Approval checking . . . 73
7.11.1. Approval Checker Assignment . . . 73
7.11.2. VRF computation . . . 74
7.11.3. One-Shot Approval Checker Assignment . . . 74
7.11.4. Extra Approval Checker Assigment . . . 74
7.11.5. Additional Checking in Case of Equivocation . . . 75
7.12. The Approval Check . . . 75
7.12.0.1. Retrieval . . . 75
7.12.0.2. Reconstruction . . . 75
7.12.1. Verification . . . 76
7.12.2. Process validity and invalidity messages . . . 76
7.12.3. Invalidity Escalation . . . 76

8. Message Passing . . . 79

8.1. Overview . . . 79
8.2. Message Queue Chain (MQC) . . . 81
8.3. HRMP . . . 83
8.3.1. Channels . . . 83
8.3.2. Opening Channels . . . 83
8.3.2.1. Workflow . . . 84
8.3.3. Accepting Channels . . . 84
8.3.3.1. Workflow . . . 84
8.3.4. Closing Channels . . . 85
8.3.5. Workflow . . . 85
8.3.6. Sending messages . . . 85
8.3.7. Receiving Messages . . . 86
8.4. XCMP . . . 86
8.4.1. CST: Channel State Table . . . 88
8.4.2. Message content . . . 88
8.4.3. Watermark . . . 88
8.5. SPREE . . . 89

Appendix A. Cryptographic Algorithms . . . 91

A.1. Hash Functions . . . 91
A.2. BLAKE2 . . . 91
A.3. Randomness . . . 91
A.4. VRF . . . 91
A.5. Cryptographic Keys . . . 91
A.5.1. Holding and staking funds . . . 92
A.5.2. Creating a Controller key . . . 92
A.5.3. Designating a proxy for voting . . . 92
A.5.4. Controller settings . . . 92

A.5.5. Certifying keys 92

Appendix B. Auxiliary Encodings 93

B.1. SCALE Codec 93
B.1.1. Length and Compact Encoding 94
B.2. Hex Encoding 95

Appendix C. Genesis State Specification 97

Appendix D. Network Messages 99

D.1. Detailed Message Structure 99
D.1.1. Status Message 99
D.1.2. Block Request Message 100
D.1.3. Block Response Message 100
D.1.4. Block Announce Message 101
D.1.5. Transactions 101
D.1.6. Consensus Message 101
D.1.7. Neighbor Packet 101

Appendix E. Polkadot Host API 103

E.1. Storage 103
E.1.1. `ext_storage_set` 103
E.1.1.1. Version 1 - Prototype 103
E.1.2. `ext_storage_get` 103
E.1.2.1. Version 1 - Prototype 103
E.1.3. `ext_storage_read` 104
E.1.3.1. Version 1 - Prototype 104
E.1.4. `ext_storage_clear` 104
E.1.4.1. Version 1 - Prototype 104
E.1.5. `ext_storage_exists` 104
E.1.5.1. Version 1 - Prototype 104
E.1.6. `ext_storage_clear_prefix` 104
E.1.6.1. Version 1 - Prototype 104
E.1.7. `ext_storage_append` 105
E.1.7.1. Version 1 - Prototype 105
E.1.8. `ext_storage_root` 105
E.1.8.1. Version 1 - Prototype 105
E.1.9. `ext_storage_changes_root` 105
E.1.9.1. Version 1 - Prototype 105
E.1.10. `ext_storage_next_key` 105
E.1.10.1. Version 1 - Prototype 105
E.1.11. `ext_storage_start_transaction` 106
E.1.11.1. Version 1 - Prototype 106
E.1.12. `ext_storage_rollback_transaction` 106
E.1.12.1. Version 1 - Prototype 106
E.1.13. `ext_storage_commit_transaction` 106
E.1.13.1. Version 1 - Prototype 106
E.2. Child Storage 106
E.2.1. `ext_default_child_storage_set` 107
E.2.1.1. Version 1 - Prototype 107
E.2.2. `ext_default_child_storage_get` 107
E.2.2.1. Version 1 - Prototype 107
E.2.3. `ext_default_child_storage_read` 107
E.2.3.1. Version 1 - Prototype 107
E.2.4. `ext_default_child_storage_clear` 107
E.2.4.1. Version 1 - Prototype 108

E.2.5. `ext_default_child_storage_storage_kill` . . . 108
E.2.5.1. Version 1 - Prototype . . . 108
E.2.6. `ext_default_child_storage_exists` . . . 108
E.2.6.1. Version 1 - Prototype . . . 108
E.2.7. `ext_default_child_storage_clear_prefix` . . . 108
E.2.7.1. Version 1 - Prototype . . . 108
E.2.8. `ext_default_child_storage_root` . . . 108
E.2.8.1. Version 1 - Prototype . . . 108
E.2.9. `ext_default_child_storage_next_key` . . . 109
E.2.9.1. Version 1 - Prototype . . . 109
E.3. Crypto . . . 109
E.3.1. `ext_crypto_ed25519_public_keys` . . . 109
E.3.1.1. Version 1 - Prototype . . . 110
E.3.2. `ext_crypto_ed25519_generate` . . . 110
E.3.2.1. Version 1 - Prototype . . . 110
E.3.3. `ext_crypto_ed25519_sign` . . . 110
E.3.3.1. Version 1 - Prototype . . . 110
E.3.4. `ext_crypto_ed25519_verify` . . . 110
E.3.4.1. Version 1 - Prototype . . . 111
E.3.5. `ext_crypto_sr25519_public_keys` . . . 111
E.3.5.1. Version 1 - Prototype . . . 111
E.3.6. `ext_crypto_sr25519_generate` . . . 111
E.3.6.1. Version 1 - Prototype . . . 111
E.3.7. `ext_crypto_sr25519_sign` . . . 111
E.3.7.1. Version 1 - Prototype . . . 111
E.3.8. `ext_crypto_sr25519_verify` . . . 112
E.3.8.1. Version 2 - Prototype . . . 112
E.3.8.2. Version 1 - Prototype . . . 112
E.3.9. `ext_crypto_ecdsa_public_keys` . . . 112
E.3.9.1. Version 1 - Prototype . . . 112
E.3.10. `ext_crypto_ecdsa_generate` . . . 113
E.3.10.1. Version 1 - Prototype . . . 113
E.3.11. `ext_crypto_ecdsa_sign` . . . 113
E.3.11.1. Version 1 - Prototype . . . 113
E.3.12. `ext_crypto_ecdsa_verify` . . . 113
E.3.12.1. Version 1 - Prototype . . . 113
E.3.13. `ext_crypto_secp256k1_ecdsa_recover` . . . 114
E.3.13.1. Version 1 - Prototype . . . 114
E.3.14. `ext_crypto_secp256k1_ecdsa_recover_compressed` . . . 114
E.3.14.1. Version 1 - Prototype . . . 114
E.3.15. `ext_crypto_start_batch_verify` . . . 114
E.3.15.1. Version 1 - Prototype . . . 114
E.3.16. `ext_crypto_finish_batch_verify` . . . 115
E.3.16.1. Version 1 - Prototype . . . 115
E.4. Hashing . . . 115
E.4.1. `ext_hashing_keccak_256` . . . 115
E.4.1.1. Version 1 - Prototype . . . 115
E.4.2. `ext_hashing_sha2_256` . . . 115
E.4.2.1. Version 1 - Prototype . . . 115
E.4.3. `ext_hashing_blake2_128` . . . 115
E.4.3.1. Version 1 - Prototype . . . 115
E.4.4. `ext_hashing_blake2_256` . . . 116
E.4.4.1. Version 1 - Prototype . . . 116
E.4.5. `ext_hashing_twox_64` . . . 116
E.4.5.1. Version 1 - Prototype . . . 116
E.4.6. `ext_hashing_twox_128` . . . 116

E.4.6.1. Version 1 - Prototype . . . 116
E.4.7. `ext_hashing_twox_256` . . . 116
E.4.7.1. Version 1 - Prototype . . . 116
E.5. Offchain . . . 117
E.5.1. `ext_offchain_is_validator` . . . 117
E.5.1.1. Version 1 - Prototype . . . 117
E.5.2. `ext_offchain_submit_transaction` . . . 118
E.5.2.1. Version 1 - Prototype . . . 118
E.5.3. `ext_offchain_network_state` . . . 118
E.5.3.1. Version 1 - Prototype . . . 118
E.5.4. `ext_offchain_timestamp` . . . 118
E.5.4.1. Version 1 - Prototype . . . 118
E.5.5. `ext_offchain_sleep_until` . . . 118
E.5.5.1. Version 1 - Prototype . . . 118
E.5.6. `ext_offchain_random_seed` . . . 118
E.5.6.1. Version 1 - Prototype . . . 119
E.5.7. `ext_offchain_local_storage_set` . . . 119
E.5.7.1. Version 1 - Prototype . . . 119
E.5.8. `ext_offchain_local_storage_compare_and_set` . . . 119
E.5.8.1. Version 1 - Prototype . . . 119
E.5.9. `ext_offchain_local_storage_get` . . . 119
E.5.9.1. Version 1 - Prototype . . . 119
E.5.10. `ext_offchain_http_request_start` . . . 120
E.5.10.1. Version 1 - Prototype . . . 120
E.5.11. `ext_offchain_http_request_add_header` . . . 120
E.5.11.1. Version 1 - Prototype . . . 120
E.5.12. `ext_offchain_http_request_write_body` . . . 120
E.5.12.1. Version 1 - Prototype . . . 120
E.5.13. `ext_offchain_http_response_wait` . . . 121
E.5.13.1. Version 1- Prototype . . . 121
E.5.14. `ext_offchain_http_response_headers` . . . 121
E.5.14.1. Version 1 - Prototype . . . 121
E.5.15. `ext_offchain_http_response_read_body` . . . 121
E.5.15.1. Version 1 - Prototype . . . 122
E.6. Trie . . . 122
E.6.1. `ext_trie_blake2_256_root` . . . 122
E.6.1.1. Version 1 - Prototype . . . 122
E.6.2. `ext_trie_blake2_256_ordered_root` . . . 122
E.6.2.1. Version 1 - Prototype . . . 122
E.6.3. `ext_trie_keccak_256_root` . . . 122
E.6.3.1. Version 1 - Prototype . . . 123
E.6.4. `ext_trie_keccak_256_ordered_root` . . . 123
E.6.4.1. Version 1 - Prototype . . . 123
E.7. Miscellaneous . . . 123
E.7.1. `ext_misc_chain_id` . . . 123
E.7.1.1. Version 1 - Prototype . . . 123
E.7.2. `ext_misc_print_num` . . . 123
E.7.2.1. Version 1 - Prototype . . . 123
E.7.3. `ext_misc_print_utf8` . . . 123
E.7.3.1. Version 1 - Prototype . . . 124
E.7.4. `ext_misc_print_hex` . . . 124
E.7.4.1. Version 1 - Prototype . . . 124
E.7.5. `ext_misc_runtime_version` . . . 124
E.7.5.1. Version 1 - Prototype . . . 124
E.8. Allocator . . . 124
E.8.1. `ext_allocator_malloc` . . . 124

E.8.1.1. Version 1 - Prototype . . . 124
E.8.2. ext_allocator_free . . . 124
E.8.2.1. Version 1 - Prototype . . . 125
E.9. Logging . . . 125
E.9.1. ext_logging_log . . . 125
E.9.1.1. Version 1 - Prototype . . . 125

Appendix F. Legacy Polkadot Host API . . . 127

F.1. Storage . . . 127
F.1.1. ext_set_storage . . . 127
F.1.2. ext_storage_root . . . 127
F.1.3. ext_blake2_256_enumerated_trie_root . . . 127
F.1.4. ext_clear_prefix . . . 128
F.1.5. ext_clear_storage . . . 128
F.1.6. ext_exists_storage . . . 128
F.1.7. ext_get_allocated_storage . . . 129
F.1.8. ext_get_storage_into . . . 129
F.1.9. ext_set_child_storage . . . 129
F.1.10. ext_clear_child_storage . . . 130
F.1.11. ext_exists_child_storage . . . 130
F.1.12. ext_get_allocated_child_storage . . . 131
F.1.13. ext_get_child_storage_into . . . 131
F.1.14. ext_kill_child_storage . . . 132
F.1.15. Memory . . . 132
F.1.15.1. ext_malloc . . . 132
F.1.15.2. ext_free . . . 132
F.1.15.3. Input/Output . . . 132
F.1.16. Cryptograhpic Auxiliary Functions . . . 132
F.1.16.1. ext_blake2_256 . . . 132
F.1.16.2. ext_keccak_256 . . . 133
F.1.16.3. ext_twox_128 . . . 133
F.1.16.4. ext_ed25519_verify . . . 133
F.1.16.5. ext_sr25519_verify . . . 134
F.1.16.6. To be Specced . . . 134
F.1.17. Offchain Worker . . . 134
F.1.17.1. ext_is_validator . . . 135
F.1.17.2. ext_submit_transaction . . . 135
F.1.17.3. ext_network_state . . . 135
F.1.17.4. ext_timestamp . . . 135
F.1.17.5. ext_sleep_until . . . 136
F.1.17.6. ext_random_seed . . . 136
F.1.17.7. ext_local_storage_set . . . 136
F.1.17.8. ext_local_storage_compare_and_set . . . 136
F.1.17.9. ext_local_storage_get . . . 137
F.1.17.10. ext_http_request_start . . . 137
F.1.17.11. ext_http_request_add_header . . . 138
F.1.17.12. ext_http_request_write_body . . . 138
F.1.17.13. ext_http_response_wait . . . 138
F.1.17.14. ext_http_response_headers . . . 139
F.1.17.15. ext_http_response_read_body . . . 139
F.1.18. Sandboxing . . . 139
F.1.18.1. To be Specced . . . 139
F.1.19. Auxillary Debugging API . . . 140
F.1.19.1. ext_print_hex . . . 140
F.1.19.2. ext_print_utf8 . . . 140
F.1.20. Misc . . . 140

F.1.20.1. To be Specced 140
F.1.21. Block Production 140
F.2. Validation 140

APPENDIX G. RUNTIME ENTRIES 141

G.1. List of Runtime Entries 141
G.2. Argument Specification 141
G.2.1. `Core_version` 142
G.2.2. `Core_execute_block` 142
G.2.3. `Core_initialize_block` 142
G.2.4. `hash_and_length` 143
G.2.5. `BabeApi_configuration` 143
G.2.6. `GrandpaApi_grandpa_authorities` 143
G.2.7. `TaggedTransactionQueue_validate_transaction` 143
G.2.8. `BlockBuilder_apply_extrinsic` 145
G.2.9. `BlockBuilder_inherent_extrinsics` 147
G.2.10. `BlockBuilder_finalize_block` 147

GLOSSARY 149

BIBLIOGRAPHY 151

INDEX 153

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 most of the state transition logic for the Polkadot protocol and is designed and expected to be upgradable as part of the state transition process. The Polkadot Host consists of parts of the protocol, 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 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*

$$\delta: S \times \Sigma \rightarrow S$$

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 $(v_1, \ldots, v_n)$ 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:*

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

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. *By the* ***little-endian*** *representation of a non-negative integer, I, represented as*

$$I = (B_n ... B_0)_{256}$$

in base 256, we refer to a byte array $B = (b_0, b_1, ..., b_n)$ such that

$$b_i := B_i$$

Accordingly, define the function $\mathrm{Enc}_{\mathrm{LE}}$*:*

$$\begin{array}{rlcl} \mathrm{Enc}_{\mathrm{LE}}\colon & \mathbb{Z}^+ & \rightarrow & \mathbb{B} \\ & (B_n ... B_0)_{256} & \mapsto & (B_0, B_1, ..., B_n) \end{array}$$

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 $B_1 \rightarrow B_2$ 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 86400 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 all the nodes in the system such as every such block is a node in the graph and B_1 is connected to B_2 if B_1 is a parent of B_2.*

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 6.42. 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 $B := \textsc{Head}(C)$. We define $|C|$, the length of C as a path graph. If B' is another node on $\textsc{Chain}(B)$, then by $\textsc{SubChain}(B', B)$ we refer to the subgraph of $\textsc{Chain}(B)$ path graph which contains both B and B' and by $|\textsc{SubChain}(B', B)|$ we refer to its length. Accordingly, $\mathbb{C}_{B'}((P)\mathrm{BT})$ is the set of all subchains of $(P)\mathrm{BT}$ rooted at B'. The set of all chains of $(P)\mathrm{BT}$, $\mathbb{C}_G((P)\mathrm{BT})$ 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 $\mathrm{Head}(C_1)$ is less than the block arrival time of $\mathrm{Head}(C_2)$ as defined in Definition 6.17. 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 $(P)\mathrm{BT}$ which is the longest among all paths in $(P)\mathrm{BT}$ and has the earliest block arrival time as defined in Definition 6.17.* 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* $\boldsymbol{B > B'}$ *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 :*

$$\begin{array}{rc} \text{StoredValue:} & \mathcal{K} \rightarrow \mathcal{V} \\ & k \mapsto \left\{ \begin{array}{ll} v & \textit{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{array} \right. \end{array}$$

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 *modified Merkle Patricia Tree*, which hereafter we refer to 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 Trie, the key k is encoded to k_{enc} using KeyEncode functions:*

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

such that:

$$\text{KeyEncode}(k)\colon \left\{ \begin{array}{lll} \mathbb{B} & \rightarrow & \text{Nibbles}^4 \\ k := (b_1, \ldots, b_n) := & \mapsto & (b_1^1, b_1^2, b_2^1, b_2^2, \ldots, b_n^1, b_n^2) \\ & & := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) \end{array} \right.$$

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*:*

$$(b_i^1, b_i^2) := (b_i / 16, b_i \text{mod}\, 16)$$

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

Definition 2.4. *The State Trie is a radix-16 tree. Each Node 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 is a node which has one child or more. A branch node can have at most 16 children. A* ***leaf*** *node is a childless node. Accordingly:*

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

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***, $\mathbf{pk}_{\boldsymbol{N}}^{\mathbf{Agr}}$, *aggregated by Algorithm 2.1 and a* ***partial key***, $\mathbf{pk}_{\boldsymbol{N}}$ *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* $\mathbf{Index}_{\boldsymbol{N}}$ *function as:*

$$\begin{array}{rl} \text{Index}_N\colon & \{N_c \in \mathcal{N} | N_c \textit{ is a child of } N\} \rightarrow \text{Nibbles}_1^4 \\ & N_c \mapsto i \end{array}$$

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: =(TrieRoot $= N_1, ..., N_j = N$))

```
$\text{pk}_N^{\text{Agr}} \leftarrow \phi$
$i \leftarrow 1$
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 $\text{pk}_N^{\text{Agr}}$
```

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

Node Header	Partial key	Node Subvalue

Formally noted as:

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

where Head_N*,* pk_N*,* $\text{Enc}_{\text{nibbles}}$ *and* sv_N *are defined in Definitions 2.9,2.6, B.13 and 2.11, respectively.*

Definition 2.9. *The* ***node header*** *of node N,* Head_N*, consists of $l + 1 \geqslant 1$ bytes* $\text{Head}_{N,1}, ...,$ $\text{Head}_{N,l+1}$ *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 $\|\mathbf{pk}_{\boldsymbol{N}}\|_{\mathbf{nib}}$ *is the length of* pk_N *in number nibbles.* $\text{Head}_{N,2}, ..., \text{Head}_{N,l+1}$ *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** $\text{Head}_N$
    $\text{Head}_{N,1} \leftarrow \text{Head}_{N,1} + 63$
    $l \leftarrow \|\text{pk}_N\|_{\text{nib}} - 63$
    $i \leftarrow 2$
    **while** $(l > 255)$
        $\text{Head}_{N,i} \leftarrow 255$
        $l \leftarrow l - 255$
        $i \leftarrow i + 1$
    $\text{Head}_{N,i} \leftarrow l$
    **return** $\text{Head}_N$
```

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. Additionally, as Section 2.2.1 clarifies, the Merkle proof of each **child trie** must be updated first before the final Polkadot state root can be calculated.

We use the auxilary function introduced in Definition 2.10 to encode and decode information stored in a branch node.

Definition 2.10. *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:*

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

where

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

Definition 2.11. *For a given node N, the **subvalue** of N, formally referred to as sv_N, is determined as follows: in a case which:*

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

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

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

$N_{C_1} \ldots N_{C_n}$ with $n \leqslant 16$ are the children nodes of the branch node N and Enc_{SC}, StoredValue, H, and ChildrenBitmap(N) are defined in Definitions B.1,2.1, 2.12 and 2.10 respectively.

The Trie deviates from a traditional Merkle tree where node value, v_N (see Definition 2.8) is presented instead of its hash if it occupies less space than its hash.

Definition 2.12. *For a given node N, the **Merkle value** of N, denoted by $H(N)$ is defined as follows:*

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

*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 $H(R)$.*

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

In the exact way that the state trie is used to track and verify changes in the state storage, the changes in the child storage are tracked and verified. Therefore, 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 $\mathcal{N}$ must be updated first. After that, the final Merkle proof of the Polkadot state $\mathcal{N}$ can be calculated. 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 state changes via 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 Runtime as defined in Definition 1.1. Runtime is presented as a Wasm blob in order to be easily upgradable. Nonetheless, the Polkadot Host needs to be in constant interaction with 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.

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

3.1. Interactions with Runtime

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 Runtime easily upgradable without the need to upgrade the Polkadot Host itself. The general procedure to interact with Runtime is described in Algorithm 3.1.

```
Algorithm 3.1. Interact-With-Runtime($F$: the runtime entry,
               $H_b(B)$: 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 E.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:

$$b := \text{3A,63,6F,64,65}$$

which is the byte array of ASCII representation of string ":code" (see Section C). For any call to the Runtime, the Polkadot Host makes sure that it has the Runtime corresponding to the state in which the entry has been called. This is, in part, because the calls to Runtime have potentially the ability to change the Runtime code and hence Runtime code is state sensitive. 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 whose state is set at the end of the execution of block B.*

The initial runtime code of the chain is embedded as an extrinsics into the chain initialization JSON file (representing the genesis state) and is submitted to the Polkadot Host (see Section C).

Subsequent calls to the runtime have the ability to, in turn, call the storage API (see Section Tec19) to insert a new Wasm blob into runtime storage slot to upgrade the runtime.

3.1.2. Code Executor

The Polkadot Host provides a Wasm Virtual Machine (VM) to run the Runtime. The Wasm VM exposes the Polkadot Host API to the Runtime, which, on its turn, executes a call to the Runtime entries stored in the Wasm module. 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*

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

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 call into the Runtime. In Section G we specify the parameters and the return values of each Runtime entry separately.

3.1.2.1. Access to Runtime API

When the Polkadot Host calls a Runtime entry it should make sure Runtime has access to the all Polkadot Runtime API functions described in Appendix G. This can be done for example by loading another Wasm module alongside the runtime which imports these functions from the Polkadot Host as host functions.

3.1.2.2. Sending Arguments to Runtime

In general, all data exchanged between the Polkadot Host and the Runtime is encoded using SCALE codec described in Section B.1. As a Wasm function, all runtime entries have the following identical 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 encoded using SCALE codec into a byte array B using the procedure defined in Definition B.1.

The Executor then needs to retrieve the Wasm memory buffer of the Runtime Wasm module and extend it to fit the size of the byte array. Afterwards, it needs to copy the byte array B value in the correct offset of the extended buffer. Finally, when the Wasm method `runtime_entry`, corresponding to the entry is invoked, two UINT32 integers are sent to the method as arguments. The first argument `data` is set to the offset where the byte array B is stored in the Wasm the extended shared memory buffer. The second argument `len` sets the length of the data stored in B., and the second one is the size of B.

3.1.2.3. The Return Value 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 Tec19). 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 G.2.7), 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 entery. 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 G.2.2) 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 the state transition. This section describes the specifications of the extrinsics and their inclusion in the blocks.

3.2.1. Preliminaries

The extrinsics are divided in two main categories and 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

3.2.2.1. Transaction Submission

Transaction submission is made by sending a *Transactions* network message. The structure of this message is specified in Section D.1.5. Upon receiving a Transactions message, the Polkadot Host decodes the SCALE-encoded blob and decouples the transactions into individually SCALE-encoded transactions. Afterward, it should call `validate_trasaction` Runtime entry on each individual transaction, defined in Section G.2.7, to check the validity of the received transaction. If `validate_transaction` considers the submitted transaction as a valid one, the Polkadot Host makes the transaction available for the consensus engine for inclusion in future blocks.

3.2.3. Transaction Queue

A Block producer node should listen to all transaction messages. The transactions are submitted to the node through the *transactions* network message specified in Section D.1.5. Upon receiving a transactions message, the Polkadot Host separates the submitted transactions in the transactions message into individual transactions and passes them to the Runtime by executing Algorithm 3.2 to validate and store them for inclusion into future blocks. Valid transactions are propagated to connected peers of the Polkadot Host. Additionally, 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 D.1.5). 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)

```
$L \leftarrow \text{Dec}_{\text{SC}}(M_T)$
for $T$ in $L$ such that $E \notin \text{TQ}$ and $E \notin \text{TP}$:
    $B_d \leftarrow$ 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$_{\text{Sc}}$ decodes the SCALE encoded message.
- Longest-Chain is defined in Definition 1.14.
- `TaggedTransactionQueue_validate_transaction` is a Runtime entry specified in Section G.2.7 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 G.2, 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]

3.2.3.1. Inherents

Inherents are unsigned pieces of information and only inserted into a block by the block author. Those entries are not gossiped on the network or stored in the transaction queue. It is up to the Polkadot Host to decide the validity of those entries and mostly serve as unverified metadata.

Block inherent data represents the totality of inherent extrinsics included in each block. This data is collected or generated by the Polkadot Host and handed to the Runtime for inclusion in the block. It's the responsability of the Polkadot Host implementation to keep track of those values. Table 3.1 lists these inherent data, identifiers, and types. [define uncles]

Identifier	Value type	Description
timstap0	u64	Unix epoch time in number of milliseconds
finalnum	compact integer$^{B.12}$	Header number$^{3.6}$ of the last finalized block
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.7), 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 happens across 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

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

3.3.1.1. Block Header

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

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

- **parent_hash:** *formally indicated as 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 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 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 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.3) such that*

$$H_d(B){:}=H_d^1,...,H_d^n$$

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*	$E_{\rm id}, \mathbb{B}$
4	*Consensus Message*	$E_{\rm id}, \mathbb{B}$
5	*Seal*	$E_{\rm id}, \mathbb{B}$

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

Where $E_{\rm id}$ *is the unique consensus engine identifier defined in Section D.1.6. 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 item represents messages produced by a consensus engine to the Runtime.*
- ***Consensus Message*** *digest item represents a message 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 6.20.*

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 simple 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 3.3.1.1 and denoted by Head(B).
- **justification**: as defined by the consensus specification indicated by Just(B) [link this to its definition from consensus].
- **authority Ids**: This is the list of the Ids of authorities, which have voted for the block to be stored and is formally referred to as $A(B)$. An authority Id is 32bit.

3.3.1.3. Block Body

The Block Body consists of array extrinsics each encoded as a byte array. The internal of extrinsics 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* $\mathbf{Body}(\boldsymbol{B})$ *is defined to be*

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

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

Blocks can be handed to the Polkadot Host both from the network stack for example by means of Block response network message (see Section D.1.3) and from the consensus engine.Both the Runtime and the Polkadot Host need to work together to assure block validity. A block is deemed valid if the block author had the authorship right for the slot during which the slot was built 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 G.2.2 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))

```
Set-Storage-State-At($P(B)$)
if Just($B$) $\neq \emptyset$
    Verify-Block-Justification($B$, Just($B$))
    if $B$ is Finalized and $P(B)$ is not Finalized
        Mark-as-Final($P(B)$)
if $H_p(B) \notin$ PBT
    return
Verify-Authorship-Right(Head($B$))
$B \leftarrow$ Remove-Seal($B$)
$R \leftarrow$ Call-Runtime-Entry(Core_execute_block, $B$)
$B \leftarrow$ Add-Seal($B$)
if $R$ = True
    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.

For the definition of the finality and the finalized block see Section 6.3. 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.

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 6.42), 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*

$$\textbf{Set-State-At}(B)$$

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 which allows identifying changes in the blockchain without requiring 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 E.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

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

and

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

is SCALE encoded byte array.

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

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):

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

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):

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

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 $l \in \{1, \dots, L\}$
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:

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

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

Chapter 4

Networking

Chapter Status: This document in its current form is incomplete and considered work in progress. Any reports regarding falseness or clarifications are appreciated.

4.1. Introduction

The Polkadot network is decentralized and does not rely on any central authority or entity in order to achieve a its fullest potential of provided functionality. Each node with the network can authenticate itself and its peers by using cryptographic keys, including establishing fully encrypted connections. The networking protocol is based on the open and standardized `libp2p` protocol, including the usage of the distributed Kademlia hash table for peer discovery.

4.1.1. External Documentation

The completeness of implementing the Polkadot networking protocol requires the usage of external documentation.

- libp2p
- Kademlia
- Noise
- Protocol Buffers

4.1.2. Discovery mechanism

The Polkadot Host uses varies mechanism 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.

The Polkadot Host uses various mechanism for peer dicovery.

- Bootstrap nodes - hard-coded node identities and addresses provided by network configuration itself. Those addresses are selected an updated by the developers of the Polkadot Host. Node addresses should be selected based on a reputation metric, such as reliability and uptime.
- mDNS - performs a broadcast to the local network. Nodes that might be listing can respond the the broadcast.
- Kademlia requests - Kademlia supports `FIND_NODE` requests, where nodes respond with their list of available peers.

4.1.3. Connection establishment

The Polkadot Host can establish a connection with any peer it knows the address. `libp2p` uses the `multistream-select` protocol in order to establish an encryption and multiplexing layer. The Polkadot Host supports multiple base-layer protocols:

- TCP/IP - addresses in the form of `/ip4/1.2.3.4/tcp/` establish a TCP connection and negotiate a encryption and multiplexing layer.
- Websockets - addresses in the form of `/ip4/1.2.3.4/ws/` establish a TCP connection and negotiate the Websocket protocol within the connection. Additionally, a encryption and multiplexing layer is negotiated within the Websocket connection.

- DNS - addresses in form of `/dns/website.domain/tcp/` and `/dns/website.domain/ws/`.

After a base-layer protocol is established, the Polkadot Host will apply the Noise protocol.

4.1.4. Substreams

After the node establishes a connection with a peer, the use of multiplexing allows the Polkadot Host to open substreams. Substreams allow the negotiation of *application-specific protocols*, where each protocol servers a specific utility.

The Polkadot Host adoptes the following, standardized `libp2p` application-specific protocols:

- `/ipfs/ping/1.0.0` - Open a substream to a peer and initialize a ping to verify if a connection is till alive. If the peer does not respond, the connection is dropped.
- `/ipfs/id/1.0.0` - Open a substream to a peer to ask information about that peer.
- `/dot/kad/` - Open a substream for Kademlia `FIND_NODE` requests.

Additional, non-standardized protocols:

- `/dot/sync/2` - a request and response protocol that allows the Polkadot Host to perform information about blocks.
- `/dot/light/2` - a request and response protocol that allows a light client to perform information about the state.
- `/dot/transactions/1` - a notification protocol which sends transactions to connected peers.
- `/dot/block-announces/1` - a notification protocol which sends blocks to connected peers.

4.2. Network Messages

4.2.1. API Package

ProtoBuf details:

- syntax: proto3
- package: api.v1

4.2.1.1. BlockRequest

Request block data from a peer.

```
message BlockRequest {
    // Bits of block data to request.
    uint32 fields = 1;
    // Start from this block.
    oneof from_block {
        // Start with given hash.
        bytes hash = 2;
        // Start with given block number.
        bytes number = 3;
    }
    // End at this block. An implementation defined
    // maximum is used when unspecified.
    bytes to_block = 4; // optional
    // Sequence direction.
    Direction direction = 5;
    // Maximum number of blocks to return. An implementation
    // defined maximum is used when unspecified.
    uint32 max_blocks = 6; // optional
}
```

```
// Block enumeration direction
enum Direction {
    // Enumerate in ascending order
    // (from child to parent).
    Ascending = 0;
    // Enumerate in descending order
    // (from parent to canonical child).
    Descending = 1;
}
```

4.2.1.2. BlockResponse

Response to Block Request.

```
message BlockResponse {
    // Block data for the requested sequence.
    repeated BlockData blocks = 1;
}

// Block data sent in the response.
message BlockData {
    // Block header hash.
    bytes hash = 1;
    // Block header if requested.
    bytes header = 2; // optional
    // Block body if requested.
    repeated bytes body = 3; // optional
    // Block receipt if requested.
    bytes receipt = 4; // optional
    // Block message queue if requested.
    bytes message_queue = 5; // optional
    // Justification if requested.
    bytes justification = 6; // optional
    // True if justification should be treated as present but
    // empty. This hack is unfortunately necessary because
    // shortcomings in the protobuf format otherwise doesn't
    // make it possible to differentiate between a lack of
    // justification and an empty justification.
    bool is_empty_justification = 7; // optional, false if absent
}
```

4.2.2. Light Package

ProtoBuf details:

- syntax: proto3
- package: api.v1.light

4.2.2.1. Request

Enumerate all possible light client request messages.

```
message Request {
    oneof request {
        RemoteCallRequest remote_call_request = 1;
        RemoteReadRequest remote_read_request = 2;
```

```
            RemoteHeaderRequest remote_header_request = 3;
            RemoteReadChildRequest remote_read_child_request = 4;
            RemoteChangesRequest remote_changes_request = 5;
        }
    }
```

4.2.2.2. Response

Enumerate all possible light client response messages.

```
message Response {
    oneof response {
        RemoteCallResponse remote_call_response = 1;
        RemoteReadResponse remote_read_response = 2;
        RemoteHeaderResponse remote_header_response = 3;
        RemoteChangesResponse remote_changes_response = 4;
    }
}
```

4.2.2.3. RemoteCallRequest

Remote call request.

```
message RemoteCallRequest {
    // Block at which to perform call.
    bytes block = 2;
    // Method name.
    string method = 3;
    // Call data.
    bytes data = 4;
}
```

4.2.2.4. RemoteCallResponse

Remote call response.

```
message RemoteCallResponse {
    // Execution proof.
    bytes proof = 2;
}
```

4.2.2.5. RemoteReadRequest

Remote storage read request.

```
message RemoteReadRequest {
    // Block at which to perform call.
    bytes block = 2;
    // Storage keys.
    repeated bytes keys = 3;
}
```

4.2.2.6. RemoteReadResponse

Remote read response.

```
message RemoteReadResponse {
    // Read proof.
    bytes proof = 2;
}
```

4.2.2.7. RemoteReadChildRequest

Remote storage read child request.

```
message RemoteReadChildRequest {
    // Block at which to perform call.
    bytes block = 2;
    // Child Storage key, this is relative
    // to the child type storage location.
    bytes storage_key = 3;
    // Storage keys.
    repeated bytes keys = 6;
}
```

4.2.2.8. RemoteHeaderRequest

Remote header request.

```
message RemoteHeaderRequest {
    // Block number to request header for.
    bytes block = 2;
}
```

4.2.2.9. RemoteHeaderResponse

Remote header response.

```
message RemoteHeaderResponse {
    // Header. None if proof generation has failed
    // (e.g. header is unknown).
    bytes header = 2; // optional
    // Header proof.
    bytes proof = 3;
}
```

4.2.2.10. RemoteChangesRequest

Remote changes request.

```
message RemoteChangesRequest {
    // Hash of the first block of the range (including first)
    // where changes are requested.
    bytes first = 2;
    // Hash of the last block of the range (including last)
    // where changes are requested.
    bytes last = 3;
    // Hash of the first block for which the requester has
    // the changes trie root. All other
    // affected roots must be proved.
    bytes min = 4;
```

```
        // Hash of the last block that we can use when
        // querying changes.
        bytes max = 5;
        // Storage child node key which changes are requested.
        bytes storage_key = 6; // optional
        // Storage key which changes are requested.
        bytes key = 7;
    }
```

4.2.2.11. RemoteChangesResponse

Remote changes response.

```
    message RemoteChangesResponse {
        // Proof has been generated using block with this number
        // as a max block. Should be less than or equal to the
        // RemoteChangesRequest::max block number.
        bytes max = 2;
        // Changes proof.
        repeated bytes proof = 3;
        // Changes tries roots missing on the requester' node.
        repeated Pair roots = 4;
        // Missing changes tries roots proof.
        bytes roots_proof = 5;
    }

    // A pair of arbitrary bytes.
    message Pair {
        // The first element of the pair.
        bytes fst = 1;
        // The second element of the pair.
        bytes snd = 2;
    }
```

4.2.3. Finality Package

ProtoBuf details:

- syntax: proto3
- package: api.v1.finality

4.2.3.1. FinalityProofRequest

Request a finality proof from a peer.

```
    message FinalityProofRequest {
        // SCALE-encoded hash of the block to request.
        bytes block_hash = 1;
        // Opaque chain-specific additional request data.
        bytes request = 2;
    }
```

4.2.3.2. FinalityProofResponse

Response to a finality proof request.

```
message FinalityProofResponse {
    // Opaque chain-specific finality proof.
    // Empty if no such proof exists.
    bytes proof = 1; // optional
}
```

4.3. To be migrated

Warning 4.1. Polkadot network protocol is work-in-progress. The API specification and usage may change in future.

This chapter offers a high-level description of the network protocol based on [Tec19]. Polkadot network protocol relies on *libp2p*. Specifically, the following libp2p modules are being used in the Polkadot Networking protocol:

- mplex.
- yamux
- secio
- noise
- kad (kademlia)
- identity
- ping

For more detailed specification of these modules and the Peer-to-Peer layer see libp2p specification document [lab19].

4.3.1. Node Identities and Addresses

Similar to other decentralized networks, each Polkadot Host node possesses a network private key and a network public key representing an ED25519 key pair [LJ17].

[SPEC: local node's keypair must be passed as part of the network configuration.]

Definition 4.2. ***Peer Identity***, *formally noted by P_{id} is derived from the node's public key as follows:*

[SPEC: How to derive P_{id}] and uniquely identifies a node on the network.

Because the P_{id} is derived from the node's public key, running two or more instances of Polkadot network using the same network key is contrary to the Polkadot protocol.

All network communications between nodes on the network use encryption derived from both sides' keys.

[SPEC: p2p key derivation]

4.3.2. Discovery Mechanisms

In order for a Polkadot node to join a peer-to-peer network, it has to know a list of Polkadot nodes that already take part in the network. This process of building such a list is referred to as *Discovery*. Each element of this list is a pair consisting of the peer's node identities and their addresses.

[SPEC: Node address]

Polkadot discovery is done through the following mechanisms:

- *Bootstrap nodes*: These are hard-coded node identities and addresses passed alongside with the network configuration.
- *mDNS*, performing a UDP broadcast on the local network. Nodes that listen may respond with their identity as described in the mDNS section of [lab19]. (Note: mDNS can be disabled in the network configuration.)

- *Kademlia random walk*. Once connected to a peer node, a Polkadot node can perform a random Kademlia ‘FIND_NODE‘ requests for the nodes [which nodes?] to respond by propagating their view of the network.

4.3.3. Transport Protocol

A Polkadot node can establish a connection with nodes in its peer list. All the connections must always use encryption and multiplexing. While some nodes' addresses (eg. addresses using ‘/quic‘) already imply the encryption and/or multiplexing to use, for others the “multistream-select” protocol is used in order to negotiate an encryption layer and/or a multiplexing layer.

The following transport protocol is supported by a Polkadot node:

- *TCP/IP* for addresses of the form ‘/ip4/1.2.3.4/tcp/5‘. Once the TCP connection is open, an encryption and a multiplexing layers are negotiated on top.
- *WebSockets* for addresses of the form ‘/ip4/1.2.3.4/tcp/5/ws‘. A TC/IP connection is open and the WebSockets protocol is negotiated on top. Communications then happen inside WebSockets data frames. Encryption and multiplexing are additionally negotiated again inside this channel.

• DNS for addresses of the form ‘/dns4/example.com/tcp/5‘ or ‘/dns4/example.com/tcp/5/ws‘. A node's address can contain a domain name.

4.3.3.1. Encryption

The following encryption protocols from libp2p are supported by Polkadot protocol:

∗ **Secio**: A TLS-1.2-like protocol but without certificates [lab19]. Support for secio will likely to be deprecated in the future.

∗ **Noise**: Noise is a framework for crypto protocols based on the Diffie-Hellman key agreement [Per18]. Support for noise is experimental and details may change in the future.

4.3.3.2. Multiplexing

The following multiplexing protocols are supported:

- **Mplex**: Support for mplex will be deprecated in the future.
- **Yamux**.

4.3.4. Substreams

Once a connection has been established between two nodes and is able to use multiplexing, substreams can be opened. When a substream is open, the *multistream-select* protocol is used to negotiate which protocol to use on that given substream.

4.3.4.1. Periodic Ephemeral Substreams

A Polkadot Host node should open several substreams. In particular, it should periodically open ephemeral substreams in order to:

- ping the remote peer and check whether the connection is still alive. Failure for the remote peer to reply leads to a disconnection. This uses the libp2p *ping* protocol specified in [lab19].
- ask information from the remote. This is the *identity* protocol specified in [lab19].
- send Kademlia random walk queries. Each Kademlia query is done in a new separate substreams. This uses the libp2p *Kademlia* protocol specified in [lab19].

4.3.4.2. Polkadot Communication Substream

For the purposes of communicating Polkadot messages, the dailer of the connection opens a unique substream. Optionally, the node can keep a unique substream alive for this purpose. The name of the protocol negotiated is based on the *protocol ID* passed as part of the network configuration. This protocol ID should be unique for each chain and prevents nodes from different chains to connect to each other.

The structure of SCALE encoded messages sent over the unique Polkadot communication substream is described in Appendix D.

Once the substream is open, the first step is an exchange of a *status* message from both sides described in Section D.1.1.

Communications within this substream include:

- Syncing. Blocks are announced and requested from other nodes.
- Gossiping. Used by various subprotocols such as GRANDPA.
- Polkadot Network Specialization: [spec this protocol for polkadot].

□

Chapter 5

Bootstrapping

[https://github.com/w3f/polkadot-spec/issues/135]

Chapter 6

Consensus

Consensus in the Polkadot Host is achieved during the execution of two different procedures. The first procedure is 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 engine 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. In order to verifiy the validity of each block, Polkadot node must track the current list of authorities for that block as formalised in Definition 6.1

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

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

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* $\mathrm{Auth}_C(B)$ *is part of the Polkadot state. The value for* $\mathrm{Auth}_C(B_0)$ *is set in the genesis state (see Section C) and can be retrieved using a runtime entery 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 appropiate consensus message as defined in Definition 6.3, in form of a digest item to the block header of the block B which caused the transition in the authority set.

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

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

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

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

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

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

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 6.21*
- 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 6.10. 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_{2nd} *is the the second slot configuration encoded as a 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.*
- $N_{\text{delay}} := |\textsc{SubChain}(B, B')|$ *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 consesus 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 is intended to be used in 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 change. 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 pretain 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 $N_{\text{delay}} := |\textsc{SubChain}(B, B')|$ 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 and the delay has not passed completely. If such an inconsitency occures, the scheduled change should be ignored.

- **Forced Change**: Force an authority set change after the given delay of $N_{\text{delay}} := |\textsc{SubChain}(B, B')|$ where B' in the definition of N_{delay}, is an *imported* block which has been validated by the block production conensus engine. Hence, the authority change set is valid for every subchain which contains 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 and the delay has not passed completely. If such an inconsitency occures, the scheduled change should be ignored.
- **On Disabled**: An index to the individual authority in the current authority list that should be immediately disabled until the next authority set change. When an authority gets disabled, the node should stop performing any authority functionality from that authority, including authoring blocks and casting GRANDPA votes for finalization. Similarly, other nodes should ignore all messages from the indicated authority which pretain to their authority role.
- **Pause**: A signal to pause the current authority set after the given delay of $N_{\text{delay}} := |\textsc{SubChain}(B, B')|$ 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 $N_{\text{delay}} := |\textsc{SubChain}(B, B')|$ where B' in the definition of N_{delay}, is an *imported* block and validated by the block production consensus engine. After authoring the block B', the authorities should resume voting.

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

6.2. Block Production

The Polkadot Host uses BABE protocol [Gro19] for block production. It is designed based on Ouroboros praos [DGKR18]. 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 which 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 gets a turn in producing blocks.*

Definition 6.5. ***Block authoring session key pair** $(\mathbf{sk}^s_j, \mathbf{pk}^s_j)$ 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 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 **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 prefered 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 G.2.5) 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 *which contains 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)*.*

6.2.2. Block Production Lottery

DEFINITION 6.10. *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 6.11).*

The babe constant (Definition 6.10) is initialized at genesis to the value returned by calling `BabeApi_configuration` (see G.2.5). 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 6.11. ***Winning threshold*** *denoted by* $\boldsymbol{\tau_{\varepsilon_n}}$ *is the threshold which is used alongside with the result of Algorirthm 6.1 to decide if a block producer is the winner of a specific slot.* τ_{ε_n} *is calculated as follows:*

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

where $\mathrm{AuthorityDirectory}^{\mathcal{E}_n}$ *is the set of BABE authorities for epoch* ε_n *and* $c \in (0,1)$ *is the BABE constant as defined in definition 6.10.*

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 epoch for whose slots the block producer is running the lottery.

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

```
$r \leftarrow$ EPOCH-RANDOMNESS($n$)
for $i := 1$ to $\mathrm{sc}_n$
    $(\pi, d) \leftarrow \mathrm{VRF}(r, i, \mathrm{sk})$
    $A[i] \leftarrow (d, \pi)$
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 6.12. *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 time continuum. As such, we have* SLOT-OFFSET$(s_i, s_i) = 0$.

It is imperative for the security of the network that each block producer correctly determine the current slots number at a given time by regularly estimating the local clock offset in relation to the network (Definition 6.13).

Definition 6.13. *The* ***relative time syncronization*** *is a tuple of a slot number and local clock timestamp* $(s_{\text{sync}}, t_{\text{sync}})$ *describing the last point at which slot numbers have been syncronized with the local clock.*

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

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

Note 6.14. The calculation described in this section is still to be implemented and deployed. For now each block producer is required to syncronize 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 syncronization 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 6.15) and $\boldsymbol{s_{\text{cq}}}$ (Definition 6.16). These are choosen based on the results of 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 a resistance against network delay up to $1/3$ of the slot time and a Babe constant (Definitwion 6.10) of $c = 0.38$.

Definition 6.15. *The* ***prunned best chain*** $\boldsymbol{C^{\ulcorner k}}$ *is the longest chain selected according to Definition 1.14 with the last k Blocks prunned. We choose $k = 140$. The* ***last (probabilistically) finalized block*** *describes the last block in this prunned best chain.*

Definition 6.16. *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 6.17) measured by a clock that is otherwise not adjusted by any external protocol.

Definition 6.17. *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 the 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 clean definition of when block arrival times are considered valid and how to differentiated imported block on intial sync from "fresh" blocks that were just produced.]

Definition 6.18. *A* ***sync period*** *is the interval at which each validator (re-)evaluates its local clock offsets. The first sync period $\mathfrak{C}_1$ starts just after the genesis block is released. Consequently each sync period $\mathfrak{C}_i$ starts after $\mathfrak{C}_{i-1}$. The length of sync period is equal to s_{qc} as defined in Definition 6.16 and expressed in number of slots.*

All validators are then required to run Algorithm 6.3 at beginning of each sync period (Definition 6.18) 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 6.15). The target slot to which to synchronize should be the first slot in the new sync period.

Algorithm 6.3. Median-Algorithm($\mathfrak{C}_j$: sync period used for estimate, s_{sync}: slot time to estimate)

```
$T_s \leftarrow \{\}$
```

```
for $B_i$ in $\mathfrak{C}_j$
    $t_{\text{estimate}}^{B_i} \leftarrow T_{B_i} + \text{SLOT-OFFSET}(s_{B_i}, s_{\text{sync}}) \times \mathcal{T}$
    $T_s \leftarrow T_s \cup t_{\text{estimate}}^{B_i}$
return Median($T_s$)
```

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

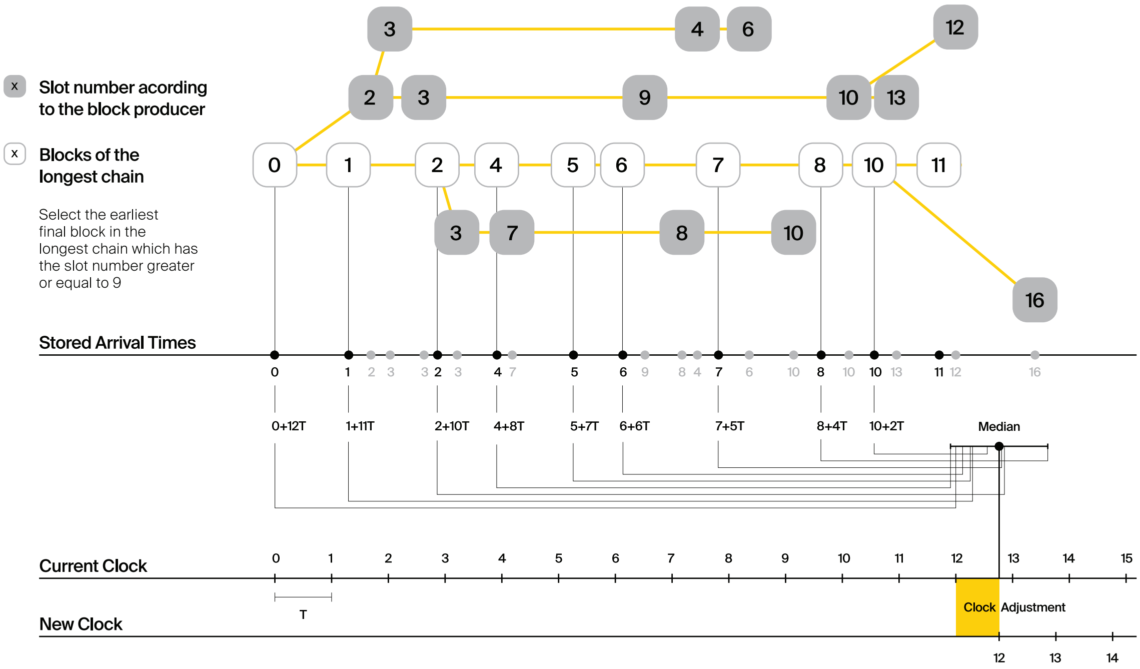
x Slot number acording to the block producer
x Blocks of the longest chain
Select the earliest final block in the longest chain which has the slot number greater or equal to 9
Stored Arrival Times
0+12T
1+11T
2+10T
4+8T
5+7T
6+6T
7+5T
8+4T
10+2T
Median
Current Clock
New Clock
T
Clock Adjustment

Figure 6.1. Examplary 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 *BABE header* as well as the *block signature* as Pre-Runtime and Seal digest items defined in Definition 6.19 and 6.20 respectively.

DEFINITION 6.19. *The* ***BABE Header*** *of block B, referred to formally by* $\boldsymbol{H_{\text{BABE}}(B)}$ *is a tuple that consists of the following components:*

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

in which:

π, d: *are the results of the block lottery for slot s.*

j: *is index of the block producer producing block in the current authority directory of current epoch.*

s: *is 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 6.20. *The* ***Block Signature*** S_B *is a signature of the block header hash (see Definition3.8) and defined as*

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

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, $E_{\text{id}}(\text{BABE})$ is the BABE consensus engine unique identifier defined in Section D.1.6. The Seal digest item is referred to as ***BABE Seal****.*

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

```
$A \leftarrow$ Block-production-lottery(sk, $n$)
for $s \leftarrow 1$ to $\text{sc}_n$
    Wait(until Slot-Time(s))
    $(d, \pi) \leftarrow A[s]$
    if $d < \tau$
        $C_{\text{Best}} \leftarrow$ Longest-Chain(BT)
        $B_s \leftarrow$ Build-Block($C_{\text{Best}}$)
        Add-Digest-Item($B_s$,Pre-Runtime,$E_{\text{id}}(\text{BABE})$, $H_{\text{BABE}}(B_s)$)
        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 6.21. *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.*

In the beginning of each epoch $\mathcal{E}_n$ the host will receive the randomness seed $\mathcal{R}_{\mathcal{E}_{n+1}}$(Definition 6.21) necessary to participate in the block production lottery in the next epoch $\mathcal{E}_{n+1}$ from the runtime, through the *Next Epoch Data* consesus 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 6.17.
- $H_d(B)$ 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.6 and 3.7)
- $\text{AuthorityDirectory}^{\mathcal{E}_c}$ 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)

```
$s \leftarrow$ Slot-Number-At-Given-Time($T_B$)
$\mathcal{E}_c \leftarrow$ Current-Epoch()
$(D_1, \ldots, D_{\text{length}(H_d(B))}) \leftarrow H_d(B)$
$D_s \leftarrow D_{\text{length}(H_d(B))}$
$H_d(B) \leftarrow (D_1, \ldots, D_{\text{length}(H_d(B))-1})$ //remove the seal from the digest
$(\text{id}, \text{Sig}_B) \leftarrow \text{Dec}_{\text{SC}}(D_s)$
if $\text{id} \neq$ Seal-Id
    error "Seal missing"
```

```
AuthorID ←AuthorityDirectory$^{\mathcal{E}_c}$[$H_{\mathrm{BABE}}(B)$.SingerIndex]
Verify-Signature(AuthorID, $H_h(B)$, $\mathrm{Sig}_B$)
if $\exists B' \in \mathrm{BT}: H_h(B) \neq H_h(B)$ and $s_B = s'_B$ and $\mathrm{SignerIndex}_B = \mathrm{SignerIndex}_{B'}$
    error “Block producer is equivocating”
Verify-Slot-Winner(($d_B, \pi_B$), $s$,AuthorID)
```

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

- Epoch-Randomness is defined in Definition 6.21.
- $H_{\mathrm{BABE}}(B)$ is the BABE header defined in Definition 6.19.
- Verify-VRF is described in Section A.4.
- τ is the winning threshold defined in 6.11.

Algorithm 6.6. Verify-Slot-Winner(B: the block whose winning status to be verified)

```
$\mathcal{E}_c \leftarrow$ Current-Epoch
$\rho \leftarrow$ Epoch-Randomness($c$)
Verify-VRF($\rho, H_{\mathrm{BABE}}(B).(d_B, \pi_B), H_{\mathrm{BABE}}(B).s, c$)
if $d_B \geqslant \tau$
    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 where at its head, the block to be constructed, s: Slot number)

```
$P_B \leftarrow$ Head($C_{\mathrm{Best}}$)
Head($B$) $\leftarrow (H_p \leftarrow H_h(P_B), H_i \leftarrow H_i(P_B) + 1, H_r \leftarrow \phi, H_e \leftarrow \phi, H_d \leftarrow \phi)$
Call-Runtime-Entry(Core_initialize_block, Head($B$))
$I\text{-}D$ ← Call-Runtime-Entry(BlockBuilder_inherent_extrinsics, Inherent-Data)
for $E$ in $I\text{-}D$
    Call-Runtime-Entry(BlockBuilder_apply_extrinsics,E)
while not End-Of-Slot(s)
    $E \leftarrow$ Next-Ready-Extrinsic()
    $R \leftarrow$ Call-Runtime-Entry(BlockBuilder_apply_extrinsics,$E$)
    if not Block-Is-FULL($R$)
        Drop(Ready-Extrinsics-Queue,$E$)
    else
        break
Head($B$) $\leftarrow$ Call-Runtime-Entry(BlockBuilder_finalize_block,$B$)
```

- 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.
- TRANSACTION-QUEUE is defined in Definition 3.4.
- BLOCK-IS-FULL indicates that the maximum block size as been used.
- END-OF-SLOT indicates the end of the BABE slot as defined in Algorithm 6.3 respectively Definition 6.6.
- OK-RESULT indicates whether the result of `BlockBuilder_apply_extrinsics` is successfull. The error type of the Runtime function is defined in Section G.2.8.
- READY-EXTRINSICS-QUEUE indicates picking an extrinsics from the extrinsics queue (Definition 3.4).
- DROP indicates removing the extrinsic from the transaction queue (Definition 3.4).

6.3. FINALITY

The Polkadot Host uses GRANDPA Finality protocol [Ste19] to finalize blocks. Finality is obtained by consecutive rounds of voting by validator nodes. Validators execute GRANDPA finality process in parallel to Block Production as an independent service. In this section, we describe the different functions that GRANDPA service performs to successfully participate in the block-finalization process.

6.3.1. Preliminaries

DEFINITION 6.22. *A **GRANDPA Voter**, v, is represented by a key pair $(k_v^{\text{pr}}, v_{\text{id}})$ where k_v^{pr} represents its private key which is an* ED25519 *private key, is a node running GRANDPA protocol and broadcasts 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 Runtime described in Section G.2.6. We refer to $\mathbb{V}_B$ as $\mathbb{V}$ when there is no chance of ambiguity.*

DEFINITION 6.23. *The **authority set Id** $(\text{id}_{\mathbb{V}})$ is an incremental counter which tracks the amount of authority list (Definition 6.3) changes that occurred. 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 64bit integer.*

DEFINITION 6.24. ***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 6.23.*
$\boldsymbol{r}$*: 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 6.25. *A **GRANDPA vote** or simply a vote for block B is an ordered pair defined as*

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

where $H_h(B)$ and $H_i(B)$ are the block hash and the block number defined in Definitions 3.6 and 3.8 respectively.

DEFINITION 6.26. *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 $\boldsymbol{V_v^{r,\text{pv}}}$ and $\boldsymbol{V_v^{r,\text{pc}}}$ 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.

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 6.27. *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 $V_v^{r,\text{stage}}(B)$ cast by v in that sub-round is an **equivocatory vote**, and*

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

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

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

DEFINITION 6.28. *A vote $V_v^{r,\text{stage}} = V(B)$ is **invalid** if*

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

DEFINITION 6.29. *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 <u>valid</u> votes $V_{v_i}^{r,\text{stage}}$ cast in round r and received by v such that $V_{v_i}^{r,\text{stage}} = V(B)$.*

DEFINITION 6.30. *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:*

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

DEFINITION 6.31. *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 6.32. *[Replace with GHOST] The current **pre-voted** block $B_v^{r,\text{pv}}$ is the block with*

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

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

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 finalised in this round.

Definition 6.33. *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. GRANDPA Messages Specification

Definition 6.34. ***GRANDPA Gossip*** *is a variant, as defined in Definition B.3, which identifies the message type that is casted by a voter. This type, followed by the sub-component, is sent to other validators.*

Id	*Type*
0	*Grandpa message (vote)*
1	*Grandpa pre-commit*
2	*Grandpa neighbor packet*
3	*Grandpa catch up request message*
4	*Grandpa catch up message*

Table 6.3.

The sub-components are the individual message types described in this section.

6.3.2.1. Vote Messages

Voting is done by means of broadcasting voting messages to the network. Validators inform their peers about the block finalized in round r by broadcasting a finalization message (see Algorithm 6.9 for more details). These messages are specified in this section.

Definition 6.35. $\mathbf{Sign}^{r,\text{stage}}_{v_i}$ *refers to the signature of a voter for a specific message in a round and is formally defined as:*

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

Where:

msg	*the message to be signed*	*arbitrary*
r:	*round number*	*unsigned 64-bit integer*
$\text{id}_{\mathbb{V}}$	*authority set Id (Definition 6.23) of v*	*unsigned 64-bit integer*

Table 6.4. *Signature for a message in a round.*

Definition 6.36. *A vote casted by voter v should be broadcasted as a* ***message*** $\boldsymbol{M^{r,\text{stage}}_v}$ *to the network by voter v with the following structure:*

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

Where:

r:	*round number*	*unsigned 64-bit integer*
$\mathrm{id}_{\mathbb{V}}$	*authority set Id (Definition 6.23)*	*unsigned 64-bit integer*
$\mathrm{Sig}_{v_i}^{r,\mathrm{stage}}$	*signature (Definition 6.35)*	*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*

This message is the sub-component of the GRANDPA Gossip as defined in Definition 6.34 of type Id 0 and 1.

6.3.2.2. Finalizing Message

Definition 6.37. *The* ***justification for block B in round r*** *of GRANDPA protocol defined $J^{r,\mathrm{stage}}(B)$ 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 6.35 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.*

We say $J^{r,\mathrm{pc}}(B)$ ***justifies the finalization*** *of $B' \geqslant B$ if the number of valid signatures in $J^{r,\mathrm{pc}}(B)$ for B' is greater than $\frac{2}{3}|\mathbb{V}_B|$. [It should be either the estimate of last round or estimate of this round]*

Definition 6.38. **GRANDPA** ***finalizing message for block B in round r*** *represented as* $\boldsymbol{M_v^{r,\mathrm{Fin}}(B)}$ *is a message broadcasted by voter v to the network indicating that voter v has finalized block B in round r. It has the following structure:*

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

in which $J^r(B)$ in the justification defined in Definition 6.37.

6.3.2.3. Catch-up Messages

Whenever a Polkadot node detects that it is lagging behind the finality procedure and therefore needs to initiate a catch-up procedure. Neighbor packet network message (see Section D.1.7) the round number for the last finalized GRANDPA round which the sending peer has observed. This provides a mean to identifys such a descrepency and to initiate the catch-up procedure explained in Section 6.4.1.

This procedure involves sending *catch-up request* and processing *catch-up response* messages specified here:

Definition 6.39. ***GRANDPA catch-up request message for round r*** *represented as* $\boldsymbol{M_{i,v}^{\mathrm{Cat}-q}(\mathrm{id}_{\mathbb{V}}, r)}$ *is a message sent from node i to its voting peer node v requesting the latest status of a GRANDPA round $r' > r$ of of authority set $\mathbb{V}_{\mathrm{id}}$ along with the justification of the status and has the followin structure:*

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

This message is the sub-component of the GRANDPA Gossip as defined in Definition 6.34 of type Id 3.

Definition 6.40. ***GRANDPA catch-up response message for round $\boldsymbol{r}$*** *formally denoted as* $\boldsymbol{M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)}$ *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 GRNADPA round which v has prove of its finalization and has the following structure:*

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

Where B is the highest block which v believes to be finalized in round r. B' is the highest anscestor of all blocks voted on in $J^{r,\text{pc}}(B)$ with the exception of the equivocationary votes. This message is the sub-component of the GRANDPA Gossip as defined in Definition 6.34 of type Id 4.

6.3.3. Initiating the GRANDPA State

A validator needs to initiate its state and sync it with other validators, to be able to participate coherently in the voting process. In particular, considering that voting is happening in different rounds and each round of voting is assigned a unique sequential round number r_v, it needs to determine and set its round counter r in accordance with the current voting round r_n, which is currently undergoing in the network. Algorithm 6.8

Algorithm 6.8. Initiate-Grandpa(r_{last}: last round number or 0 if the voter starting a new authority set, , B_{last}: the last block which has been finalized on the chain)

```
Last-Finalized-Block ← $B_{\text{last}}$
if $r_{\text{last}} = 0$
    Best-Final-Candidate(0) ← $B_{\text{last}}$
    Grandpa-GHOST(0) ← $B_{\text{last}}$
$r_n \leftarrow r_{\text{last}+1}$
Play-Grandpa-round($r_n$)
```

6.3.3.1. Voter Set Changes

Voter set changes are signaled 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 higher number than the block at which the change is supposed to happen. The new authority set should reinitiate GRANDPA protocol by exectutig Algorithm 6.8.

6.3.4. 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)

```
$t_{r,v} \leftarrow$ Current local time
primary ← Derive-Primary($r$)
if $v =$ primary
    Broadcast($M_v^{r-1,\text{Fin}}$(Best-Final-Candidate($r$-1)))
    if Best-Final-Candidate($r-1$) $\geqslant$ Last-Finalized-Block
        Broadcast($M_v^{r-1,\text{Prim}}$(Best-Final-Candidate($r$-1)))
Receive-Messages(until Time $\geqslant t_{r,v} + 2 \times T$ or $r$ is completable)
$L \leftarrow$ Best-Final-Candidate($r$-1)
$N \leftarrow$ Best-PreVote-Candidate($r$)
Broadcast($M_v^{r,\text{pv}}(N)$)
Receive-Messages(until $B_v^{r,\text{pv}} \geqslant L$ and (Time $\geqslant t_{r,v} + 4 \times T$ or $r$ is completable))
```

```
Broadcast($M_v^{r,\text{pc}}(B_v^{r,\text{pv}})$)
Attempt-To-Finalize-Round($r$)
Receive-Messages(until $r$ is completable and Finalizable($r-1$)
      and Last-Finalized-Block$\geqslant$Best-Final-Candidate($r$-1))
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 6.33.
- 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)

```
return $r \bmod |\mathbb{V}|$
```

$\mathbb{V}$ is the GRANDPA voter set as defined in Definition 6.22.

Algorithm 6.11. Best-Final-Candidate(r)

```
$B_v^{r,\text{pv}} \leftarrow$ GRANDPA-GHOST($r$)
$\mathcal{C} \leftarrow \{B' | B' \leqslant B_v^{r,\text{pv}} : \#V_{\text{obv}(v),\text{pot}}^{r,\text{pc}}(B') > 2/3|\mathbb{V}|\}$
if $\mathcal{C} = \phi$
    $E \leftarrow$ GRANDPA-GHOST($r$)
return $E \in \mathcal{C} : H_n(E) = \max \{H_n(B') : B' \in \mathcal{C}\}$
```

$\#V_{\text{obv}(v),\text{pot}}^{r,\text{stage}}$ is defined in Definition 6.31.

Algorithm 6.12. [GRANDPA-GHOST][is highest vot is equal ghost?]

```
return $B' : H_n(B') = \max \{H_n(B') : B' > L\}$
```

Algorithm 6.13. Best-PreVote-Candidate(r: voting round to cast the pre-vote in)[imporve/fix me]

```
$L \leftarrow$ Best-Final-Candidate($r-1$)
$B_v^{r,\text{pv}} \leftarrow$ GRANDPA-GHOST($r$)
if Received($M_{v_{\text{primary}}}^{r,\text{prim}}(B)$) and $B_v^{r,\text{pv}} \geqslant B > L$
    $N \leftarrow B$
else
    $N \leftarrow B_v^{r,\text{pv}}$
```

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 6.33.

Algorithm 6.15. Attempt-To-Finalize-Round(r)

```
1: $L \leftarrow$ Last-Finalized-Block
2: $E \leftarrow$ Best-Final-Candidate$(r)$
3: if $E \geqslant L$ and $V^{r,\text{pc}}_{\text{obs}(v)}(E) > 2/3|\mathbb{V}|$
4:     Last-Finalized-Block$\leftarrow E$
5:     if $M_v^{r,\text{Fin}}(E) \notin$ Received-Messages
6:         Broadcast$(M_v^{r,\text{Fin}}(E))$
7:         return
8: 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 6.41 serves to demonstrate such a situation:

Example 6.41. 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 prevotes for B_2 and at least one prevote for B_1 which means that GRANDPA-GHOST$(1) = B_2$.

Subsequently potentially honest voters who could claim not seeing all the prevotes for B_2 but receiving the prevotes for B_1 would precommit to B_1. In this way, we receive 66 precommits for B_1 and 1 precommit 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 precommits are going to be casted 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 6.42. *A Polkadot relay chain node n should consider block B as* ***finalized*** *if any of the following criteria holds for $B' \geqslant B$:*

- $V_{\text{obs}(n)}^{r,\text{pc}}(B') > 2/3|\mathbb{V}_{B'}|$.
- *it receives a $M_v^{r,\text{Fin}}(B')$ message in which $J^r(B)$ justifies the finalization (according to Definition 6.37).*
- *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 the node n has invoked Algorithm 6.9 if n is a GRANDPA voter.

Note that all Polkadot relay chain nodes are supposed to listen to GRANDPA finalizing messages regardless if they are GRANDPA voters.

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 which it is missing in form of blocks. Each finalized block comes with the Justification of its finalization as defined in Definition 6.37 [Verify: you can't trust your neigbour for their set, you need to get it from the chain]. Through this process, they can synchronize the authority list currently performing the finalization process.

6.4.1.1. Sending 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 6.39 to the reporting peer in order to catch-up to the more advanced finalized round, provided that the following criteria holds:

- 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 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 6.39)
)

if $M_{i,v}^{\text{Cat}-q}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}} \neq \text{id}_{\mathbb{V}}$
    error "Catching up on different set"
if $i \notin \mathbb{P}$
    error "Requesting catching up from a non-peer"
if $r >$ Last-Completed-Round
    error "Catching up on a round in the future"
Send($i$,$M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)$)
```

In which:

- $\mathrm{id}_{\mathbb{V}}$ is the voter set id 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]
- $M_{v,i}^{\mathrm{Cat}-s}(\mathrm{id}_{\mathbb{V}}, r)$ is the catch-up response defined in Definition 6.40.

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 finality of the Relay chain according to Algorithm 6.17.

Algorithm 6.17. Process-Catchup-Response(
$M_{v,i}^{\mathrm{Cat}-s}(\mathrm{id}_{\mathbb{V}}, r)$: the catch-up response received from node v (See Definition 6.40)
)

```
$M_{v,i}^{\mathrm{Cat}-s}(\mathrm{id}_{\mathbb{V}}, r).\mathrm{id}_{\mathbb{V}}, r, J^{r,\mathrm{pv}}(B), J^{r,\mathrm{pc}}(B), H_h(B'), H_i(B') \leftarrow \mathrm{Dec}_{\mathrm{SC}}(M_{v,i}^{\mathrm{Cat}-s}(\mathrm{id}_{\mathbb{V}}, r))$
if $M_{v,i}^{\mathrm{Cat}-s}(\mathrm{id}_{\mathbb{V}}, r).\mathrm{id}_{\mathbb{V}} \neq \mathrm{id}_{\mathbb{V}}$
    error "Catching up on different set"
if $r \leqslant$ Leading-Round
    error "Catching up in to the past"
if $J^{r,\mathrm{pv}}(B)$ is not valid
    error "Invalid pre-vote justification"
if $J^{r,\mathrm{pc}}(B)$ is not valid
    error "Invalid pre-commit justification"
$G \leftarrow$ GRANDPA-GHOST$(J^{r,\mathrm{pv}}(B))$
if $G = \phi$
    error "GHOST-less Catch-up"
if $r$ is not completable
    error "Catch-up round is not completable"
if $J^{r,\mathrm{pc}}(B)$ justifies $B'$ finalization
    error "Unjustified Catch-up target finalization"
Last-Completed-Round $\leftarrow r$
if $i \in \mathbb{V}$
    Play-Grandpa-round$(r + 1)$
```

□

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 $\langle b|w\rangle$itness 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.*

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

[@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:*

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

where each value represents:

- *B: the parachain block itself.*
- *$head(B_p)$: 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.*

$$v_B^{GVS} := (Max_{size}^R, Max_{size}^{head}, H_i(B_{chain}^{relay}))$$

where each value represents:

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

- $H_i(B_{chain}^{relay})$: *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:*

$$v_B^{LVP} := (h\,e\,a\,d(B_p), U\,I\,N\,T\,128, B\,l\,a\,k\,e\,2\,b(R_\rho), R_\rho^{up})$$

where each value represents:

- $h\,e\,a\,d(B_p)$: *the parent head data (Definition 7.12) of block B.*
- $U\,I\,N\,T\,128$: *the balance of the parachain at the moment of validation.*
- $B\,l\,a\,k\,e\,2\,b(R_\rho)$: *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:*

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

where each value represents:

- $h\,e\,a\,d(B)$: *the new head data (Definition 7.12) of block B.*
- $O\,p\,t\,i\,o\,n(P_\rho^B)$: *a varying data (Definition 7.1) containing an update to the validation code that should be scheduled in the relay chain.*
- $M\,s\,g$: *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, $E\,n\,c_{SC}(b_0, ..\,b_n)$, contains the raw bytes of the message which varies in size.

- $U\,I\,N\,T\,32$: *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:*

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

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].

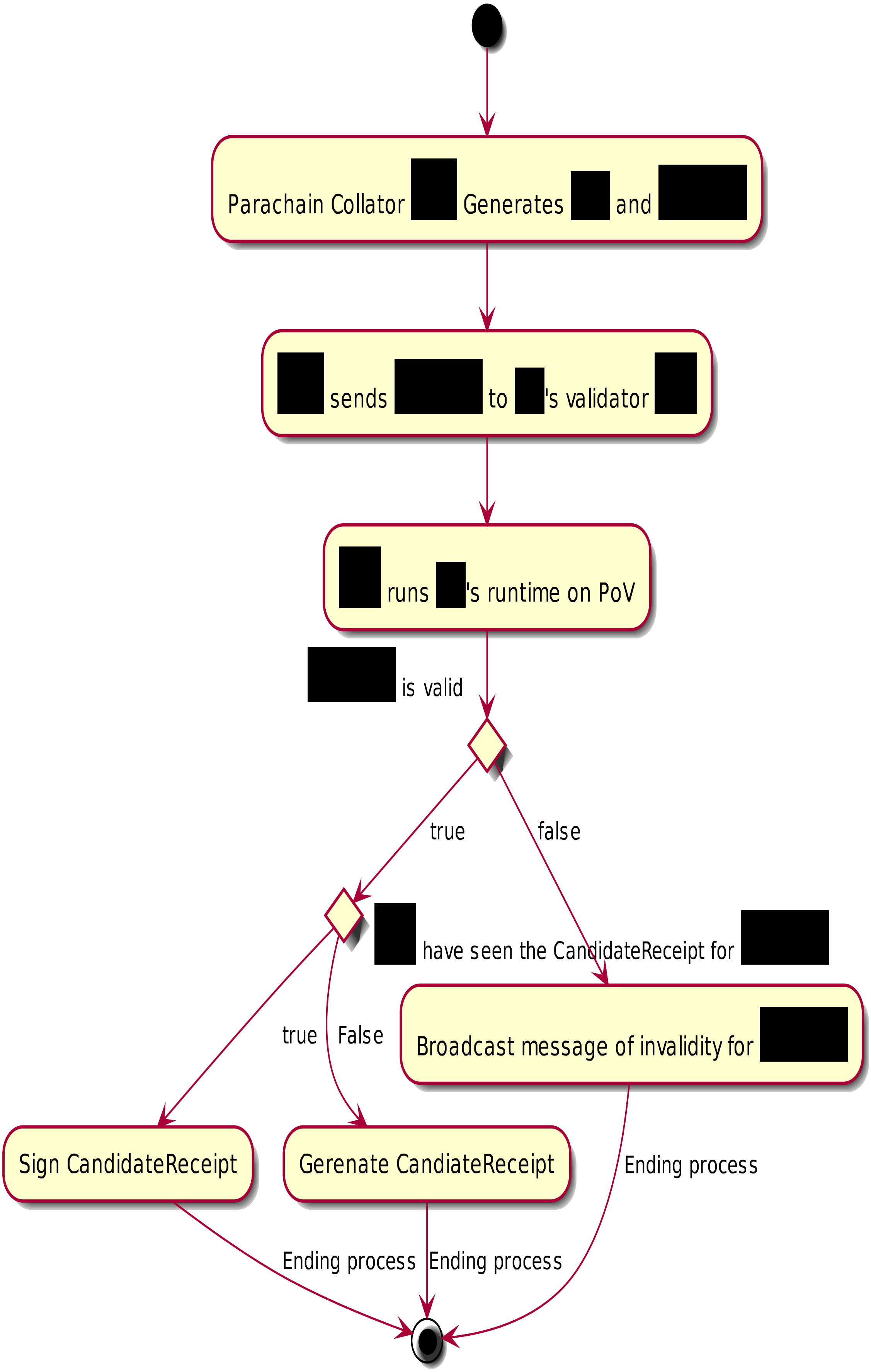
Parachain Collator [redacted] Generates [redacted] and [redacted]
[redacted] sends [redacted] to [redacted]'s validator [redacted]
[redacted] runs [redacted]'s runtime on PoV
[redacted] is valid
true
false
[redacted] have seen the CandidateReceipt for [redacted]
true
False
Broadcast message of invalidity for [redacted]
Sign CandidateReceipt
Gerenate CandiateReceipt
Ending process
Ending process
Ending process

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**, $C_{coll}(PoV_B)$, 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:*

$$C_{coll}(PoV_B) := (id_p, h_b(B_{\substack{relay\\parent}}), id_C, Sig_{SR25519}^{Collator}, head(B), h_b(PoV_B))$$

where each value represents:

- *id_p: the Parachain Id this candidate is for.*
- *$h_b(B_{\substack{relay\\parent}})$: 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].*
- *$Sig_{SR25519}^{Collator}$: the signature on the 256-bit Blake2 hash of the block data by the collator.*
- *$head(B)$: the head data (Definition 7.12) of block B.*
- *$h_b(PoV_B)$: the 32-byte Blake2 hash of the PoV block.*

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

$$head(B) := (H_i(B), H_p(B), H_r(B))$$

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. ⟨caption*||PrimaryValidation⟩ Require: $B$, $\pi_B$, relay chain parent block $B_{parent}^{relay}$
    Retrieve $v_B$ from the relay chain state at $B_{parent}^{relay}$
    Run Algorithm 7.2 using $B, \pi_B, v_B$
?
```

```
Algorithm 7.2. ⟨caption*||ValidateBlock⟩ Require: $B, \pi_B, v_B$
    retrieve the runtime code $R_\rho$ that is specified by $v_B$ from the relay chain state.
    check that the initial state root in $\pi_B$ is the one claimed in $v_B$
    Execute $R_\rho$ on $B$ using $\pi_B$ to simulate the state.
    If the execution fails, return fail.
    Else return success, the new header data $h_B$ and the outgoing messages $M$. [@fabio: same as hea
?
```

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**, $C_{receipt}(PoV_B)$, 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:

$$C_{receipt}(PoV_B) := (id_p, h_b(B_{\substack{relay\\parent}}), head(B), id_C, Sig_{SR25519}^{Collator}, h_b(PoV_B), Blake2b(CC(PoV_B)))$$

where each value represents:

- id_p*: the Parachain Id this candidate is for.*
- $h_b(B_{\substack{relay\\parent}})$*: the hash of the relay chain block that this candidate should be executed in the context of.*
- $head(B)$*: 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].*
- $Sig_{SR25519}^{Collator}$*: the signature on the 256-bit Blake2 hash of the block data by the collator.*
- $h_b(PoV_B)$*: the hash of the PoV block.*
- $Blake2b(CC(PoV_B))$*: The hash of the commitments made as a result of validation, as defined in Definition 7.14.*

Definition 7.14. ***Candidate commitments**, $CC(PoV_B)$, 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:*

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

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, $Enc_{SC}(\mathrm{b}_0, .. \mathrm{b}_n)$, 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?].*
- $Option(R_\rho)$*: 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:*

$$(h_b(B_{\substack{relay\\parent}}), h_b(C_{coll}(PoV_B)), PoV_B)$$

where $h_b(B_{\substack{relay\\parent}})$ is the block hash of the relay chain being referred to and $h_b(C_{coll}(PoV_B))$ 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:*

$$(Stmt, id_{\mathbb{V}}, Sig_{SR25519}^{Valdator})$$

where $Sig_{SR25519}^{Validator}$ is the signature of the validator and $id_{\mathbb{V}}$ 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:

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

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. ⟨caption*|PrimaryValidationAnnouncement|PrimaryValidationAnnouncement⟩ **Require:** P

```
Init $Stmt$;
if ValidateBlock($PoV_B$) is valid then
    $Stmt \leftarrow$ SetValid($PoV_B$)
else
    $Stmt \leftarrow$ SetInvalid($PoV_B$)
    BlacklistCollatorOf($PoV_B$)
end if
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. ⟨caption*||ConfirmCandidateReceipt⟩ **Require:** $Stmt_{peer}$

```
Init $Stmt$;
$PoV_B \leftarrow$ Retrieve($Stmt_{peer}$)
if ValidateBlock($PoV_B$) is valid then
    if AlreadySeconded($B_{chain}^{relay}$) then
        $Stmt \leftarrow$ SetValid($PoV_B$)
    else
        $Stmt \leftarrow$ SetSeconded($PoV_B$)
    end if
else
    $Stmt \leftarrow$ SetInvalid($PoV_B$)
    AnnounceMisbehaviorOf($PoV_B$)
end if
Propagate($Stmt$)
```

?

- $Stmt_{peer}$: 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. ⟨b|P⟩*arachain 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.   ⟨caption*||IncludeParachainProposal($P_\rho^B$)⟩ Require:
                 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.   ⟨caption*||PrimaryValidationDisagreement⟩ Require:
                 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** $\boldsymbol{encode_{k,n}/decoder_{k,n}}$ is defined to be the Reed-Solomon encoder defined in [?].*

Algorithm 7.7.

```
⟨caption*||Erasure-Encode⟩ Require: $\bar{B}$: blob defined in Definition 7.10
Init $Shards \leftarrow$ Make-Shards($\mathcal{V}_\rho, v_B$) ⟨Statex⟩ ⟨Statex⟩// Create a trie from the shards in orde
Init $Trie$
Init $index = 0$
for $shard \in Shards$ do
    Insert($Trie, index$, Blake2($shard$))
    $index = index + 1$
end for
⟨Statex⟩ ⟨Statex⟩// Insert individual chunks into collection (Definition 7.19). Init $Er_B$
Init $index = 0$
for $shard \in Shards$ do
    Init $nodes \leftarrow$ Get-Nodes($Trie, index$)
    Add($Er_B, (shard, index, nodes)$)
    $index = index + 1$
end for
⟨Statex⟩
return $Er_B$
```

?

- Make-Shards(..): return shards for each validator as described in algorithm 7.8. Return value is defined as $(\mathbb{S}_0, ..., \mathbb{S}_n)$ where $\mathbb{S} := (b_0, ..., b_n)$
- 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.

```
⟨caption*||Make-Shards⟩ Require: $\mathcal{V}_\rho, v_B$ ⟨Statex⟩// Calculate the required values for Reed-Sol
Init $Shard_{data} = \frac{(|\mathcal{V}_\rho| - 1)}{3} + 1$
Init $Shard_{parity} = |\mathcal{V}_\rho| - \frac{(|\mathcal{V}_\rho| - 1)}{3} - 1$
⟨vcenter|    Init $base_{len} = \begin{cases} 0 & if|\mathcal{V}_\rho| \bmod Shard_{data} = 0 \\ 1 & if|\mathcal{V}_\rho| \bmod Shard_{data} \neq 0 \end{cases}$
Init $Shard_{len} = base_{len} + (base_{len} \bmod 2)$ ⟨Statex⟩ ⟨Statex⟩// Prepare shards, each padded wit
Init $Shards$
for $n \in (Shard_{data} + Shard_{partiy})$ do
    Add($Shards, (0_0, ..0_{Shard_{len}})$)
end for
⟨Statex⟩ ⟨Statex⟩// Copy shards of $v_b$ into each shard.
for $(chunk, shard) \in$ (Take($Enc_{SC}(v_B), Shard_{len}$), $Shards$) do
    Init $len \leftarrow$ Min($Shard_{len}, |chunk|$)
    $shard \leftarrow$ Copy-From($chunk, len$)
end for
⟨Statex⟩ ⟨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:*

$$Er_B := (e_1, ..., e_n)$$

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

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

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. *⟨b|P⟩oV erasure chunk message $M_{PoV_{\bar{B}}}(i)$ 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. *⟨b|A⟩vailability vote message M_{PoV}^{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:

Algorithm 7.9. ⟨caption*||Relay chain's signature processing⟩ The relay chain stores the last vote from each valic
For each block within the last t blocks where t is some timeout period, the relay chain computes
The relay chain initialises a vector of counts with one entry for each parachain to zero. After exec

1. The relay chain computes b_v and $b\,m_n$ where n is the block number of the validator's last v
2. For each bit in b_v and $b\,m_n$

- add the ith bit to the ith count.

For each count that is >2/3 of the number of validators, the relay chain sets the candidates statu
The relay chain acts on available candidates and discards unavailable ones, and then clears the re

?

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 ⟨b|a⟩vailable, 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 ⟨b|u⟩navailabile 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. ⟨caption*|VRF-for-Approval|VRF-for-Approval(B, z, s_k)⟩

```
Require: $B$: the block to be approved
         $z$: randomness for approval assignment
         $s_k$: session secret key of validator planning to participate in approval
$(\pi, d) \leftarrow VRF(H_h(B), sk(z))$

return $(\pi, d)$
```

?

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. ⟨caption*||OneShotAssignment⟩ **Require:**

```
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

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

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. ⟨caption*||OneShotAssignment⟩ **Require:**

```
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. ⟨caption*||EquivocatedAssignment⟩ **Require:** 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. ⟨b|P⟩*oV 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. ⟨b|E⟩*rasuree-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. ⟨b|E⟩*rasuree-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 $2\,f+1$ 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. ⟨caption*|Reconstruct-PoV-Erasure|RECONSTRUCT-POV-ERASURE(S_{Er_B})⟩ **Require:** $S_{Er_B} := (e$

```
$\bar{B} \rightarrow$ ERASURE-DECODER($e_{j_1}, \cdots, e_{j_k}$)
if ERASURE-DECODER failed then
ANNOUNCE-FAILURE

return
end if
$Er_B \rightarrow$ ERASURE-ENCODER($\bar{B}$)
if VERIFY-MERKLE-PROOF($S_{Er_B}$, $Er_B$) failed then
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. ⟨caption*||REVALIDATINGRECONSTRUCTEDPOV⟩ **Require:**

```
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. ⟨b|S⟩*econdary 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. ⟨caption*||VERIFYAPPROVALATTESTATION⟩ **Require:**

```
TBS
```

?

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

DEFINITION 7.30. ⟨b|A⟩*pproval 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. ⟨b|C⟩*ollator Invalidity Transaction TBS*

DEFINITION 7.32. ⟨b|C⟩*ollator unavailability Transaction*

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.

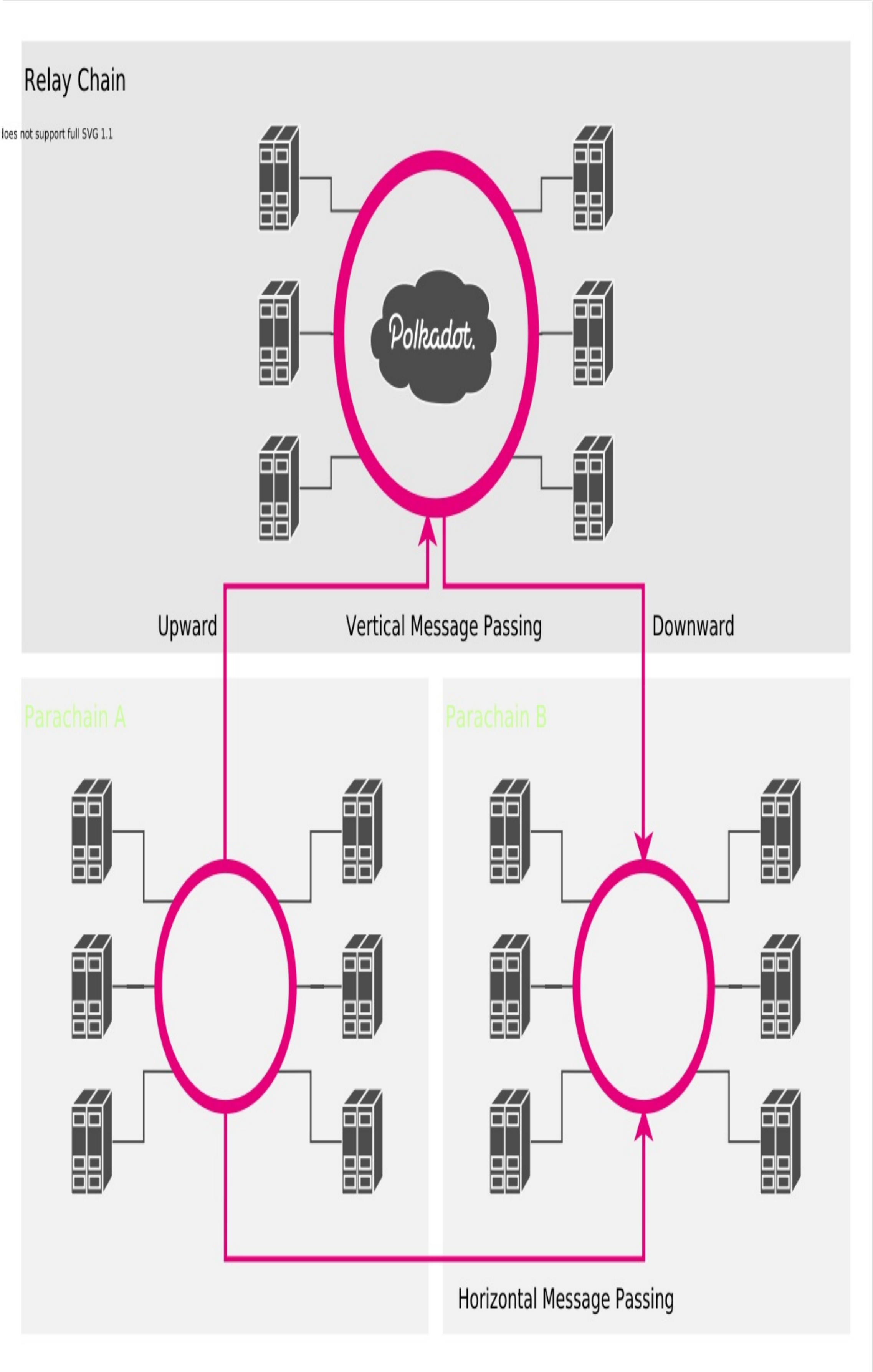

Figure 8.1. Parachain Message Passing Overview

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.

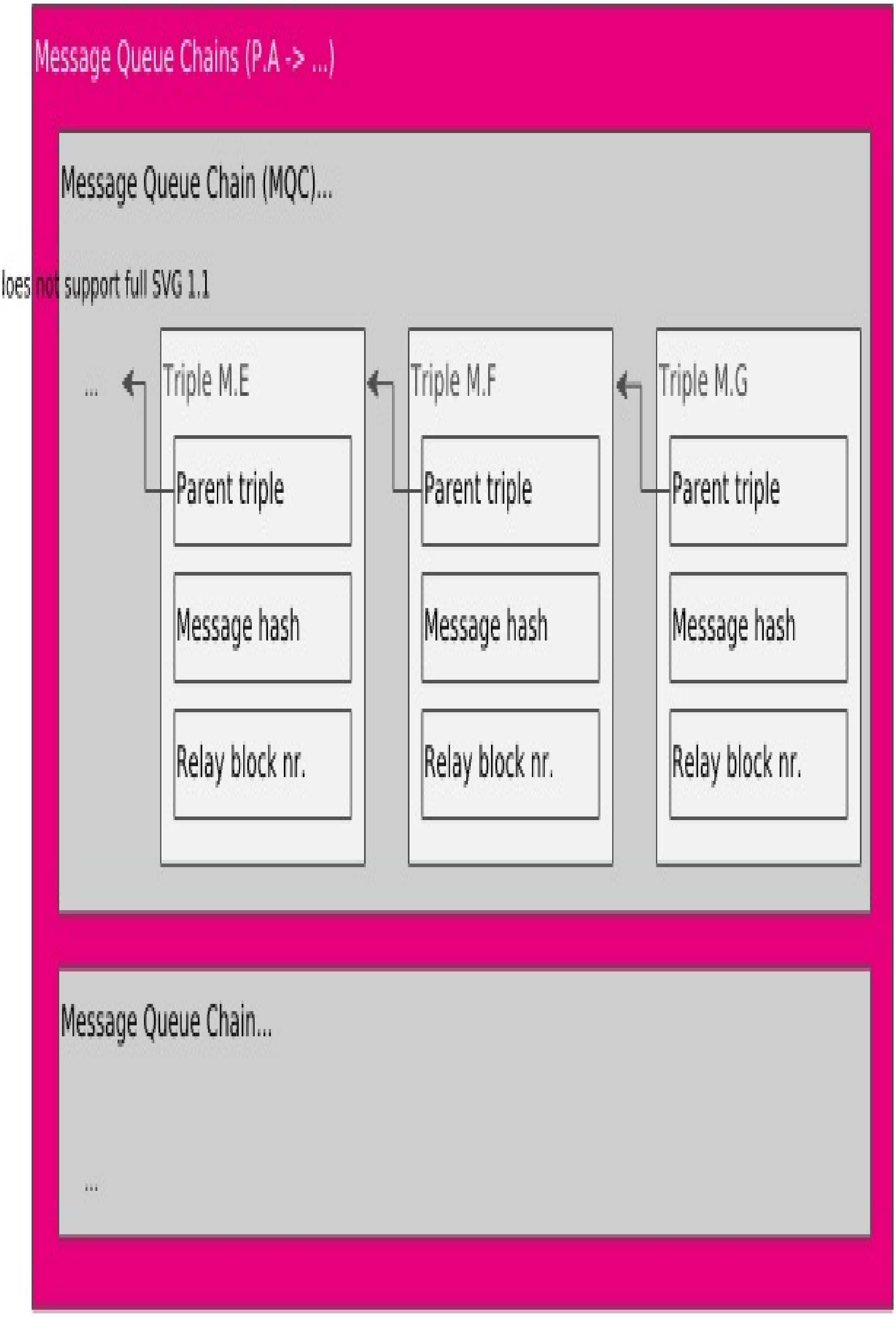

Figure 8.2. MQC Overview

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 array 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.

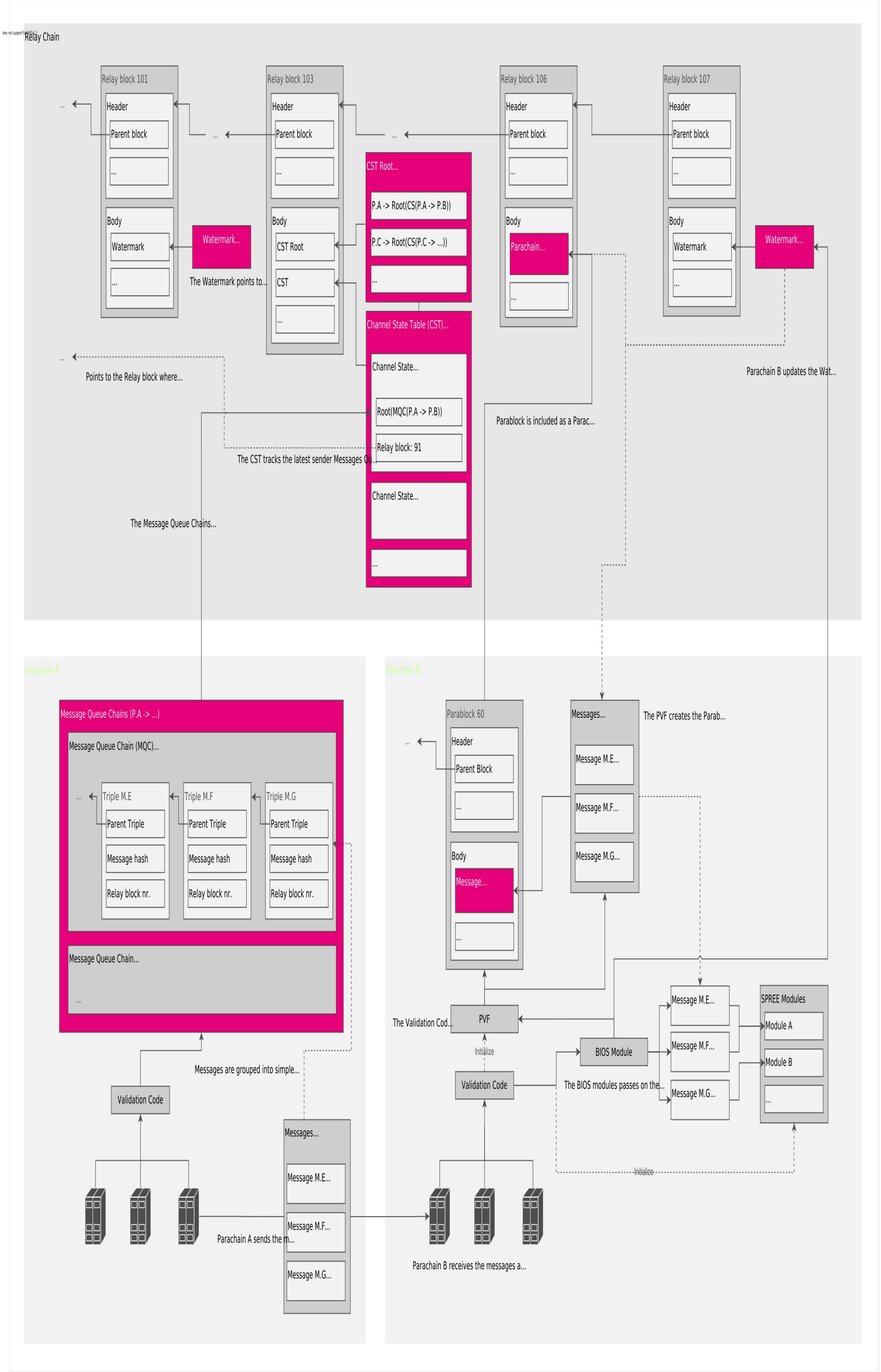

Figure 8.3. Parachain XCMP Overview

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 SHA-3 competition.

Polkadot is using Blake2b variant which is optimized for 64bit platforms. Unless otherwise specified, Blake2b hash function with 256bit 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 [SA15].

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 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 [Bur19]*
ED25519	*The standard ED25519 signature complying with [JL17]*
secp256k1	*Only for outgoing transfer transactions*

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

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 an 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 other keys. It is a key that will be in the 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 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 Tec19. 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

Session keys should be changed regularly. As such, new session keys need to be certified by a controller key before putting in use. The controller only needs to create a certificate by signing a session public key and broadcastg 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*

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

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

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

where $\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}$ is defined in Definition B.12.

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**, is defined as:*

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

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. *We define a **varying data** type 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 $(A_{\mathrm{Type}}, A_{\mathrm{Value}})$ where $A_{\mathrm{Type}} = T_i$ for some $T_i \in \mathcal{T}$ and A_{Value} is its value of type T_i, which can be empty. We define $\mathrm{idx}(T_i) = i - 1$, 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.4. *The **Option** type is a varying data type of $\{\mathrm{None}, T_2\}$ 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 $\mathrm{idx}(T_{\mathrm{None}}) = 0$, 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 $\mathrm{idx}(T_2) = 1$ implies that the data is available.*

Definition B.5. *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.6. *Scale coded for value* $\boldsymbol{A = (A_{\mathrm{Type}}, A_{\mathrm{Value}})}$ ***of varying data type*** $\mathcal{T} = \{T_1, \ldots, T_n\}$

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

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

In particular, for the optional type defined in Definition B.3, we have:

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

SCALE codec does not encode the correspondence between the value of Idx defined in Definition B.6 and the data type it represents; the decoder needs prior knowledge of such correspondence to decode the data.

DEFINITION B.7. *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 defined as:*

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

where $\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}$ *is defined in Definition B.12.*

DEFINITION B.8. *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):*

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

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

DEFINITION B.9. *The* ***SCALE codec*** *for* ***boolean value*** b *defined as a byte as follows:*

$$\begin{aligned} \mathrm{Enc}_{\mathrm{SC}}\colon\ & \{\mathrm{False}, \mathrm{True}\} \rightarrow \mathbb{B}_1 \\ & b \rightarrow \begin{cases} 0 & b = \mathrm{False} \\ 1 & b = \mathrm{True} \end{cases} \end{aligned}$$

DEFINITION B.10. *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.11. *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.12. ***SCALE Length Encoding,*** $\mathbf{Enc_{SC}^{Len}}$ *also known as compact encoding of a non-negative integer number* n *is defined as follows:*

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

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

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

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

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

such that:

$$k_1^7 \ldots k_1^3\, k_1^2 \colon= m - 4$$

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:

Definition B.13. *Suppose that* $\mathrm{PK} = (k_1, \ldots, k_n)$ *is a sequence of nibbles, then* $\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK}) :=$

$$\left\{\begin{array}{lcl} \mathrm{Nibbles}_4 & \rightarrow & \mathbb{B} \\ \mathrm{PK} = (k_1, \ldots, k_n) & \mapsto & \left\{\begin{array}{ll} (16k_1 + k_2, \ldots, 16k_{2i-1} + k_{2i}) & n = 2\,i \\ (k_1, 16k_2 + k_3, \ldots, 16k_{2i} + k_{2i+1}) & n = 2\,i + 1 \end{array}\right. \end{array}\right.$$

□

Appendix C

Genesis State Specification

The genesis state represents the intial state of Polkadot state storage as a set of key-value pairs, which can be retrieved from [Fou20]. While each of those key/value pairs offer important identifyable information which can be used by the Runtime, from the Polkadot Host points of view, it is a set of arbitrary key-value pair data as it is chain and network dependent. Except for the `:code` described in Section 3.1.1 which needs to be identified by the Polkadot Host to load its content as the Runtime. The other keys and values are unspecifed and its usage depends on the chain respectively its corresponding Runtime. The data should be inserted into the state storage with the `set_storage` Host API, as defined in Section F.1.1.

As such, Polkadot does not defined a formal genesis block. Nonetheless for the complatibilty 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 2.12 after inserting the genesis state in it.*
extrinsics_root	*0*
digest	*0*

Table C.1. *Genesis header values*

□

Appendix D
Network Messages

In this section, we will specify various types of messages which the Polkadot Host receives from the network. Furthermore, we also explain the appropriate responses to those messages.

Definition D.1. *A **network message** is a byte array, **M** of length $\|M\|$ such that:*

$$\begin{array}{rl} M_1 & \text{Message Type Indicator} \\ M_2 \ldots M_{\|M\|} & \text{Enc}_{\text{SC}}(\text{MessageBody}) \end{array}$$

The body of each message consists of different components based on its type. The different possible message types are listed below in Table D.1. We describe the sub-components of each message type individually in Section D.1.

M_1	Message Type	Description
0	Status	D.1.1
1	Block Request	D.1.2
2	Block Response	D.1.3
3	Block Announce	D.1.4
4	Transactions	D.1.5
5	Consensus	D.1.6
6	Remote Call Request	
7	Remote Call Response	
8	Remote Read Request	
9	Remote Read Response	
10	Remote Header Request	
11	Remote Header Response	
12	Remote Changes Request	
13	Remote Changes Response	
14	FinalityProofRequest	
15	FinalityProofResponse	
255	Chain Specific	

Table D.1. List of possible network message types.

D.1. Detailed Message Structure

This section disucsses the detailed structure of each network message.

D.1.1. Status Message

A *Status* Message represented by M_S is sent after a connection with a neighbouring node is established and has the following structure:

$$M_S := \text{Enc}_{\text{SC}}(v, r, N_B, \text{Hash}_B, \text{Hash}_G, C_S)$$

Where:

v:	Protocol version	32 bit integer
$v_{\min}$:	Minimum supported version	32 bit integer
r:	Roles	1 byte
N_B:	Best Block Number	64 bit integer
Hash_B	Best block Hash	$\mathbb{B}_{32}$
Hash_G	Genesis Hash	$\mathbb{B}_{32}$
C_S	Chain Status	Byte array

In which, Role is a bitmap value whose bits represent different roles for the sender node as specified in Table D.2:

Value	Binary representation	Role
0	00000000	No network
1	00000001	Full node, does not participate in consensus
2	00000010	Light client node
4	00000100	Act as an authority

Table D.2. Node role representation in the status message.

D.1.2. Block Request Message

A Block request message, represented by M_{BR}, is sent to request block data for a range of blocks from a peer and has the following structure:

$$M_{\mathrm{BR}} := \mathrm{Enc}_{\mathrm{SC}}(\mathrm{id}, A_B, S_B, \mathrm{Hash}_E, d, \mathrm{Max})$$

where:

id:	Unique request id	32 bit integer
A_B:	Requested data	1 byte
S_B:	Starting Block	Varying $\{\mathbb{B}_{32}, 64\mathrm{bit\,integer}\}$
Hash_E	End block Hash	$\mathbb{B}_{32}$ optional type
d	Block sequence direction	1 byte
Max	Maximum number of blocks to return	32 bit integer optional type

in which

- A_B, the requested data, is a bitmap value, whose bits represent the part of the block data requested, as explained in Table D.3:

Value	Binary representation	Requested Attribute
1	00000001	Block header
2	00000010	Block Body
4	00000100	Receipt
8	00001000	Message queue
16	00010000	Justification

Table D.3. Bit values for block attribute A_B, to indicate the requested parts of the data.

- S_B is SCALE encoded varying data type (see Definition B.6) of either $\mathbb{B}_{32}$ representing the block hash, H_B, or 64bit integer representing the block number of the starting block of the requested range of blocks.
- Hash_E is optionally the block hash of the last block in the range.
- d is a flag; it defines the direction on the block chain where the block range should be considered (starting with the starting block), as follows

$$d = \begin{cases} 0 & \mathrm{child\,to\,parent\,direction} \\ 1 & \mathrm{parent\,to\,child\,direction} \end{cases}$$

Optional data type is defined in Definition B.3.

D.1.3. Block Response Message

A *block response message* represented by M_{BS} is sent in a response to a requested block message (see Section D.1.2). It has the following structure:

$$M_{\mathrm{BS}} := \mathrm{Enc}_{\mathrm{SC}}(\mathrm{id}, D)$$

where:

id:	Unique id of the requested response was made for	32 bit integer
D:	Block data for the requested sequence of Block	Array of block data

In which block data is defined in Definition D.2.

Definition D.2. ***Block Data*** *is defined as the follownig tuple:[Block Data definition should go to block format section]*

$$(H_B, \text{Header}_B, \text{Body}, \text{Receipt}, \text{MessageQueue}, \text{Justification})$$

Whose elements, with the exception of H_B, are all of the following *optional type* (see Definition B.3) and are defined as follows:

H_B:	Block header hash	$\mathbb{B}_{32}$
Header_B:	Block header	5-tuple (Definition 3.6)
Body	Array of extrinsics	Array of Byte arrays (Section 3.2)
Receipt	Block Receipt	Byte array
Message Queue	Block message queue	Byte array
Justification	Block Justification	Byte array

D.1.4. Block Announce Message

A *block announce message* represented by M_{BA} is sent when a node becomes aware of a new complete block on the network and has the following structure:

$$M_{\text{BA}} := \text{Enc}_{\text{SC}}(\text{Header}_B)$$

Where:

Header_B:	Header of new block B	5-tuple header (Definition 3.6)

D.1.5. Transactions

The transactions Message 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 sepearate extrinsic. The Polkadot Host is indifferent about the content of an extrinsic and treats it as a blob of data.

D.1.6. Consensus Message

A *consensus message* represented by M_C is sent to communicate messages related to consensus process:

$$M_C := \text{Enc}_{\text{SC}}(E_{\text{id}}, D)$$

Where:

E_{id}:	The consensus engine unique identifier	$\mathbb{B}_4$
D	Consensus message payload	$\mathbb{B}$

in which

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

The network agent should hand over D to approperiate consensus engine which identified by E_{id}.

D.1.7. Neighbor Packet

[Place holder for speccing Neighbor Packet message]

□

Appendix E

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 E.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 E.2. *The **Runtime pointer-size** type is an `i64` integer, representing two consecutive `i32` integers in which the least significant one indicates the pointer to the memory buffer. The most significant one provides the size of the buffer. This pointer is the primary way to exchange data of arbitrary sizes between the Runtime and the Polkadot Host.*

Definition E.3. ***Lexicographic ordering** refers to the ascending ordering of bytes or byte arrays, such as:*

$$[0,0,2] < [0,1,1] < [0,2,0] < [1] < [1,1,0] < [2] < [...]$$

The functions are specified in each subsequent subsection for each category of those functions.

E.1. Storage

Interface for accessing the storage from within the runtime.

E.1.1. ext_storage_set

Sets the value under a given key into storage.

E.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 E.2 containing the key.
- `value`: a pointer-size as defined in Definition E.2 containing the value.

E.1.2. ext_storage_get

Retrieves the value associated with the given key from storage.

E.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 E.2 containing the key.

- `result`: a pointer-size as defined in Definition E.2 returning the SCALE encoded `Option` as defined in Definition B.4 containing the value.

E.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.

E.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 E.2 containing the key.
- `value_out`: a pointer-size as defined in Definition E.2 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 u32 integer containing the offset beyond the value should be read from.
- `result`: a pointer-size (Definition E.2) pointing to a SCALE encoded `Option` (Definition B.4) containing an unsinged 32-bit interger representing the number of bytes left at supplied `offset`. Returns `None` if the entry does not exists.

E.1.4. `ext_storage_clear`

Clears the storage of the given key and its value.

E.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 E.2 containing the key.

E.1.5. `ext_storage_exists`

Checks whether the given key exists in storage.

E.1.5.1. Version 1 - Prototype

```
(func $ext_storage_exists_version_1
   (param $key_data i64) (return i32))
```

Arguments:

- `key`: a pointer-size as defined in Definition E.2 containing the key.
- `return`: an i32 integer value equal to `1` if the key exists or a value equal to `0` if otherwise.

E.1.6. `ext_storage_clear_prefix`

Clear the storage of each key/value pair where the key starts with the given prefix.

E.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 E.2 containing the prefix.

E.1.7. `ext_storage_append`

Append the SCALE encoded value to the SCALE encoded storage item at the given key. This function loads the storage item with the given key, assumes that the existing storage item is a SCALE encoded byte array and that the given value is a SCALE encoded item (type) of that array. It then adds that given value to the end of that byte array.

For improved performance, this function does not decode the entire SCALE encoded byte array. Instead, it simply appends the value to the storage item and adjusts the length prefix $\text{Enc}_{\text{SC}}^{\text{Len}}$.

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.

E.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 E.2 containing the key.
- `value`: a pointer-size as defined in Definition E.2 containing the value to be appended.

E.1.8. `ext_storage_root`

Compute the storage root.

E.1.8.1. Version 1 - Prototype

```
(func $ext_storage_root_version_1
   (return i32))
```

Arguments:

- `return`: a regular pointer to the buffer containing the 256-bit Blake2 storage root.

E.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.

E.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 E.2 indicating the SCALE encoded block hash.
- `return`: a regular pointer to the buffer containing the 256-bit Blake2 changes root.

E.1.10. `ext_storage_next_key`

Get the next key in storage after the given one in lexicographic order (Definition E.3). The key provided to this function may or may not exist in storage.

E.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 E.2 indicating the key.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the next key in lexicographic order.

E.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` (E.1.12) or `ext_storage_commit_transaction` (E.1.13). This is also effective for all values manipulated using the child storage API (E.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.

E.1.11.1. Version 1 - Prototype

```
(func $ext_storage_start_transaction_version_1)
```

Arguments:

- None.

E.1.12. `ext_storage_rollback_transaction`

Rollback the last transaction started by `ext_storage_start_transaction` (E.1.11). Any changes made during that transaction are discarded.

Warning: Panics if there is no open transaction (`ext_storage_start_transaction` (E.1.11) was not called)

E.1.12.1. Version 1 - Prototype

```
(func $ext_storage_rollback_transaction_version_1)
```

Arguments:

- None.

E.1.13. `ext_storage_commit_transaction`

Commit the last transaction started by `ext_storage_start_transaction` (E.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` (E.1.11) was not called)

E.1.13.1. Version 1 - Prototype

```
(func $ext_storage_commit_transaction_version_1)
```

Arguments:

- None.

E.2. Child Storage

Interface for accessing the child storage from within the runtime.

Definition E.4. ***Child storage*** *key is a unprefixed location of the child trie in the main trie.*

E.2.1. `ext_default_child_storage_set`

Sets the value under a given key into the child storage.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `value`: a pointer-size as defined in Definition E.2 indicating the value.

E.2.2. `ext_default_child_storage_get`

Retrieves the value associated with the given key from the child storage.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the value.

E.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.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `value_out`: a pointer-size as defined in Definition E.2 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 u32 integer containing the offset beyond the value should be read from.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the number of bytes written into the **value_out** buffer. Returns `None` if the entry does not exists.

E.2.4. `ext_default_child_storage_clear`

Clears the storage of the given key and its value from the child storage.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.

E.2.5. `ext_default_child_storage_storage_kill`

Clears an entire child storage.

E.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 E.2 indicating the child storage key as defined in Definition E.4.

E.2.6. `ext_default_child_storage_exists`

Checks whether the given key exists in the child storage.

E.2.6.1. Version 1 - Prototype

```
(func $ext_default_child_storage_exists_version_1
     (param $child_storage_key i64) (param $key i64) (return i32))
```

Arguments:

- `child_storage_key`: a pointer-size as defined in Definition E.2 indicating the child storage key as defined in Defintion E.4.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `return`: an i32 integer value equal to `1` if the key exists or a value equal to `0` if otherwise.

E.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.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `prefix`: a pointer-size as defined in Definition E.2 indicating the prefix.

E.2.8. `ext_default_child_storage_root`

Commits all existing operations and computes the resulting child storage root.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded storage root.

E.2.9. `ext_default_child_storage_next_key`

Gets the next key in storage after the given one in lexicographic order (Definition E.3). The key provided to this function may or may not exist in storage.

E.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 E.2 indicating the child storage key as defined in Definition E.4.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the next key in lexicographic order. Returns `None` if the entry cannot be found.

E.3. Crypto

Interfaces for working with crypto related types from within the runtime.

Definition E.5. *Cryptographic keys are saved in their own storages in order to avoid collision with each other. The storages are identified by their 4-byte ASCII **key type ID**. The following known types are available:*

Id	*Description*
babe	*Key type for the Babe module*
gran	*Key type for the Grandpa module*
acco	*Key type for the controlling accounts*
imon	*Key type for the ImOnline module*
audi	*Key type for the AuthorityDiscovery module*

Table E.1. *Table of known key type identifiers*

Definition E.6. ***EcdsaVerifyError*** *is a varying data type as defined in Definition B.3 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 E.2. *Table of error types in ECDSA recovery*

E.3.1. `ext_crypto_ed25519_public_keys`

Returns all `ed25519` public keys for the given key id from the keystore.

E.3.1.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_public_keys_version_1
   (param $key_type_id i64) (return i64))
```

Arguments:

- `key_type_id`: an i32 integer indicating the key type ID as defined in Defintion E.5.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded 256-bit public keys.

E.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.

E.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`: an i32 integer indicating the key type ID as defined in Definition E.5.
- `seed`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the BIP-39 seed which must be valid UTF8.
- `return`: a regular pointer to the buffer containing the 256-bit public key.

E.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.

E.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`: an i32 integer indicating the key type ID as defined in Definition E.5.
- `key`: a regular pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size as defined in Definition E.2 indicating the message that is to be signed.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the signature. This function returns `None` if the public key cannot be found in the key store.

E.3.4. `ext_crypto_ed25519_verify`

Verifies an `ed25519` 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` (E.3.16).

The verification extension is explained more in detail in `ext_crypto_start_batch_verify` (E.3.15).

E.3.4.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_verify_version_1
     (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments:

- `sig`: a regular pointer to the buffer containing the 64-byte signature.
- `msg`: a pointer-size as defined in Definition E.2 indicating the message that is to be verified.
- `key`: a regular pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to `1` if the signature is valid or batched or a value equal to `0` if otherwise.

E.3.5. `ext_crypto_sr25519_public_keys`

Returns all `sr25519` public keys for the given key id from the keystore.

E.3.5.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_public_keys_version_1
   (param $key_type_id i64) (return i64))
```

Arguments:

- `key_type_id`: an i32 integer containg the key type ID as defined in E.5.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded 256-bit public keys.

E.3.6. `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.

E.3.6.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_generate_version_1
     (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments:

- `key_type_id`: an i32 integer containg the key ID as defined in Definition E.5.
- `seed`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the BIP-39 seed which must be valid UTF8.
- `return`: a regular pointer to the buffer containing the 256-bit public key.

E.3.7. `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.

E.3.7.1. 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`: an i32 integer containg the key ID as defined in Definition E.5

- key: a regular pointer to the buffer containing the 256-bit public key.
- msg: a pointer-size as defined in Definition E.2 indicating the message that is to be signed.
- return: a pointer-size as defined in Definition E.2 indicating the SCALE encoded Option as defined in Definition B.4 containing the signature. This function returns None if the public key cannot be found in the key store.

E.3.8. 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 (E.3.16).

The verification extension is explained more in detail in ext_crypto_start_batch_verify (E.3.15).

E.3.8.1. Version 2 - Prototype

```
(func $ext_crypto_sr25519_verify_version_2
     (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments:

- sig: a regular pointer to the buffer containing the 64-byte signature.
- msg: a pointer-size as defined in Definition E.2 indicating the message that is to be verified.
- key: a regular pointer to the buffer containing the 256-bit public key.
- return: a i32 integer value equal to 1 if the signature is valid or a value equal to 0 if otherwise.

E.3.8.2. Version 1 - Prototype

```
(func $ext_crypto_sr25519_verify_version_1
     (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments:

- sig: a regular pointer to the buffer containing the 64-byte signature.
- msg: a pointer-size as defined in Definition E.2 indicating the message that is to be verified.
- key: a regular pointer to the buffer containing the 256-bit public key.
- return: a i32 integer value equal to 1 if the signature is valid or a value equal to 0 if otherwise.

E.3.9. ext_crypto_ecdsa_public_keys

Returns all ecdsa public keys for the given key id from the keystore.

E.3.9.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_public_keys_version_1
   (param $key_type_id i64) (return i64))
```

Arguments:

- key_type_id: an i32 integer containg the key type ID as defined in E.5.

- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded 33-byte compressed public keys.

E.3.10. 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.

E.3.10.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_generate_version_1
    (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments:

- `key_type_id`: an i32 integer containg the key ID as defined in Definition E.5.
- `seed`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the BIP-39 seed which must be valid UTF8.
- `return`: a regular pointer to the buffer containing the 33-byte compressed public key.

E.3.11. 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.

E.3.11.1. 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`: an i32 integer containg the key ID as defined in Definition E.5
- `key`: a regular pointer to the buffer containing the 33-byte compressed public key.
- `msg`: a pointer-size as defined in Definition E.2 indicating the message that is to be signed.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 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.

E.3.12. 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` (E.3.16).

The verification extension is explained more in detail in `ext_crypto_start_batch_verify` (E.3.15).

E.3.12.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments:

- `sig`: a regular 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 E.2 indicating the message that is to be verified.
- `key`: a regular pointer to the buffer containing the 33-byte compressed public key.
- `return`: a i32 integer value equal to `1` if the signature is valid or a value equal to `0` if otherwise.

E.3.13. `ext_crypto_secp256k1_ecdsa_recover`

Verify and recover a `secp256k1` ECDSA signature.

E.3.13.1. Version 1 - Prototype

```
(func $ext_crypto_secp256k1_ecdsa_recover_version_1
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments:

- `sig`: a regular pointer to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a regular pointer to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined in Definition B.5. On success it contains the 64-byte recovered public key or an error type as defined in Definition E.6 on failure.

E.3.14. `ext_crypto_secp256k1_ecdsa_recover_compressed`

Verify and recover a `secp256k1` ECDSA signature.

E.3.14.1. Version 1 - Prototype

```
(func $ext_crypto_secp256k1_ecdsa_recover_compressed_version_1
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments:

- `sig`: a regular pointer to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a regular pointer to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined in Definiton B.5. On success it contains the 33-byte recovered public key in compressed form on success or an error type as defined in Definition E.6 on failure.

E.3.15. `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` (E.3.4), `ext_crypto_sr25519_verify` (E.3.8) or `ext_crypto_ecdsa_verify` (E.3.12). Verification will start immediatly and the Runtime can retrieve the result when calling `ext_crypto_finish_batch_verify` (E.3.16).

E.3.15.1. Version 1 - Prototype

```
(func $ext_crypto_start_batch_verify_version_1)
```

Arguments:

- None.

E.3.16. `ext_crypto_finish_batch_verify`

Finish verifying the batch of signatures since the last call to this function. Blocks until all the signatures are verified. Panics if the verification extension was not registered (`ext_crypto_start_batch_verify` (E.3.15) was not called).

Warning: Panics if no verification extension is registered (`ext_crypto_start_batch_verify` (E.3.15) was not called.)

E.3.16.1. Version 1 - Prototype

```
(func $ext_crypto_finish_batch_verify_version_1
   (return i32))
```

Arguments:

- `return`: an i32 integer value equal to `1` if all the signatures are valid or a value equal to `0` if one or more of the signatures are invalid.

E.4. HASHING

Interface that provides functions for hashing with different algorithms.

E.4.1. `ext_hashing_keccak_256`

Conducts a 256-bit Keccak hash.

E.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 E.2 indicating the data to be hashed.
- `return`: a reglar pointer to the buffer containing the 256-bit hash result.

E.4.2. `ext_hashing_sha2_256`

Conducts a 256-bit Sha2 hash.

E.4.2.1. Version 1 - Prototype

```
(func $ext_hashing_sha2_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the data to be hashed.
- `return`: a regular pointer to the buffer containing the 256-bit hash result.

E.4.3. `ext_hashing_blake2_128`

Conducts a 128-bit Blake2 hash.

E.4.3.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_128_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the data to be hashed.
- `return`: a regular pointer to the buffer containing the 128-bit hash result.

E.4.4. `ext_hashing_blake2_256`

Conducts a 256-bit Blake2 hash.

E.4.4.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the data to be hashed.
- `return`: a regular pointer to the buffer containing the 256-bit hash result.

E.4.5. `ext_hashing_twox_64`

Conducts a 64-bit xxHash hash.

E.4.5.1. Version 1 - Prototype

```
(func $ext_hashing_twox_64_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the data to be hashed.
- `return`: a regular pointer to the buffer containing the 64-bit hash result.

E.4.6. `ext_hashing_twox_128`

Conducts a 128-bit xxHash hash.

E.4.6.1. Version 1 - Prototype

```
(func $ext_hashing_twox_128_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the data to be hashed.
- `return`: a regular pointer to the buffer containing the 128-bit hash result.

E.4.7. `ext_hashing_twox_256`

Conducts a 256-bit xxHash hash.

E.4.7.1. Version 1 - Prototype

```
(func $ext_hashing_twox_256_version_1
     (param $data i64) (return i32))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the data to be hashed.
- `return`: a regular pointer to the buffer containing the 256-bit hash result.

E.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 E.7. ***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 E.8. ***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 E.9. ***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 E.10. ***HTTP error*** *is a varying data type as defined in Definition B.3 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 E.3. *Table of possible HTTP error types*

E.5.1. `ext_offchain_is_validator`

Verifies if the local node is a potential validator. Even if this function returns true, it does not mean that any keys are configured or that the validator is registered in the chain.

E.5.1.1. Version 1 - Prototype

```
(func $ext_offchain_is_validator_version_1 (return i32))
```

Arguments:

- `return`: a i32 integer which is equal to `1` if the local node is a potential validator or a integer equal to `0` if it is not.

E.5.2. `ext_offchain_submit_transaction`

Given an extrinsic as a SALE encoded byte array, the system decodes the byte array and submits the extrinsic in the inherent pool as an extrinsic to be included in the next produced block.

E.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 E.2 indicating the byte array storing the encoded extrinsic.
- `return`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined in Definition B.5. Neither on success or failure is there any additional data provided.

E.5.3. `ext_offchain_network_state`

Returns the SCALE encoded, opaque information about the local node's network state. This information is fetched by calling into `libp2p`, which *might* include the `PeerId` and possible `Multiaddress(-es)` by which the node is publicly known by. Those values are unique and have to be known by the node individually. Due to its opaque nature, it's unknown whether that information is available prior to execution.

E.5.3.1. Version 1 - Prototype

```
(func $ext_offchain_network_state_version_1 (result i64))
```

Arguments:

- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined in Definition B.5. On success it contains the SCALE encoded network state. This includes none or one `PeerId` followed by none, one or more IPv4 or IPv6 `Multiaddress(-es)` by which the node is publicly known by. On failure no additional data is provided.

E.5.4. `ext_offchain_timestamp`

Returns current timestamp.

E.5.4.1. Version 1 - Prototype

```
(func $ext_offchain_timestamp_version_1 (result i64))
```

Arguments:

- `result`: an i64 integer indicating the current UNIX timestamp as defined in Definition 1.10.

E.5.5. `ext_offchain_sleep_until`

Pause the execution until ‘deadline‘ is reached.

E.5.5.1. Version 1 - Prototype

```
(func $ext_offchain_sleep_until_version_1 (param $deadline i64))
```

Arguments:

- `deadline`: an i64 integer specifying the UNIX timestamp as defined in Definition 1.10.

E.5.6. `ext_offchain_random_seed`

Generates a random seed. This is a truly random non deterministic seed generated by the host environment.

E.5.6.1. Version 1 - Prototype

```
(func $ext_offchain_random_seed_version_1 (result i32))
```

Arguments:

- `result`: a pointer to the buffer containing the 256-bit seed.

E.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.

E.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 F.1 and a value equal to 2 for local storage as defined in Definition F.2.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `value`: a pointer-size as defined in Definition E.2 indicating the value.

E.5.8. `ext_offchain_local_storage_compare_and_set`

Sets a new value in the local storage if the condition matches the current value.

E.5.8.1. 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 i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition F.1 and a value equal to 2 for local storage as defined in Definition F.2.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `old_value`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the old key.
- `new_value`: a pointer-size as defined in Definition E.2 indicating the new value.
- `result`: an i32 integer equal to `1` if the new value has been set or a value equal to `0` if otherwise.

E.5.9. `ext_offchain_local_storage_get`

Gets a value from the local storage.

E.5.9.1. 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 F.1 and a value equal to 2 for local storage as defined in Definition F.2.
- `key`: a pointer-size as defined in Definition E.2 indicating the key.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the value or the corresponding key.

E.5.10. `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.

E.5.10.1. 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 E.2 indicating the HTTP method. Possible values are ‘‘`GET`’’ and ‘‘`POST`’’.
- `urli`: a pointer-size as defined in Definition E.2 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 E.2 indicating the SCALE encoded `Result` as defined in Definition B.5 containing the i16 ID of the newly started request. On failure no additionally data is provided.

E.5.11. `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).

E.5.11.1. 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 E.2 indicating the HTTP header name.
- `value`: a pointer-size as defined in Definition E.2 indicating the HTTP header value.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined in Definition B.5. Neither on success or failure is there any additional data provided.

E.5.12. `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.

E.5.12.1. 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 E.2 indicating the chunk of bytes. Writing an empty chunk finalizes the request.
- `deadline`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the UNIX timestamp as defined in Definition 1.10. Passing `None` blocks indefinitely.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined Definition B.5. On success, no additional data is provided. On error it contains the HTTP error type as defined in Definition E.10.

E.5.13. `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.

E.5.13.1. 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 E.2 indicating the SCALE encoded array of started request IDs.
- `deadline`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the UNIX timestamp as defined in Definition 1.10. Passing `None` blocks indefinitely.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded array of request statuses as defined in Definition E.9.

E.5.14. `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.

E.5.14.1. 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 E.2 indicating a SCALE encoded array of key/value pairs.

E.5.15. `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.

E.5.15.1. 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 E.2 indicating the buffer where the body gets written to.
- `deadline`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the UNIX timestamp as defined in Definition 1.10. Passing `None` will block indefinitely.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Result` as defined in Definition B.5. On success it contains an i32 integer specifying the number of bytes written or a HTTP error type as defined in Definition E.10 on faiure.

E.6. Trie

Interface that provides trie related functionality.

E.6.1. `ext_trie_blake2_256_root`

Compute a 256-bit Blake2 trie root formed from the iterated items.

E.6.1.1. 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 E.2 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 regular pointer to the buffer containing the 256-bit trie root.

E.6.2. `ext_trie_blake2_256_ordered_root`

Compute a 256-bit Blake2 trie root formed from the enumerated items.

E.6.2.1. 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 E.2 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.12.
- `result`: a regular pointer to the buffer containing the 256-bit trie root result.

E.6.3. `ext_trie_keccak_256_root`

Compute a 256-bit Keccak trie root formed from the iterated items.

E.6.3.1. 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 E.2 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 regular pointer to the buffer containing the 256-bit trie root.

E.6.4. ext_trie_keccak_256_ordered_root

Compute a 256-bit Keccak trie root formed from the enumerated items.

E.6.4.1. 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 E.2 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.12.
- result: a regular pointer to the buffer containing the 256-bit trie root result.

E.7. Miscellaneous

Interface that provides miscellaneous functions for communicating between the runtime and the node.

E.7.1. ext_misc_chain_id

Returns the current relay chain identifier.

E.7.1.1. Version 1 - Prototype

```
(func $ext_misc_chain_id_version_1 (result i64))
```

Arguments:

- result: the current relay chain identifier.

E.7.2. ext_misc_print_num

Print a number.

E.7.2.1. Version 1 - Prototype

```
(func $ext_misc_print_num_version_1 (param $value i64))
```

Arguments:

- value: the number to be printed.

E.7.3. ext_misc_print_utf8

Print a valid UTF8 buffer.

E.7.3.1. Version 1 - Prototype

```
(func $ext_misc_print_utf8_version_1 (param $data i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the valid `UTF8` buffer to be printed.

E.7.4. `ext_misc_print_hex`

Print any buffer in hexadecimal representation.

E.7.4.1. Version 1 - Prototype

```
(func $ext_misc_print_hex_version_1 (param $data i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the buffer to be printed.

E.7.5. `ext_misc_runtime_version`

Extract the Runtime version of the given Wasm blob by calling `Core_version` as defined in Definition G.2.1. Returns the SCALE encoded runtime version or `None` as defined in Definition B.4 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.

E.7.5.1. Version 1 - Prototype

```
(func $ext_misc_runtime_version_version_1 (param $data i64) (result i64))
```

Arguments:

- `data`: a pointer-size as defined in Definition E.2 indicating the Wasm blob.
- `result`: a pointer-size as defined in Definition E.2 indicating the SCALE encoded `Option` as defined in Definition B.4 containing the Runtime version of the given Wasm blob.

E.8. Allocator

Provides functionality for calling into the memory allocator.

E.8.1. `ext_allocator_malloc`

Allocates the given number of bytes and returns the pointer to that memory location.

E.8.1.1. 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 regular pointer to the allocated buffer.

E.8.2. `ext_allocator_free`

Free the given pointer.

E.8.2.1. Version 1 - Prototype

```
(func $ext_allocator_free_version_1 (param $ptr i32))
```

Arguments:

- `ptr`: a regular pointer to the memory buffer to be freed.

E.9. Logging

Interface that provides functions for logging from within the runtime.

Definition E.11. ***Log Level*** *is a varying data type as defined in Definition B.3 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 E.4. *Log Levels for the logging interface*

E.9.1. `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.

E.9.1.1. 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 E.11.
- `target`: a pointer-size as defined in Definition E.2 indicating the string which contains the path, module or location from where the log was executed.
- `message`: a pointer-size as defined in Definition E.2 indicating the log message.

□

Appendix F
Legacy Polkadot Host API

The Legacy Polkadot Host APIs were exceeded and replaces by the current API as described in Appendix E. Those legacy functions are only required for executing Runtimes prior the official Polkadot Runtime, such as the Kusama test network.

Note: This section will be removed in the future.

F.1. Storage

Interface for accessing the storage utilities from within the runtime, including child storages. Child storages are described in Section 2.2.1.

F.1.1. `ext_set_storage`

Sets the value of a specific key in the state storage.

Prototype:

```
(func $ext_storage
  (param $key_data i32) (param $key_len i32) (param $value_data i32)
             (param $value_len i32))
```

Arguments:

- `key`: a pointer indicating the buffer containing the key.
- `key_len`: the key length in bytes.
- `value`: a pointer indicating the buffer containing the value to be stored under the key.
- `value_len`: the length of the value buffer in bytes.

F.1.2. `ext_storage_root`

Retrieves the root of the state storage.

Prototype:

```
(func $ext_storage_root
  (param $result_ptr i32))
```

Arguments:

- `result_ptr`: a memory address pointing at a byte array which contains the root of the state storage after the function concludes.

F.1.3. `ext_blake2_256_enumerated_trie_root`

Given an array of byte arrays, it arranges them in a Merkle trie, defined in Section 2.1.4, where the key under which the values are stored is the 0-based index of that value in the array. It computes and returns the root hash of the constructed trie.

Prototype:

```
(func $ext_blake2_256_enumerated_trie_root
      (param $values_data i32) (param $lens_data i32) (param $lens_len i32)
      (param $result i32))
```

Arguments:

- `values_data`: a memory address pointing at the buffer containing the array where byte arrays are stored consecutively.
- `lens_data`: an array of `i32` elements each stores the length of each byte array stored in `value_data`.
- `lens`_len: the number of `i32` elements in `lens_data`.
- `result`: a memory address pointing at the beginning of a 32-byte byte array containing the root of the Merkle trie corresponding to elements of `values_data`.

F.1.4. `ext_clear_prefix`

Given a byte array, this function removes all storage entries whose key matches the prefix specified in the array.

Prototype:

```
(func $ext_clear_prefix
      (param $prefix_data i32) (param $prefix_len i32))
```

Arguments:

- `prefix_data`: a memory address pointing at the buffer containing the byte array containing the prefix.
- `prefix_len`: the length of the byte array in number of bytes.

F.1.5. `ext_clear_storage`

Given a byte array, this function removes the storage entry whose key is specified in the array.

Prototype:

```
(func $ext_clear_storage
      (param $key_data i32) (param $key_len i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.

F.1.6. `ext_exists_storage`

Given a byte array, this function checks if the storage entry corresponding to the key specified in the array exists.

Prototype:

```
(func $ext_exists_storage
      (param $key_data i32) (param $key_len i32) (result i32)
    )
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.

- `result`: An `i32` integer which is equal to 1 verifies if an entry with the given key exists in the storage or 0 if the key storage does not contain an entry with the given key.

F.1.7. `ext_get_allocated_storage`

Given a byte array, this function allocates a large enough buffer in the memory and retrieves the value stored under the key that is specified in the array. Then, it stores it in the allocated buffer if the entry exists in the storage.

Prototype:

```
    (func $get_allocated_storage
      (param $key_data i32) (param $key_len i32) (param $written_out i32)
(result i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `written_out`: the function stores the length of the retrieved value in number of bytes if the enty exists. If the entry does not exist, it returns $2^{32} - 1$.
- `result`: A pointer to the buffer in which the function allocates and stores the value corresponding to the given key if such an entry exist; otherwise it is equal to 0.

F.1.8. `ext_get_storage_into`

Given a byte array, this function retrieves the value stored under the key specified in the array and stores the specified chunk starting at the offset into the provided buffer, if the entry exists in the storage.

Prototype:

```
    (func $ext_get_storage_into
      (param $key_data i32) (param $key_len i32) (param $value_data i32)
      (param $value_len i32) (param $value_offset i32) (result i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `value_data`: a pointer to the buffer in which the function stores the chunk of the value it retrieves.
- `value_len`: the (maximum) length of the chunk in bytes the function will read of the value and will store in the `value_data` buffer.
- `value_offset`: the offset of the chunk where the function should start storing the value in the provided buffer, i.e. the number of bytes the functions should skip from the retrieved value before storing the data in the `value_data` in number of bytes.
- `result`: The number of bytes the function writes in `value_data` if the value exists or $2^{32} - 1$ if the entry does not exist under the specified key.

F.1.9. `ext_set_child_storage`

Sets the value of a specific key in the child state storage.

Prototype:

```
(func $ext_set_child_storage
   (param $storage_key_data i32) (param $storage_key_len i32) (param $key_data
i32)
   (param $key_len i32) (param $value_data i32) (param $value_len i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key. This key **must** be prefixed with `:child_storage:default:`
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key`: a pointer indicating the buffer containing the key.
- `key_len`: the key length in bytes.
- `value`: a pointer indicating the buffer containing the value to be stored under the key.
- `value_len`: the length of the value buffer in bytes.

F.1.10. `ext_clear_child_storage`

Given a byte array, this function removes the child storage entry whose key is specified in the array.

Prototype:

```
(func $ext_clear_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32)
  (param $key_data i32) (param $key_len i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the key byte array in number of bytes.

F.1.11. `ext_exists_child_storage`

Given a byte array, this function checks if the child storage entry corresponding to the key specified in the array exists.

Prototype:

```
(func $ext_exists_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32)
  (param $key_data i32) (param $key_len i32) (result i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the key byte array in number of bytes.
- `result`: an i32 integer which is equal to 1 verifies if an entry with the given key exists in the child storage or 0 if the child storage does not contain an entry with the given key.

F.1.12. `ext_get_allocated_child_storage`

Given a byte array, this function allocates a large enough buffer in the memory and retrieves the child value stored under the key that is specified in the array. Then, it stores in in the allocated buffer if the entry exists in the child storage.

Prototype:

```
    (func $ext_get_allocated_child_storage
      (param $storage_key_data i32) (param $storage_key_len i32) (param
$key_data i32)             (param $key_len i32) (param $written_out) (result i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the key byte array in number of bytes.
- `written_out`: the function stores the length of the retrieved value in number of bytes if the enty exists. If the entry does not exist, it stores $2^{32} - 1$.
- `result`: A pointer to the buffer in which the function allocates and stores the value corresponding to the given key if such an entry exist; otherwise it is equal to 0.

F.1.13. `ext_get_child_storage_into`

Given a byte array, this function retrieves the child value stored under the key specified in the array and stores the specified chunk starting the offset into the provided buffer, if the entry exists in the storage.

Prototype:

```
    (func $ext_get_child_storage_into
      (param $storage_key_data i32) (param $storage_key_len i32)
      (param $key_data i32) (param $key_len i32) (param $value_data i32)
      (param $value_len i32) (param $value_offset i32) (result i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `value_data`: a pointer to the buffer in which the function stores the chunk of the value it retrieves.
- `value_len`: the (maximum) length of the chunk in bytes the function will read of the value and will store in the `value_data` buffer.
- `value_offset`: the offset of the chunk where the function should start storing the value in the provided buffer, i.e. the number of bytes the functions should skip from the retrieved value before storing the data in the `value_data` in number of bytes.
- `result`: The number of bytes the function writes in `value_data` if the value exists or $2^{32} - 1$ if the entry does not exist under the specified key.

F.1.14. ext_kill_child_storage

Given a byte array, this function removes all entries of the child storage whose child storage key is specified in the array.

Prototype:

```
(func $ext_kill_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.

F.1.15. Memory

F.1.15.1. ext_malloc

Allocates memory of a requested size in the heap.

Prototype:

```
(func $ext_malloc
  (param $size i32) (result i32))
```

Arguments:

- `size`: the size of the buffer to be allocated in number of bytes.

Result:

a memory address pointing at the beginning of the allocated buffer.

F.1.15.2. ext_free

Deallocates a previously allocated memory.

Prototype:

```
(func $ext_free
      (param $addr i32))
```

Arguments:

- `addr`: a 32bit memory address pointing at the allocated memory.

F.1.15.3. Input/Output

- `ext_print_hex`
- `ext_print_num`
- `ext_print_utf8`

F.1.16. Cryptograhpic Auxiliary Functions

F.1.16.1. ext_blake2_256

Computes the Blake2b 256bit hash of a given byte array.

Prototype:

```
(func (export "ext_blake2_256")
      (param $data i32) (param  $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer containing the byte array to be hashed.
- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 32-byte byte array containining the Blake2b hash of the data.

F.1.16.2. `ext_keccak_256`

Computes the Keccak-256 hash of a given byte array.

Prototype:

```
(func $ext_keccak_256
      (param $data i32) (param $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer containing the byte array to be hashed.
- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 32-byte byte array containining the Keccak-256 hash of the data.

F.1.16.3. `ext_twox_128`

Computes the *xxHash64* algorithm (see [Col19]) twice initiated with seeds 0 and 1 and applied on a given byte array and outputs the concatenated result.

Prototype:

```
(func $ext_twox_128
       (param $data i32) (param $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer containing the byte array to be hashed.
- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 16-byte byte array containing $xxhash64_0(\texttt{data})||xxhash64_1(\texttt{data})$ where $xxhash64_i$ is the xxhash64 function initiated with seed i as a 64bit unsigned integer.

F.1.16.4. `ext_ed25519_verify`

Given a message signed by the ED25519 signature algorithm alongside with its signature and the allegedly signer public key, it verifies the validity of the signature by the provided public key.

Prototype:

```
(func $ext_ed25519_verify
      (param $msg_data i32) (param $msg_len i32) (param $sig_data i32)
      (param $pubkey_data i32) (result i32))
```

Arguments:

- `msg_data`: a pointer to the buffer containing the message body.
- `msg_len`: an `i32` integer indicating the size of the message buffer in bytes.
- `sig_data`: a pointer to the 64 byte memory buffer containing the ED25519 signature corresponding to the message.
- `pubkey_data`: a pointer to the 32 byte buffer containing the public key and corresponding to the secret key which has signed the message.

- `result`: an integer value equal to 0 indicating the validity of the signature or a nonzero value otherwise.

F.1.16.5. `ext_sr25519_verify`

Given a message signed by the SR25519 signature algorithm alongside with its signature and the allegedly signer public key, it verifies the validity of the signature by the provided public key.

Prototype:

```
(func $ext_sr25519_verify
      (param $msg_data i32) (param $msg_len i32) (param $sig_data i32)
      (param $pubkey_data i32) (result i32))
```

Arguments:

- `msg_data`: a pointer to the buffer containing the message body.
- `msg_len`: an `i32` integer indicating the size of the message buffer in bytes.
- `sig_data`: a pointer to the 64 byte memory buffer containing the SR25519 signature corresponding to the message.
- `pubkey_data`: a pointer to the 32 byte buffer containing the public key and corresponding to the secret key which has signed the message.
- `result`: an integer value equal to 0 indicating the validity of the signature or a nonzero value otherwise.

F.1.16.6. To be Specced

- `ext_twox_256`

F.1.17. Offchain Worker

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 F.1. ***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 F.2. ***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 F.3. ***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.*

F.1.17.1. `ext_is_validator`

Returns if the local node is a potential validator. Even if this function returns 1, it does not mean that any keys are configured and that the validator is registered in the chain.

Prototype:

```
(func $ext_is_validator
      (result i32))
```

Arguments:

- `result`: an i32 integer which is equal to 1 if the local node is a potential validator or a equal to 0 if it is not.

F.1.17.2. `ext_submit_transaction`

Given an extrinsic as a SCALE encoded byte array, the system decodes the byte array and submits the extrinsic in the inherent pool as an extrinsic to be included in the next produced block.

Prototype:

```
(func $ext_submit_transaction
      (param $data i32) (param $len i32) (result i32))
```

Arguments:

- `data`: a pointer to the buffer containing the byte array storing the encoded extrinsic.
- `len`: an `i32` integer indicating the size of the encoded extrinsic.
- `result`: an integer value equal to 0 indicates that the extrinsic is successfully added to the pool or a nonzero value otherwise.

F.1.17.3. `ext_network_state`

Returns the SCALE encoded, opaque information about the local node's network state. This information is fetched by calling into `libp2p`, which *might* include the `PeerId` and possible `Multiaddress(-es)` by which the node is publicly known by. Those values are unique and have to be known by the node individually. Due to its opaque nature, it's unknown whether that information is available prior to execution.

Prototype:

```
(func $ext_network_state
      (param $written_out i32)(result i32))
```

Arguments:

- `written_out`: a pointer to the 4-byte buffer where the size of the opaque network state gets written to.
- `result`: a pointer to the buffer containing the SCALE encoded network state. This includes none or one `PeerId` followed by none, one or more IPv4 or IPv6 `Multiaddress(-es)` by which the node is publicly known by.

F.1.17.4. `ext_timestamp`

Returns current timestamp.

Prototype:

```
(func $ext_timestamp
      (result i64))
```

Arguments:

- `result`: an i64 integer indicating the current UNIX timestamp as defined in Definition 1.10.

F.1.17.5. `ext_sleep_until`

Pause the execution until ‘deadline‘ is reached.

Prototype:

```
(func $ext_sleep_until
      (param $deadline i64))
```

Arguments:

- `deadline`: an i64 integer specifying the UNIX timestamp as defined in Definition 1.10.

F.1.17.6. `ext_random_seed`

Generates a random seed. This is a truly random non deterministic seed generated by the host environment.

Prototype:

```
(func $ext_random_seed
      (param $seed_data i32))
```

Arguments:

- `seed_data`: a memory address pointing at the beginning of a 32-byte byte array containing the generated seed.

F.1.17.7. `ext_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.

Prototype:

```
(func $ext_local_storage_set
      (param $kind i32) (param $key i32) (param $key_len i32)
      (param $value i32) (param $value_len i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition F.1 and a value equal to 2 for local storage as defined in Definition F.2.
- `key`: a pointer to the buffer containing the key.
- `key_len`: an i32 integer indicating the size of the key.
- `value`: a pointer to the buffer containg the value.
- `value_len`: an i32 integer indicating the size of the value.

F.1.17.8. `ext_local_storage_compare_and_set`

Sets a new value in the local storage if the condition matches the current value.

Prototype:

```
(func $ext_local_storage_compare_and_set
      (param $kind i32) (param $key i32) (param $key_len i32)
      (param $old_value i32) (param $old_value_len) (param $new_value i32)
      (param $new_value_len) (result i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition F.1 and a value equal to 2 for local storage as defined in Definition F.2.
- `key`: a pointer to the buffer containing the key.
- `key_len`: an i32 integer indicating the size of the key.
- `old_value`: a pointer to the buffer containing the current value.
- `old_value_len`: an i32 integer indicating the size of the current value.
- `new_value`: a pointer to the buffer containing the new value.
- `new_value_len`: an i32 integer indicating the size of the new value.
- `result`: an i32 integer equal to 0 if the new value has been set or a value equal to 1 if otherwise.

F.1.17.9. `ext_local_storage_get`

Gets a value from the local storage.

Prototype:

```
(func $ext_local_storage_get
      (param $kind i32) (param $key i32) (param $key_len i32)
      (param $value_len i32) (result i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition F.1 and a value equal to 2 for local storage as defined in Definition F.2.
- `key`: a pointer to the buffer containing the key.
- `key_len`: an i32 integer indicating the size of the key.
- `value_len`: an i32 integer indicating the size of the value.
- `result`: a pointer to the buffer in which the function allocates and stores the value corresponding to the given key if such an entry exist; otherwise it is equal to 0.

F.1.17.10. `ext_http_request_start`

Initiates a http request given by the HTTP method and the URL. Returns the id of a newly started request.

Prototype:

```
(func $ext_http_request_start
      (param $method i32) (param $method_len i32) (param $url i32)
      (param $url_len i32) (param $meta i32) (param $meta_len i32) (result
i32))
```

Arguments:

- `method`: a pointer to the buffer containing the key.
- `method_len`: an i32 integer indicating the size of the method.

- `url`: a pointer to the buffer containing the url.
- `url_len`: an i32 integer indicating the size of the url.
- `meta`: a future-reserved field containing additional, SCALE encoded parameters.
- `meta_len`: an i32 integer indicating the size of the parameters.
- `result`: an i32 integer indicating the ID of the newly started request.

F.1.17.11. `ext_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).

Prototype:

```
(func $ext_http_request_add_header
      (param $request_id i32) (param $name i32) (param $name_len i32)
      (param $value i32) (param $value_len i32) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `name`: a pointer to the buffer containing the header name.
- `name_len`: an i32 integer indicating the size of the header name.
- `value`: a pointer to the buffer containing the header value.
- `value_len`: an i32 integer indicating the size of the header value.
- `result`: an i32 integer where the value equal to 0 indicates if the header has been set or a value equal to 1 if otherwise.

F.1.17.12. `ext_http_request_write_body`

Writes a chunk of the request body. Writing an empty chunk finalises the request. Returns a non-zero value in case the deadline is reached or the chunk could not be written.

Prototype:

```
(func $ext_http_request_write_body
      (param $request_id i32) (param $chunk i32) (param $chunk_len i32)
      (param $deadline i64) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `chunk`: a pointer to the buffer containing the chunk.
- `chunk_len`: an i32 integer indicating the size of the chunk.
- `deadline`: an i64 integer specifying the UNIX timestamp as defined in Definition 1.10. Passing '0' will block indefinitely.
- `result`: an i32 integer where the value equal to 0 indicates if the header has been set or a non-zero value if otherwise.

F.1.17.13. `ext_http_response_wait`

Blocks and waits for the responses for given requests. Returns an array of request statuses (the size is the same as number of IDs).

Prototype:

```
(func $ext_http_response_wait
        (param $ids i32) (param $ids_len i32) (param $statuses i32)
        (param $deadline i64))
```

Arguments:

- `ids`: a pointer to the buffer containing the started IDs.
- `ids_len`: an i32 integer indicating the size of IDs.
- `statuses`: a pointer to the buffer where the request statuses get written to as defined in Definition F.3. The lenght is the same as the length of `ids`.
- `deadline`: an i64 integer indicating the UNIX timestamp as defined in Definition 1.10. Passing '0' as deadline will block indefinitely.

F.1.17.14. `ext_http_response_headers`

Read all response headers. Returns a vector of key/value pairs. Response headers must be read before the response body.

Prototype:

```
(func $ext_http_response_headers
      (param $request_id i32) (param $written_out i32) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `written_out`: a pointer to the buffer where the size of the response headers gets written to.
- `result`: a pointer to the buffer containing the response headers.

F.1.17.15. `ext_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.

Prototype:

```
(func $ext_http_response_read_body
      (param $request_id i32) (param $buffer i32) (param $buffer_len)
      (param $deadline i64) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `buffer`: a pointer to the buffer where the body gets written to.
- `buffer_len`: an i32 integer indicating the size of the buffer.
- `deadline`: an i64 integer indicating the UNIX timestamp as defined in Definition 1.10. Passing '0' will block indefinitely.
- `result`: an i32 integer where the value equal to 0 indicateds a fully consumed response or a non-zero value if otherwise.

F.1.18. Sandboxing

F.1.18.1. To be Specced

- `ext_sandbox_instance_teardown`
- `ext_sandbox_instantiate`
- `ext_sandbox_invoke`

- ext_sandbox_memory_get
- ext_sandbox_memory_new
- ext_sandbox_memory_set
- ext_sandbox_memory_teardown

F.1.19. Auxillary Debugging API

F.1.19.1. ext_print_hex

Prints out the content of the given buffer on the host's debugging console. Each byte is represented as a two-digit hexadecimal number.

Prototype:

```
(func $ext_print_hex
  (param $data i32) (parm $len i32))
```

Arguments:

- data: a pointer to the buffer containing the data that needs to be printed.
- len: an i32 integer indicating the size of the buffer containing the data in bytes.

F.1.19.2. ext_print_utf8

Prints out the content of the given buffer on the host's debugging console. The buffer content is interpreted as a UTF-8 string if it represents a valid UTF-8 string, otherwise does nothing and returns.

Prototype:o

```
(func $ext_print_utf8
  (param $utf8_data i32) (param $utf8_len i32))
```

Arguments:

- utf8_data: a pointer to the buffer containing the utf8-encoded string to be printed.
- utf8_len: an i32 integer indicating the size of the buffer containing the UTF-8 string in bytes.

F.1.20. Misc

F.1.20.1. To be Specced

- ext_chain_id

F.1.21. Block Production

F.2. Validation

□

Appendix G
Runtime Entries

G.1. List of Runtime Entries

The Polkadot Host assumes that at least the following functions are implemented in the Runtime Wasm blob and have been exported as shown in Snippet G.1:

```
(export "Core_version" (func $Core_version))
(export "Core_execute_block" (func $Core_execute_block))
(export "Core_initialize_block" (func $Core_initialize_block))
(export "Metadata_metadata" (func $Metadata_metadata))
(export "BlockBuilder_apply_extrinsic" (func $BlockBuilder_apply_extrinsic))
(export "BlockBuilder_finalize_block" (func $BlockBuilder_finalize_block))
(export "BlockBuilder_inherent_extrinsics"
        (func $BlockBuilder_inherent_extrinsics))
(export "BlockBuilder_check_inherents" (func $BlockBuilder_check_inherents))
(export "BlockBuilder_random_seed" (func $BlockBuilder_random_seed))
(export "TaggedTransactionQueue_validate_transaction"
        (func $TaggedTransactionQueue_validate_transaction))
(export "OffchainWorkerApi_offchain_worker"
        (func $OffchainWorkerApi_offchain_worker))
(export "ParachainHost_validators" (func $ParachainHost_validators))
(export "ParachainHost_duty_roster" (func $ParachainHost_duty_roster))
(export "ParachainHost_active_parachains"
        (func $ParachainHost_active_parachains))
(export "ParachainHost_parachain_status" (func $ParachainHost_parachain_status))
(export "ParachainHost_parachain_code" (func $ParachainHost_parachain_code))
(export "ParachainHost_ingress" (func $ParachainHost_ingress))
(export "GrandpaApi_grandpa_pending_change"
        (func $GrandpaApi_grandpa_pending_change))
(export "GrandpaApi_grandpa_forced_change"
        (func $GrandpaApi_grandpa_forced_change))
(export "GrandpaApi_grandpa_authorities" (func $GrandpaApi_grandpa_authorities))
(export "BabeApi_configuration" (func $BabeApi_configuration))
(export "SessionKeys_generate_session_keys"
        (func $SessionKeys_generate_session_keys))
```

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

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

G.2. Argument Specification

As a wasm functions, all runtime entries have the following prototype signature:

```
(func $generic_runtime_entry
  (param $data i32) (parm $len i32) (result i64))
```

where `data` points to the SCALE encoded paramaters sent to the function and `len` is the length of the data. `result` points to the SCALE encoded data the function returns (See Sections 3.1.2.2 and 3.1.2.3).

In this section, we describe the function of each of the entries alongside with the details of the arguments and the return values for each one of these enteries.

G.2.1. `Core_version`

This entry receives no argument; it returns the version data encoded in ABI format described in Section 3.1.2.3 containing the following information:

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 (G.1)	List of supported APIs along with their version
`transaction_version`	UINT32	the version of the transaction format

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

Definition G.1. ***ApisVec*** *is a specialised type for the* `Core_version` *(G.2.1) function entry. It represents an array of tuples, where the first value of the tuple is an array of 8-bytes indicating the API name. The second value of the tuple is the version number of the corresponding API.*

$$\begin{aligned}\text{ApiVec} &:= (T_0, \ldots, T_n)\\ T &:= ((b_0, \ldots, b_7), \text{UINT32})\end{aligned}$$

G.2.2. `Core_execute_block`

Executes a full block by executing all exctrinsics included in it and update 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:

- The entry accepts 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.

G.2.3. `Core_initialize_block`

Sets up the environment required for building a new block as described in Algorithm 6.7.

Arguments:

- The block header of the new block as defined in 3.6. The values H_r, H_e and H_d are left empty.

Return:

- None.

G.2.4. `hash_and_length`

An auxilarry function which returns hash and encoding length of an extrinsics.

Arguments:

- A blob of an extrinsic.

Return:

- Pair of Blake2Hash of the blob as element of $\mathbb{B}_{32}$ and its length as 64 bit integer.

G.2.5. `BabeApi_configuration`

This entry is called to obtain the current configuration of 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 6.10.	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 G.2. The tuple provided by **BabeApi_configuration**.

G.2.6. `GrandpaApi_grandpa_authorities`

This entry fetches the list of GRANDPA authorities according to the genesis block and is used to initialize authority list defined in Definition 6.1, at genisis. Any future authority changes get tracked via Runtiem-to-consensus engine messages as described in Section 6.1.2.

G.2.7. `TaggedTransactionQueue_validate_transaction`

This entry is invoked against extrinsics submitted through the transaction network message D.1.5 and indicates if the submitted blob represents a valid extrinsics applied to the specified block. This function gets called internally when executing blocks with the `Core_execute_block` runtime function as described in section G.2.2.

Arguments:

- UTX: A byte array that contains the transaction.

Return:

This function returns a `Result` as defined in Definition B.5 which contains the type *ValidTransaction* as defined in Definition G.2 on success and the type *TransactionValidityError* as defined in Definition G.3 on failure.

Definition G.2. ***ValidTransaction*** *is a tuple which contains information concerning a valid transaction.*

Name	***Description***	***Type***
Priority	*Determines the ordering of two transactions that have all their dependencies (required tags) 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 propagated to other peers.*	*Boolean*

Table G.3. *The tuple provided by* `TaggedTransactionQueue_transaction_validity` *in the case the transaction is judged to be valid.*

Note that if *Propagate* is set to `false` the transaction will still be considered for including in blocks that are authored on the current node, but will never be sent to other peers.

Definition G.3. ***TransactionValidityError*** *is a varying data type as defined in Definition B.3, where following values are possible:*

Id	***Description***	***Appended***
0	*The transaction is invalid.*	*InvalidTransaction (G.4)*
1	*The transaction validity can't be determined.*	*UnknownTransaction (G.5)*

Table G.4. *Type variation for the return value of* `TaggedTransactionQueue_transaction_validity`.

Definition G.4. ***InvalidTransaction*** *is a varying data type as defined in Definition B.3 which can get appended to TransactionValidityError and describes the invalid transaction in more precise detail. The following values are possible:*

Id	***Description***	***Appended***
0	*Call: The call of the transaction is not expected*	
1	*Payment: Inability to pay some fees (e.g. balance too low)*	
2	*Future: Transaction not yet valid (e.g. nonce too high)*	
3	*Stale: Transaction is outdated (e.g. nonce too low)*	
4	*BadProof: Bad transaction proof (e.g. bad signature)*	
5	*AncientBirthBlock: Transaction birth block is ancient.*	
6	*ExhaustsResources: Transaction would exhaus the resources of the current block*	
7	*Custom: Any other custom message not covered by this type.*	*one byte*

Table G.5. *Type variant whichs gets appended to Id 0 of* ***TransactionValidityError****.*

Definition G.5. ***UnknownTransacion*** *is a varying data type as defined in Definition B.3 which can get appended to TransactionValidityError and describes the unknown transaction validity in more precise detail. The following values are possible:*

Id	***Description***	***Appended***
0	*CannotLookup: Could not lookup some info that is required for the transaction*	
1	*NoUnsignedValidator: No validator found for the given unsigned transaction.*	
2	*Custom: Any other custom message not covered by this type*	*one byte*

Table G.6. *Type variant whichs gets appended to Id 1 of* ***TransactionValidityError****.*

Note that when this function gets called by the Polkadot host in order to validate a transaction received from peers, Polkadot host usually disregards and rewinds state changes resulting for such a call.

G.2.8. `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:

- An extrinisic.

Return:

- Returns the varying datatype ***ApplyExtrinsicResult*** as defined in Definition G.6.

Definition G.6. ***ApplyExtrinsicResult*** *is the varying data type* **Result** *as defined in Definition B.5. 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 (G.7)*
1	*Possible errors while checking the validity of a transaction.*	*TransactionValidityError (G.10)*

Table G.7. *Possible values of varying data type* ***ApplyExtrinsicResult****.*

Definition G.7. ***DispatchOutcome*** *is the varying data type* **Result** *as defined in Definition B.5.*

Id	***Description***	***Type***
0	*Extrinsic is valid and was submitted successfully.*	*None*
1	*Possible errors while dispatching the extrinsic.*	*DispatchError (G.8)*

Table G.8. *Possible values of varying data type* ***DispatchOutcome****.*

Definition G.8. ***DispatchError*** *is a varying data type as defined in Definition B.3. Indicates various reasons why a dispatch call failed.*

Id	***Description***	***Type***
0	*Some unknown error occured.*	*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 (G.9)*

Table G.9. *Possible values of varying data type* ***DispatchError****.*

Definition G.9. ***CustomModuleError*** *is a tuple appended after a possible error in* ***DispatchError*** *as defined in Defintion G.8.*

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.4). The optional value is a SCALE encoded byte array containing a valid UTF-8 sequence.*

Table G.10. *Possible values of varying data type **CustomModuleError**.*

Definition G.10. ***TransactionValidityError*** *is a varying data type as defined in Definition B.3. It indicates possible errors that can occur while checking the validity of a transaction.*

Id	*Description*	*Type*
0	*Transaction is invalid.*	*InvalidTransaction (G.11)*
1	*Transaction validity can't be determined.*	*UnknownTransaction (G.12)*

Table G.11. *Possible values of varying data type **TransactionValidityError**.*

Definition G.11. ***InvalidTransaction*** *is a varying data type as defined in Definition B.3. Specifies the invalidity of the transaction in more detail.*

Id	*Description*	*Type*
0	*Call of the transaction is not expected.*	*None*
1	*General error to do with the inability to pay some fees (e.g. account balance too low).*	*None*
2	*General error to do with the transaction not being valid (e.g. nonce too high).*	*None*
3	*General error to do with the transaction being outdated (e.g. nonce too low).*	*None*
4	*General error to do with the transactions's proof (e.g. signature)*	*None*
5	*The transaction birth block is ancient.*	*None*
6	*The transaction would exhaust the resources of the current block.*	*None*
7	*Some unknown error occured.*	*Unsigned 8-bit integer*
8	*An extrinsic with mandatory dispatch resulted in an error.*	*None*
9	*A transaction with a mandatory dispatch (only inherents are allowed to have mandatory dispatch).*	*None*

Table G.12. *Possible values of varying data type **InvalidTransaction**.*

Definition G.12. ***UnknownTransaction*** *is a varying data type as defined in Definition B.3. Specifies the unknown invalidity of the transaction in more detail.*

Id	*Description*	*Type*
0	*Could not lookup some information that is required to validate the transaction.*	*None*
1	*No validator found for the given unsigned transaction.*	*None*
2	*Any other custom unknown validity that is not covered by this enum.*	*Unsigned 8-bit integer*

Table G.13. *Possible values of varying data type **UnknownTransaction**.*

G.2.9. `BlockBuilder_inherent_extrinsics`

Generates the inherent extrinsics, which are explained in more detail in section 3.2.3.1. This function takes a SCALE encoded hashtable as defined in section B.7 and returns an array of extrinsics. The Polkadot Host must submit each of those to `BlockBuilder_apply_extrinsic`, described in section G.2.8. This procedure is outlined in algorithm 6.7.

Arguments:

- A Inherents-Data structure as defined in 3.5.

Return:

- An array of extrinisic where each extrinsic is a variable byte array.

G.2.10. `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

□

Glossary

$\boldsymbol{P_n}$	a path graph or a path of n nodes, 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}	13
$\mathbb{B}_n$	a set of all byte arrays of length n	14
I	the little-endian representation of a non-negative interger, represented as $I = (B_n...B_0)_{256}$	14
B	a byte array $B = (b_0, b_1, ..., b_n)$ such that $b_i := B_i$	14
Enc_{LE}	$\begin{array}{lcl} \mathbb{Z}^+ & \rightarrow & \mathbb{B} \\ (B_n...B_0)_{256} & \mapsto & (B_0, B_1, ..., B_n) \end{array}$	14
C, blockchain	a blockchain C is a directed path graph.	14
Block	a node of the graph in blockchain C and indicated by B	14
Genesis Block	the unique sink of blockchain C	14
Head	the source of blockchain C	14
P	for any vertex (B_1, B_2) where $B_1 \rightarrow B_2$ we say B_2 is the parent of B_1 and we indicate it by $B_2 := P(B_1)$	14
BT, block tree	is the union of all different versions of the blockchain observed by all the nodes in the system such as every such block is a node in the graph and B_1 is connected to B_2 if B_1 is a parent of B_2	15
PBT, Pruned BT	Pruned Block Tree refers 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 6.42.	15
pruning		15
G	is the root of the block tree and B is one of its nodes.	15
CHAIN(B)	refers to the path graph from G to B in (P)BT.	15
head of C	defines the head of C to be B, formally noted as $B := \textsc{Head}(C)$.	15
$\lvert C \rvert$	defines he length of C as a path graph	15
SubChain(B', B)	If B' is another node on $\textsc{Chain}(B)$, then by $\textsc{SubChain}(B', B)$ we refer to the subgraph of $\textsc{Chain}(B)$ path graph which contains both B and B'.	15
$\mathbb{C}_{B'}((P)\text{BT})$	is the set of all subchains of (P)BT rooted at B'.	15
$\mathbb{C}$	the set of all chains of (P)BT, $\mathbb{C}_G((P)\text{BT})$ is denoted by $\mathbb{C}$((P)BT) or simply $\mathbb{C}$	15
LONGEST-CHAIN(BT)	the maximum chain given by the complete order over $\mathbb{C}$	15
LONGEST-PATH(BT)	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 6.17.	15
DEEPEST-LEAF(BT)	the head of LONGEST-PATH(BT)	15
StoredValue	the function retrieves the value stored under a specific key in the state storage and is formally defined as $\begin{array}{l} \mathcal{K} \rightarrow \mathcal{V} \\ k \mapsto \begin{cases} v & \text{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{cases} \end{array}$. Here $\mathcal{K} \subset \mathbb{B}$ and $\mathcal{V} \subset \mathbb{B}$ are respectively the set of all keys and values stored in the state storage.	17

Bibliography

[Bur19] Jeff Burdges. Schnorr VRFs and signatures on the Ristretto group. Technical Report, 2019.

[Col19] Yann Collet. Extremely fast non-cryptographic hash algorithm. Technical Report, -, `http://cyan4973.github.io/xxHash/`, 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.

[Fou20] Web3.0 Technologies Foundation. Polkadot Genisis State. Technical Report, `https://github.com/w3f/polkadot-spec/blob/master/genesis-state/`, 2020.

[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.

[lab19] Protocol labs. Libp2p Specification. Technical Report, Protocol labs, `https://github.com/libp2p/specs`, 2019.

[LJ17] Ilari Liusvaara and Simon Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA). 2017.

[Per18] Trevor Perrin. The Noise Protocol Framework. Technical Report, `https://noiseprotocol.org/noise.html`, 2018.

[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.

[Tec19] Parity Technologies. Substrate Reference Documentation. Rust ⟨keepcase | Doc⟩, Parity Technologies, `https://substrate.dev/rustdocs/`, 2019.

INDEX

Transaction Message 25
transaction pool 25
transaction queue 25