Polkadot RE Spec

September 29, 2018

1 Conventions and Definitions

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

Definition 2. *A **Path graph** of n nodes, formally referred to as $\boldsymbol{P_n}$, is a tree a 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}.*

2 Block

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

2.1 Block Header

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

Definition 3. *The header of block B, $\text{Head}(B)$ is a 5-tuple containing the following elements:*

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

2.2 Justified Block Header

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

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

2.3 Extrinsics

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

3 Entry into Runtime

4 API

4.1 Block Submission and Validation

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

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

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

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

Algorithm 1

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

4.2 Storage Access

5 State Storage and the Storage Trie

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

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

A modification is applied to the radix tree structure to improve space efficiency by accounting for the sparseness of the tree. Similarly, the modification done in storing the nodes' hash in the Merkle tree structure is meant to save space on entries storing small entries.

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

5.1 The General Tree Structure

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

To identify the node corresponding to a key value, k, first we need to encode k in a uniform way making sure the length of the encoded value is an integer multiple of 4 bits. [Check this: is probably 8bits/byte, but 4 bit will save some space]

$$k_{\text{enc}} := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_n}) := \text{KeyEncode}(k) \tag{1}$$

where function KeyEncode is defined in Definition ?. k_{enc_i}'s are 4-bit *nibbles*. When looking at k_{enc} as a sequence of nibbles, we can walk the radix tree to reach the node identifying the storage value. This simplified process is shown in Figure

$$[\text{PlaceHolderforsimpleradix} \tag{2}$$

However, the Polkadot trie structure deviate from the standard radix 16 tree structure depicted in Figure ? in order to save storage space by adhering to Axiom 4:

Axiom 4. ***No one child Axiom:*** *No node in the Polkadot RE trie can be a parent of an only child if neither the key of the parent nor the child node correspond to a value in the state database.*

Lemma 5 is a special case of Axiom 4, which states separately as the trie structure is defined separately for the situation according to this case.

Lemma 5. *No node ν in the Polkadot RE trie s hall be root of the path graph of $P_n = (\nu = v_1, \ldots, v_n)$ such that* $\text{Value}(\text{key}(v_i)) = \emptyset$ *for* $1 \leqslant i < n$.

For a definition of the path graph see Definition.

To satisfy Lemma ?, we use *leaf nodes* defined into get rid of path graphs appears in the radix 16 tree.

Definition 6. *We replace any graph path* $P_n = (v_1, \ldots, v_n)$ *in the radix 16 tree representing the state storage with a* ***leaf node*** l_{P_n}*, where* $\text{key}(l_{P_n}) = \text{key}(v_n)$*.*

In all other cases in which the radix 16 tree structure violates Axiom 4

Definition 7. *Suppose* $P_n = (v_1, \ldots, v_n)$ *is a subgraph of the radix 16 tree such that* v_i *for* $1 \leqslant i < n$ *has only one child and* $\text{Value}(\text{key}(v_i)) = \emptyset$*, while* v_n *has more than one child, then we replace* P_n *with an extension node* ex_{P_n}*, where* $\text{key}(\text{ex}_{P_n}) := \text{key}(v_n)$*.*

5.2 Node value