Polkadot Runtime

Protocol Specification

Contents

Chapter 1

Availability and Validity Verification

1.1 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 1.4.1. Once parachain validators have validated a parachain's PoV block successfully, they have to announce that according to the procedure described in Section 1.4.2 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 14 and distribute them among all validators. See Section 1.6 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 1.7 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 un-

available and allow alternative blocks to be built on its parent parachain block, see **??**.

The secondary check described in Section 1.9, 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 **??** for more details.

All validity and invalidity attestations go onto the relay chain, see Section 1.8 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.

1.2 Preliminaries

Definition 1 *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 runtime code of parachain ρ as a WASM Blob.*

Definition 2 *The witness proof of block B, denoted by π_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 3 *Accordingly we define the* **proof of validity block** *or* **PoV** *block in short, PoV_B, to be the tuple:*

$$(B, \pi_B)$$

Definition 4 *The* ***extra validation data****, v_B, 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:*

$$(Max^R_{size}, Max^{head}_{size}, H_i(B^{relay}_{chain}), Head_p(B), \mathbb{B}, Blake2b(R_\rho), R_u)$$

where each value represents:

- Max^R_{size}*: the maximum amount of bytes of the parachain Wasm code permitted.*
- Max^{head}_{size}*: the maximum amount of bytes of the head data (Def. 7) permitted.*
- $H_i(B^{relay}_{chain})$*: the relay chain block number this is in the context of.*
- $Head_p(B)$*: the head data (Def. 7) of the parachain.*

- $\mathbb{B}$: *the balance of the parachain at the moment of validation TODO: @fabio: clarify.*
- *Blake2b*(R_ρ)*: the 32-byte Blake2 hash of the parachain Wasm code.*
- R_u*: the is a varying data type as defined in Definition X TODO: @fabio and can be one of the following values:*

$$R_u = \begin{cases} 0, & None \\ 1, & followed\ by:\ H_i(B_n) \end{cases}$$

R_u implies whether the parachain is allowed to upgrade its validation code. If it's of type 1*, it contains the number of the minimum relay chain height at which the upgrade will be applied, assuming a upgrade is currently signaled TODO: @fabio: where is this signaled?*

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 8 was included on the relay chain the needed relay chain state may have changed.

Definition 5 *Accordingly we define the* **erasure-encoded blob** *or* **blob** *in short, $\bar{B}$, to be the tuple:*

$$(B, \pi_B, v_B)$$

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

1.3 Overal process

The Figure 1.3 demonstrates the overall process of assuring availability and validity in Polkadot TODO: complete the Diagram.

Restricted shell escape. PlantUML cannot be called. Start pdflatex/lualatex with -shell-escape.
@startuml
(*) –¿ "¡math¿Parachain Collator C_ρ Generates B and PoV_B¡/math¿" –¿ "¡math¿C_ρ sends PoV_B to ρ's validator V_ρ¡/math¿" –¿ "¡math¿V_ρ runs ρ's runtime on PoV¡/math¿" if "¡math¿PoV_B is valid¡/math¿" then –¿[true] if "¡math¿V_ρ have seen the CandidateReceipt for PoV_B¡/math¿" then –¿[true] Sign CandidateReceipt –¿[Ending process] (*)
else –¿ [False] "Gerenate CandiateReceipt" –¿[Ending process] (*)
endif else –¿[false] "¡math¿Broadcast message of invalidity for PoV_B¡/math¿" end if
–¿[Ending process] (*)
@enduml

Figure 1.1: Overall process to acheive availability and validity in Polkadot

1.4 Candidate

1.4.1 Candidate Selection

Collators produce candidates as defined in Definition 6. Validators verify the validity of the received candidates as described in Algorithm 1. The validator ensures the that every candidate considered for inclusion has at least one other validator backing it. Candidates without backing are discarded. Backed candidates which later prove to be invalid qualify the backer for slashing.

Definition 6 *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:*

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

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 TODO: @fabio.*
- $Sig^{Collator}_{SR25519}$*: the signature on the 256-bit Blake2 hash of the block data by the collator.*
- $Head_p(B)$*: the head data of parachain block B as defined in Definition 7.*
- $h_b(PoV_B)$*: the 32-byte Blake2 hash of the PoV block.*

Definition 7 *The **head data**, $Head_p(B)$, of a parachain block is a tuple of the following format:*

$$(H_i(B), H_p(B), H_r(B))$$

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. TODO: @fabio: clarify if H_p is the hash of the header or full block TODO: @fabio: maybe define those symbols at the start (already defined in the Host spec)?

Algorithm 1 PrimaryValidation

Input: B, π_B, relay chain parent block B^{relay}_{parent}

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

Algorithm 2 ValidateBlock

Input: B, π_B, v_B

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

1.4.2 Candidate Backing

Validators back the validity respectively the invalidity of candidates by extending those into candidate receipts as defined in Definition 8 and communicate it by issuing statements as defined in Definition 11. Validator v needs to perform Algorithm 3 to announce the result of primary validation to the Polkadot network. If the validator receiver a statement from another validator, the candidate is confirmed based on algorithm 4.

As algorithm 3 and 4 clarifies, the validator should blacklist collators which send invalid candidates. If the invalid candidate originated from another validator, a misbehavior must be announced as defined in TODO: @fabio.

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

Definition 8 *A **candidate receipt**, $C_{receipt}(PoV_B)$, is an extension of a candidate as defined in Definition 6 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 11.*

This type is a tuple of the following format:

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

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.*
- $Head_p(B)$*: the head data of parachain block B as defined in Definition 7. TODO: @fabio (collator module relevant?).*
- id_C*: the Collator relay-chain account ID as defined in Definition TODO: @fabio.*
- $Sig^{Collator}_{SR25519}$*: 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.*
- $CC(PoV_B)$: *Commitments made as a result of validation, as defined in Definition 9.*

Definition 9 ***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:*

$$CC(PoV_B) := (\mathbb{F}, Enc_{SC}(Msg_0, .., Msg_n), H_r(B), \mathbb{U}(R_\rho))$$
$$Msg := (\mathbb{O}, Enc_{SC}(b_0, ..b_n))$$

where each value represents:

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

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

  TODO: @fabio: define the concept of "origin"

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

- $H_r(B)$: *the root of a block's erasure encoding Merkle tree TODO: @fabio: use different symbol for this?.*
- $\mathbb{U}(R_\rho)$: *A varying datatype as defined in Definition TODO: @fabio and can contain one of the following values:*

$$\mathbb{P} = \begin{cases} 0, & None \\ 1, & followed\ by:\ R_\rho \end{cases}$$

  where R_ρ *is a SCALE encoded array containing the new runtime code for the parachain. TODO: @fabio: clarify further*

Definition 10 *A* ***Gossip PoV block*** *is a tuple of the following format:*

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

where $h_b(B_{\substack{relay \\ parent}})$ *is the block hash of the relay chain being referred to and* $h_b(candidate)$ *is the hash of some candidate localized to the same Relay chain block, whose PoV block is* $_b(candidate)$.

1.4.3 Statement Distribution

Definition 11 *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. TODO: @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 datatype as defined in X TODO: @fabio: define 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 3.

Algorithm 3 PRIMARYVALIDATIONANNOUNCEMENT

Input: PoV_B

1: **Init** $Stmt$;
2: **if** VALIDATEBLOCK(PoV_B) is **valid then**
3: $Stmt \leftarrow$ SETVALID(PoV_B)
4: **else**
5: $Stmt \leftarrow$ SETINVALID(PoV_B)
6: BLACKLISTCOLLATOROF(PoV_B)
7: **end if**
8: PROPAGATE($Stmt$)

- VALIDATEBLOCK: Validates PoV_B as defined in Algorithm 2.
- SETVALID: Creates a valid statement as defined in Definition 11.
- SETINVALID: Creates an invalid statement as defined in Definition 11.
- 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 4 ConfirmCandidateReceipt

Input: $Stmt_{peer}$

1: **Init** $Stmt$;
2: $PoV_B \leftarrow$ Retrieve($Stmt_{peer}$)
3: **if** ValidateBlock(PoV_B) is **valid then**
4: **if** AlreadySeconded(B^{relay}_{chain}) **then**
5: $Stmt \leftarrow$ SetValid(PoV_B)
6: **else**
7: $Stmt \leftarrow$ SetSeconded(PoV_B)
8: **end if**
9: **else**
10: $Stmt \leftarrow$ SetInvalid(PoV_B)
11: AnnounceMisbehaviorOf(PoV_B)
12: **end if**
13: Propagate($Stmt$)

- $Stmt_{peer}$: a statement received from another validator.
- Retrieve: Retrieves the PoV block from the statement (11).
- ValidateBlock: Validates PoV_B as defined in Algorithm 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 11.
- SetValid: Creates a valid statement as defined in Definition 11.
- SetSeconded: Creates a seconded statement as defined in Definition 11. Seconding a block should ensure that the next call to AlreadySeconded reliably affirms this action.
- SetInvalid: Creates an invalid statement as defined in Definition 11.
- 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 TODO: @fabio.
- Propagate: sends the statement to the connected peers.

1.4.4 Inclusion of candidate receipt on the relay chain

TODO: @fabio: should this be a subsection?

Definition 12 *Parachain Block Proposal, noted by P^B_ρ 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 12 may/should include the proposal in the block they are producing according to Algorithm 5 during block production procedure.

Algorithm 5 IncludeParachainProposal(P_ρ^B)

Input:

1: TBS

1.4.5 PoV distribution

1.4.6 Primary Validation Disagreement

Parachain validators need to keep track of candidate receipts (see Definition 8) 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 6

Algorithm 6 PrimaryValidationDisagreement

Input:

1: TBS

1.5 Availability

TODO: @fabio: remove this paragraph When a $v \in \mathcal{V}_\rho$ observes that a block containing parachain block candidate receipt is included in a relay chain block RB_p then it must invoke Algorithm 7.

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 13 *The* **erasure encoder/decoder** $encode_{k,n}/decoder_{k,n}$ *is defined to be the Reed-Solomon encoder defined in [?].*

Algorithm 7 ERASURE-ENCODE

Input: $\bar{B}$: blob defined in Definition 5

Init $Shards \leftarrow$ MAKE-SHARDS$(\mathcal{V}_\rho, v_B)$

// Create a trie from the shards in order generate the trie nodes
// which are required to verify each chunk with a Merkle root.
Init $Trie$
Init $counter = 0$
for $shard \in Shards$ **do**
    INSERT$(Trie, counter,$ BLAKE2$(shard))$
    $counter = counter + 1$
end for

// Insert individual chunks into collection.
Init $Chunks$
Init $counter = 0$
for $shard \in Shards$ **do**
    Init $nodes \leftarrow$ GET-NODES$(Trie, counter)$
    ADD$(Chunks, (shard, counter, nodes))$
    $counter = counter + 1$
end for

// $Chunks$ is a collection of individual erasure-encoded chunks (def. 14).
return $Enc_{SC}(Chunks)$

- MAKE-SHARDS(..): return shards for each validator as described in algorithm 8.
- 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.
- ADD$(sequence, item)$: add the given $item$ to the $sequence$.

Algorithm 8 MAKE-SHARDS

Input: $\mathcal{V}_\rho, v_B$

// Calculate the required values for Reed-Solomon.
// Calculate the required lengths.

Init $Shard_{data} = \frac{(|\mathcal{V}_\rho|-1)}{3} + 1$

Init $Shard_{parity} = |\mathcal{V}_\rho| - \frac{(|\mathcal{V}_\rho|-1)}{3} - 1$

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 $Shard_{len} = base_{len} + (base_{len} \bmod 2)$

// Prepare shards, each padded with zeroes.

Init *Shards*

for $n \in (Shard_{data} + Shard_{partiy})$ **do**

    ADD($Shards, (0_0, ..0_{Shard_{len}})$)

end for

// 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

// *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 14 *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. Each chunk is a tuple of the following format:

$$e := (A, u32, A(A_0, ..., A_n))$$

where each value represents:

- *A: SCALE-encoded byte array containg the erasure-encoded chunk of data.*

- *u32: the unsigned 32-bit integer representing the index of this erasure-encoded chunk of data.*
- *$A(A_0, ..., A_n)$: SCALE-encoded array of inner array, each containing the nodes of the Trie in order to verify the chunk based on the Merkle root.*

The algorithm 7 specifies how those values can be calculated.

1.6 Distribution of Chunks

Following the computation of Er_B, v must construct the $\bar{B}$ Availability message defined in Definition 15. And distribute them to target validators designated by the Availability Networking Specification [**?**].

Definition 15 *PoV erasure chunk message $M_{PoV_{\bar{B}}}(i)$ is TBS*

1.7 Announcing Availability

When validator v receives its designated chunk for $\bar{B}$ it needs to broadcast Availability vote message as defined in Definition16

Definition 16 *Availability 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. 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.

Definition 17 *An **availability bitfield** is signed by a particular validator about the availability of pending candidates. It's a tuple of the following format:*

$$(u32, ...)$$

TODO: @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.

1.7.1 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 9 Relay chain's signature processing

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

Based on the result of Algorithm **??** the validator node should mark a parachain block as either available or eventually unavailable according to definitions 18 and **??**

Definition 18 *Parachain blocks for which the corresponding blob is noted on the relay chain to be available, meaning that the candidate receipt has been voted to be available by 2/3 validators.*

After a certain time-out in blocks since we first put the candidate receipt on the relay chain if there is not enough votes of availability the relay chain logic decides that a parachain block is unavailable, see 9.

Definition 19 *An unavailabile parachain block is TBS*

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

1.8 Publishing Attestations

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.

1.9 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 1.9.3 for details. An assigned validator acquires the PoV block (see Section **??**) 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 Section1.9.5.

1.9.1 Approval Checker Assignment

Validators assign themselves to parachain block proposals as defined in Definition 12. 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 1.9.2. Each validator uses the output of the VRF to decide the block(s) they are revalidating as a secondary checker. See Section **??** for the detail.

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

1.9.2 VRF computation

Every validator needs to run Algorithm 10 for every Parachain ρ to determines assignments. TODO: Fix this. It is incorrect so far.

Algorithm 10 VRF-FOR-APPROVAL(B, z, s_k)

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

1: $(\pi, d) \leftarrow VRF(H_h(B), sk(z))$
2: **return** (π, d)

Where VRF function is defined in [**?**].

1.9.3 One-Shot Approval Checker Assignemnt

Every validator v takes the output of this VRF computed by 10 mod the number of parachain blocks that we were decided to be available in this relay chain block according to Definition 18 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 11.

Algorithm 11 OneShotAssignment

Input:

1: TBS

1.9.4 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 12 OneShotAssignment

Input:

1: TBS

1.9.5 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 13

Algorithm 13 EQUIVOCATEDASSIGNMENT

Input:
1: TBS

1.10 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.

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 **??** to parachain validator in order to receive the specific parachain block. Any parachain validator receiving must reply with PoVBlockRespose message defined in Definition 20

Definition 20 *PoV Block Respose Message TBS*

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

Definition 21 *Erasuree-coded chunks request message TBS*

On their part, when a validator receive a erasuree-coded chunks request message it response with the message specified in Definition 22.

Definition 22 *Erasuree-coded chunks response message TBS*

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

Reconstruction

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

Algorithm 14 RECONSTRUCT-POV-ERASURE(S_{Er_B})

Input: $S_{Er_B} := (e_{j_1}, m_{j_1}), \cdot, (e_{j_k}, m_{j_k}))$ such that $k > 2f$

1: $\bar{B} \rightarrow$ ERASURE-DECODER($e_{j_1}, \cdots, e_{j_k}$)
2: **if** ERASURE-DECODER **failed then**
3: ANNOUNCE-FAILURE
4: **return**
5: **end if**
6: $Er_B \rightarrow$ ERASURE-ENCODER($\bar{B}$)
7: **if** VERIFY-MERKLE-PROOF(S_{Er_B}, Er_B) **failed then**
8: ANNOUNCE-FAILURE
9: **return**
10: **end if**
11: **return** $\bar{B}$

1.10.1 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 15 REVALIDATINGRECONSTRUCTEDPOV

Input:

1: TBS

If everything checks out correctly, we declare the block is valid. This means gossiping an attestation, including a reference that identifies candidate receipt and our VRF as specified in Definition 23.

Definition 23 *Secondary approval attetstion message TBS*

1.10.2 Process validity and invalidity messages

When a Black produced receive a Secondary approval attetstion message, it execute Algorithm 16 to verify the VRF and may need to judge when enough time has passed.

Algorithm 16 VERIFYAPPROVALATTESTATION

Input:

1: TBS

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

Definition 24 *Approval Attestation Transaction TBS*

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

Definition 25 *Collator Invalidity Transaction TBS*

Definition 26 *Collator unavailability Transaction*

1.10.3 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 **??** and 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.