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. Once parachain validators have validated a parachain's PoV block successfully, they have to announce that according to the procedure described in Section 1.4.1 where they generate a candidate receipt 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 pieces as described in Section **??** and distribute them among all validators. See Section **??** for details on how this distribution takes place.

Once validators have received erasure-coded pieces 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 pieces on the relay chain by voting on the received pieces, 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 unavailable 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_{rho} 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_{rho} 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.*

This extra validation data includes things like the previous parachain block header, likely including the previous state root. 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 was included on the relay chain the needed relay chain state may have changed.

Definition 5 *Accordingly we define the* **erasure coding 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 Primary Validation

Primary validity checking refers to the process of parachain validators as defined in Definition **??** validating a parachain's PoV block as explained in Algorithm 1.

Algorithm 1 PrimaryValidation

Input: B, π_B, relay chain parent block $B_{relayparent}$

1: Retrieve v_B from the relay chain state at $B_{relayparent}$
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.

1.4.1 Primary validity announcement

Validator v needs to perform Algorithm 3 to announce the result of primary validation to the Polkadot network.

In case that validation has been successful, the announcement will either be in the form of sending the candidate receipt for block B as defined in Definition 6 to the relay chain or confirm a candidate receipt sent in from another parachain validators for this block according to Algorithm 5. However, if the validation fails, v reacts by executing Algorithm 6.

Definition 6 *Candidate Receipt is a proposal for B, TBS.*

Algorithm 3 PrimaryValidationAnnouncement

Input:

1: TBS

Algorithm 4 SendPoVCandidateReceipt

Input:

1: TBS

1.4.2 Inclusion of candidate receipt on the relay chain

Definition 7 *Parachain Block Proposal, noted by P^B_{rho} 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 may/should include the proposal in the block they are producing according to Algorithm 7 during block production procedure.

1.4.3 Primary Validation Disagreement

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

1.5 Availability

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 9.

Definition 8 *The* **erasure encoder/decoder** $encode_{k,n}/decoder_{k,n}$ *is defined to be the Reed-Solomon encoder defined in [?].*

Definition 9 *The* **set of erasure encode pieces** *of $\bar{B}$, denoted by:*

$$Er_B := (e_1, m_1), ..., (e_n, m_n)$$

is defined to be the output of the Algorithm 9.

Algorithm 5 ConfirmCandidateReceipt

Input:

1: TBS

Algorithm 6 AnnouncePrimaryValidationFailure

Input:

1: TBS

Algorithm 7 IncludeParachainProposal(P^B_{rho})

Input:

1: TBS

1.6 Distribution of Pieces

Following the computation of Er_B, v must construct the $\bar{B}$ Availability message defined in Definition 10. And distribute them to target validators designated by the Availability Networking Specification [**?**].

Definition 10 *PoV erasure piece message $M_{PoV_{\bar{B}}}(i)$ is TBS*

1.7 Announcing Availability

When validator v receives its designated piece for $\bar{B}$it needs to broadcast Availability vote message as defined in Definition11

Definition 11 *Availability vote message M^{Avail,v_i}_{PoV} 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.

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 piece of the corresponding parachain block to this candidate receipt.

These signatures go into a relay chain block.

Algorithm 8 PrimaryValidationDisagreement

Input:

1: TBS

Algorithm 9 ERASURE-ENCODE($\bar{B}$, n

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

1: TBS

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 10 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 12 and **??**

Definition 12 *Parachain blocks 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 10.

Definition 13 *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 7. 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 11 for every Parachain ρ to determines assignments. TODO: Fix this. It is incorrect so far.

Algorithm 11 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 11 mod the number of parachain blocks that we were decided to be available in this relay chain block according to Definition 12 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 12.

Algorithm 12 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 13 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 14

Algorithm 14 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 coded pieces.

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 14

Definition 14 *PoV Block Respose Message TBS*

The second method is to retrieve enough erasure coded pieces 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 coded pieces.

Definition 15 *Erasure coded pieces request message TBS*

On their part, when a validator receive a erasure coded pieces request message it response with the message specified in Definition 16.

Definition 16 *Erasure coded pieces response message TBS*

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

Reconstruction

After receiving $2f + 1$ of erasure pieces 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 15 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 16 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 17.

Definition 17 *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 17 to verify the VRF and may need to judge when enough time has passed.

Algorithm 17 VERIFYAPPROVALATTESTATION

Input:

1: TBS

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

Definition 18 *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 19 *Collator Invalidity Transaction TBS*

Definition 20 *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 16.

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 2

Implementer's Guide

2.1 Ramble / Preamble

This document aims to describe the purpose, functionality, and implementation of a host for Polkadot's parachains. It is not for the implementor of a specific parachain but rather for the implementor of the Parachain Host, which provides security and advancement for constituent parachains. In practice, this is for the implementors of Polkadot.

There are a number of other documents describing the research in more detail. All referenced documents will be linked here and should be read alongside this document for the best understanding of the full picture. However, this is the only document which aims to describe key aspects of Polkadot's particular instantiation of much of that research down to low-level technical details and software architecture.

2.2 Origins

Parachains are the solution to a problem. As with any solution, it cannot be understood without first understanding the problem. So let's start by going over the issues faced by blockchain technology that led to us beginning to explore the design space for something like parachains.

2.2.1 Issue 1: Scalability

It became clear a few years ago that the transaction throughput of simple Proof-of-Work (PoW) blockchains such as Bitcoin, Ethereum, and myriad others was simply too low. TODO: PoS, sharding, what if there were more blockchains, etc. etc.

Proof-of-Stake (PoS) systems can accomplish higher throughput than PoW blockchains. PoS systems are secured by bonded capital as opposed to spent effort - liquidity opportunity cost vs. burning electricity. The way they work is by selecting a set of validators with known economic identity who lock up tokens in exchange for earning the right to "validate" or participate in the consensus process. If they are found to carry out that process wrongly, they will be slashed, meaning some or all of the locked tokens will be burned. This provides a strong disincentive in the direction

of misbehavior.

Since the consensus protocol doesn't revolve around wasting effort, block times and agreement can occur much faster. Solutions to PoW challenges don't have to be found before a block can be authored, so the overhead of authoring a block is reduced to only the costs of creating and distributing the block.

However, consensus on a PoS chain requires full agreement of 2/3+ of the validator set for everything that occurs at Layer 1: all logic which is carried out as part of the blockchain's state machine. This means that everybody still needs to check everything. Furthermore, validators may have different views of the system based on the information that they receive over an asynchronous network, making agreement on the latest state more difficult.

Parachains are an example of a **sharded** protocol. Sharding is a concept borrowed from traditional database architecture. Rather than requiring every participant to check every transaction, we require each participant to check some subset of transactions, with enough redundancy baked in that byzantine (arbitrarily malicious) participants can't sneak in invalid transactions - at least not without being detected and getting slashed, with those transactions reverted.

Sharding and Proof-of-Stake in coordination with each other allow a parachain host to provide full security on many parachains, even without all participants checking all state transitions.
TODO: note about network effects & bridging

2.2.2 Issue 2: Flexibility / Specialization

"dumb" VMs don't give you the flexibility. Any engineer knows that being able to specialize on a problem gives them and their users more leverage. TODO: ...

Having recognized these issues, we set out to find a solution to these problems, which could allow developers to create and deploy purpose-built blockchains unified under a common source of security, with the capability of message-passing between them; a heterogeneous sharding solution, which we have come to know as **Parachains**.

2.3 Parachains: Basic Functionality

This section aims to describe, at a high level, the architecture, actors, and Subsystems involved in the implementation of parachains. It also illuminates certain subtleties and challenges faced in the design and implementation of those Subsystems. Our goal is to carry a parachain block from authoring to secure inclusion, and define a process which can be carried out repeatedly and in parallel for many different parachains to extend them over time. Understanding of the high-level approach taken here is important to provide context for the proposed architecture further on.

The Parachain Host is a blockchain, known as the relay-chain, and the actors which provide security and inputs to the blockchain.

First, it's important to go over the main actors we have involved in the parachain host.

1. Validators. These nodes are responsible for validating proposed parachain blocks. They do so by checking a Proof-of-Validity (PoV) of the block and ensuring that the PoV remains available. They put financial capital down as "skin in the game" which can be slashed (destroyed) if they are proven to have misvalidated.
2. Collators. These nodes are responsible for creating the Proofs-of-Validity that validators know how to check. Creating a PoV typically requires familiarity with the transaction format and block authoring rules of the parachain, as well as having access to the full state of the parachain.
3. Fishermen. These are user-operated, permissionless nodes whose goal is to catch misbehaving validators in exchange for a bounty. Collators and validators can behave as Fishermen too. Fishermen aren't necessary for security, and aren't covered in-depth by this document.

This alludes to a simple pipeline where collators send validators parachain blocks and their requisite PoV to check. Then, validators validate the block using the PoV, signing statements which describe either the positive or negative outcome, and with enough positive statements, the block can be noted on the relay-chain. Negative statements are not a veto but will lead to a dispute, with those on the wrong side being slashed. If another validator later detects that a validator or group of validators incorrectly signed a statement claiming a block was valid, then those validators will be slashed, with the checker receiving a bounty.

However, there is a problem with this formulation. In order for another validator to check the previous group of validators' work after the fact, the PoV must remain available so the other validator can fetch it in order to check the work. The PoVs are expected to be too large to include in the blockchain directly, so we require an alternate data availability scheme which requires validators to prove that the inputs to their work will remain available, and so their work can be checked. Empirical tests tell us that many PoVs may be between 1 and 10MB during periods of heavy load.

Here is a description of the Inclusion Pipeline: the path a parachain block (or parablock, for short) takes from creation to inclusion:

1. Validators are selected and assigned to parachains by the Validator Assignment routine.
2. A collator produces the parachain block, which is known as a parachain candidate or candidate, along with a PoV for the candidate.
3. The collator forwards the candidate and PoV to validators assigned to the same parachain via the Collation Distribution Subsystem.
4. The validators assigned to a parachain at a given point in time participate in the Candidate Backing Subsystem to validate candidates that were put forward for validation. Candidates which gather enough signed validity statements from validators are considered "backable". Their backing is the set of signed validity statements.

5. A relay-chain block author, selected by BABE, can note up to one (1) backable candidate for each parachain to include in the relay-chain block alongside its backing. A backable candidate once included in the relay-chain is considered backed in that fork of the relay-chain.

6. Once backed in the relay-chain, the parachain candidate is considered to be "pending availability". It is not considered to be included as part of the parachain until it is proven available.

7. In the following relay-chain blocks, validators will participate in the Availability Distribution Subsystem to ensure availability of the candidate. Information regarding the availability of the candidate will be noted in the subsequent relay-chain blocks.

8. Once the relay-chain state machine has enough information to consider the candidate's PoV as being available, the candidate is considered to be part of the parachain and is graduated to being a full parachain block, or parablock for short.

Note that the candidate can fail to be included in any of the following ways:

- The collator is not able to propagate the candidate to any validators assigned to the parachain.
- The candidate is not backed by validators participating in the Candidate Backing Subsystem.
- The candidate is not selected by a relay-chain block author to be included in the relay chain.
- The candidate's PoV is not considered as available within a timeout and is discarded from the relay chain.

This process can be divided further down. Steps 2 & 3 relate to the work of the collator in collating and distributing the candidate to validators via the Collation Distribution Subsystem. Steps 3 & 4 relate to the work of the validators in the Candidate Backing Subsystem and the block author (itself a validator) to include the block into the relay chain. Steps 6, 7, and 8 correspond to the logic of the relay-chain state-machine (otherwise known as the Runtime) used to fully incorporate the block into the chain. Step 7 requires further work on the validators' parts to participate in the Availability Distribution Subsystem and include that information into the relay chain for step 8 to be fully realized.

This brings us to the second part of the process. Once a parablock is considered available and part of the parachain, it is still "pending approval". At this stage in the pipeline, the parablock has been backed by a majority of validators in the group assigned to that parachain, and its data has been guaranteed available by the set of validators as a whole. Once it's considered available, the host will even begin to accept children of that block. At this point, we can consider the parablock as having been tentatively included in the parachain, although more confirmations are desired. However, the validators in the parachain-group (known as the "Parachain Validators" for that parachain) are sampled from a validator set which contains some proportion of byzantine, or arbitrarily malicious members. This implies that the Parachain Validators for some parachain may be majority-dishonest, which means that secondary checks must be done on the block before it can be considered approved. This is necessary only because the Parachain Validators for a given parachain are sampled from an overall validator set which is assumed to be up to ¡1/3 dishonest - meaning that there is a chance to randomly sample Parachain Validators for a parachain that are majority or fully dishonest and can back a candidate wrongly. The Approval Process allows us to

detect such misbehavior after-the-fact without allocating more Parachain Validators and reducing the throughput of the system. A parablock's failure to pass the approval process will invalidate the block as well as all of its descendents. However, only the validators who backed the block in question will be slashed, not the validators who backed the descendents.

The Approval Process looks like this:

1. Parablocks that have been included by the Inclusion Pipeline are pending approval for a time-window known as the secondary checking window.
2. During the secondary-checking window, validators randomly self-select to perform secondary checks on the parablock.
3. These validators, known in this context as secondary checkers, acquire the parablock and its PoV, and re-run the validation function.
4. The secondary checkers submit the result of their checks to the relay chain. Contradictory results lead to escalation, where even more secondary checkers are selected and the secondary-checking window is extended.
5. At the end of the Approval Process, the parablock is either Approved or it is rejected. More on the rejection process later.

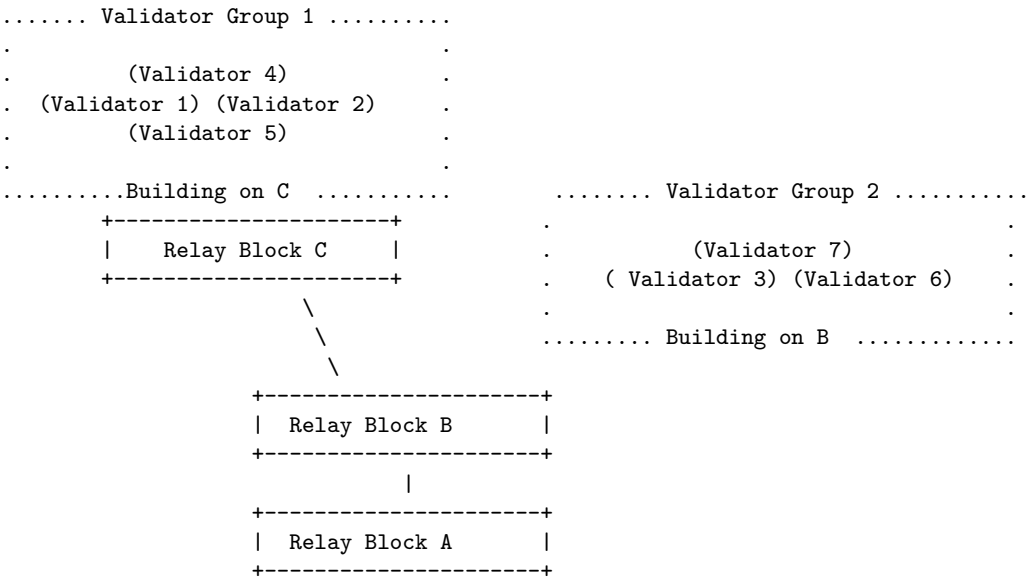
These two pipelines sum up the sequence of events necessary to extend and acquire full security on a Parablock. Note that the Inclusion Pipeline must conclude for a specific parachain before a new block can be accepted on that parachain. After inclusion, the Approval Process kicks off, and can be running for many parachain blocks at once.

Reiterating the lifecycle of a candidate:

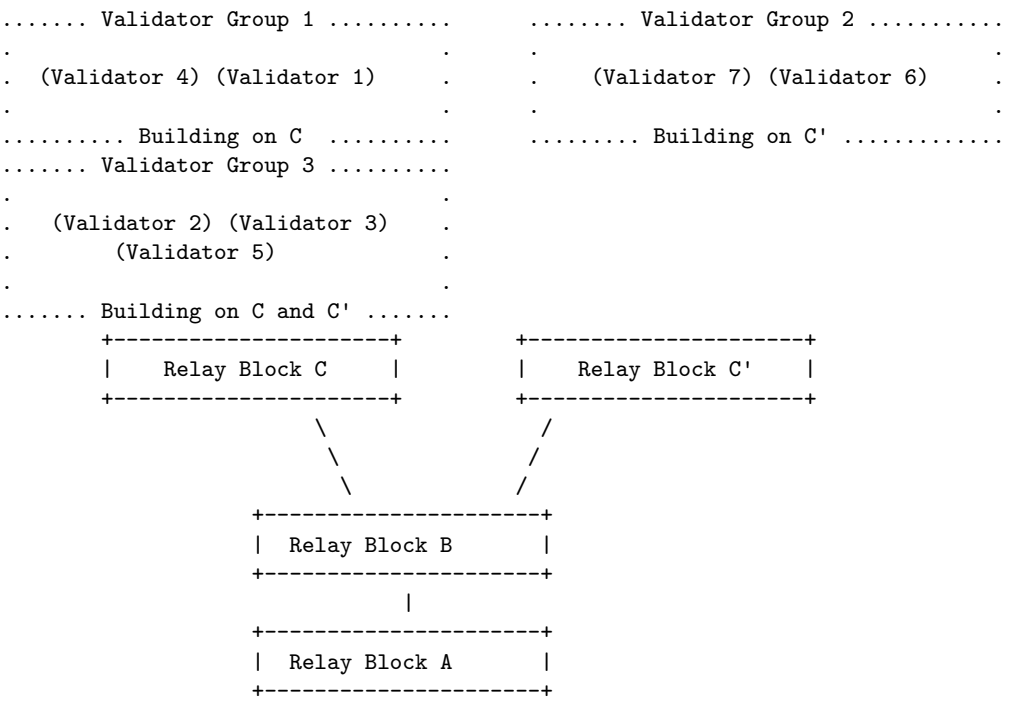
1. Candidate: put forward by a collator to a validator.
2. Seconded: put forward by a validator to other validators.
3. Backable: validity attested to by a majority of assigned validators.
4. Backed: Backable & noted in a fork of the relay-chain.
5. Pending availability: Backed but not yet considered available.
6. Included: Backed and considered available.
7. Accepted: Backed, available, and undisputed

TODO: Diagram: Inclusion Pipeline & Approval Subsystems interaction

It is also important to take note of the fact that the relay-chain is extended by BABE, which is a forkful algorithm. That means that different block authors can be chosen at the same time, and may not be building on the same block parent. Furthermore, the set of validators is not fixed, nor is the set of parachains. And even with the same set of validators and parachains, the validators' assignments to parachains is flexible. This means that the architecture proposed in the next chapters must deal with the variability and multiplicity of the network state.



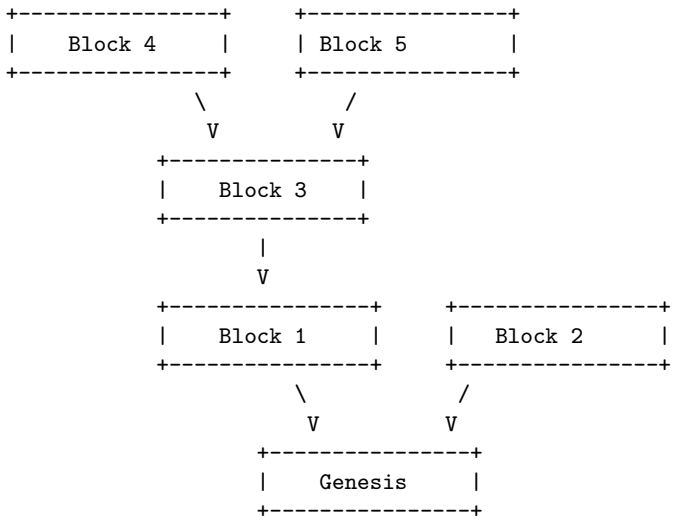
In this example, group 1 has received block C while the others have not due to network asynchrony. Now, a validator from group 2 may be able to build another block on top of B, called C'. Assume that afterwards, some validators become aware of both C and C', while others remain only aware of one.



Those validators that are aware of many competing heads must be aware of the work happening on each one. They may contribute to some or a full extent on both. It is possible that due to network asynchrony two forks may grow in parallel for some time, although in the absence of an adversarial network this is unlikely in the case where there are validators who are aware of both chain heads.

2.4 Architecture

Our Parachain Host includes a blockchain known as the relay-chain. A blockchain is a Directed Acyclic Graph (DAG) of state transitions, where every block can be considered to be the head of a linked-list (known as a "chain" or "fork") with a cumulative state which is determined by applying the state transition of each block in turn. All paths through the DAG terminate at the Genesis Block. In fact, the blockchain is a tree, since each block can have only one parent.



A blockchain network is comprised of nodes. These nodes each have a view of many different forks of a blockchain and must decide which forks to follow and what actions to take based on the forks of the chain that they are aware of.

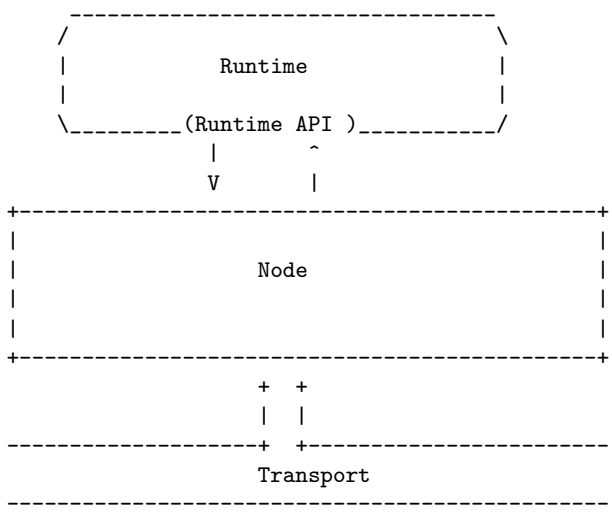
So in specifying an architecture to carry out the functionality of a Parachain Host, we have to answer two categories of questions:

1. What is the state-transition function of the blockchain? What is necessary for a transition to be considered valid, and what information is carried within the implicit state of a block?
2. Being aware of various forks of the blockchain as well as global private state such as a view of the current time, what behaviors should a node undertake? What information should a node extract from the state of which forks, and how should that information be used?

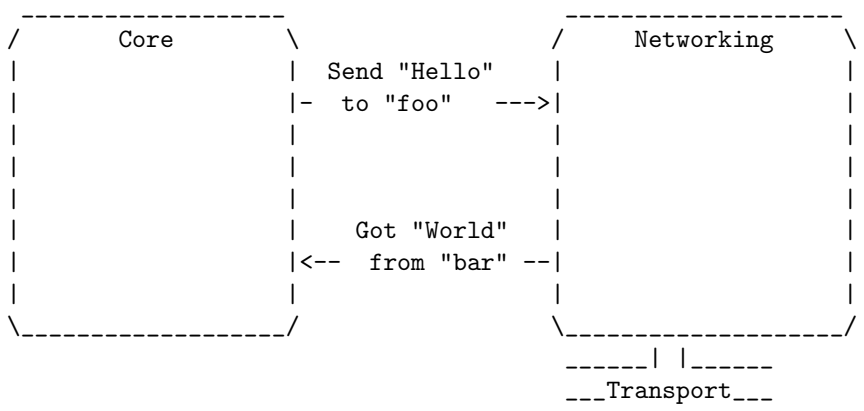
The first category of questions will be addressed by the Runtime, which defines the state-transition logic of the chain. Runtime logic only has to focus on the perspective of one chain, as

each state has only a single parent state.

The second category of questions addressed by Node-side behavior. Node-side behavior defines all activities that a node undertakes, given its view of the blockchain/block-DAG. Node-side behavior can take into account all or many of the forks of the blockchain, and only conditionally undertake certain activities based on which forks it is aware of, as well as the state of the head of those forks.



It is also helpful to divide Node-side behavior into two further categories: Networking and Core. Networking behaviors relate to how information is distributed between nodes. Core behaviors relate to internal work that a specific node does. These two categories of behavior often interact, but can be heavily abstracted from each other. Core behaviors care that information is distributed and received, but not the internal details of how distribution and receipt function. Networking behaviors act on requests for distribution or fetching of information, but are not concerned with how the information is used afterwards. This allows us to create clean boundaries between Core and Networking activities, improving the modularity of the code.



Node-side behavior is split up into various subsystems. Subsystems are long-lived workers that perform a particular category of work. Subsystems can communicate with each other, and do so via an Overseer that prevents race conditions.

Runtime logic is divided up into Modules and APIs. Modules encapsulate particular behavior of the system. Modules consist of storage, routines, and entry-points. Routines are invoked by entry points, by other modules, upon block initialization or closing. Routines can read and alter the storage of the module. Entry-points are the means by which new information is introduced to a module and can limit the origins (user, root, parachain) that they accept being called by. Each block in the blockchain contains a set of Extrinsics. Each extrinsic targets a a specific entry point to trigger and which data should be passed to it. Runtime APIs provide a means for Node-side behavior to extract meaningful information from the state of a single fork.

These two aspects of the implementation are heavily dependent on each other. The Runtime depends on Node-side behavior to author blocks, and to include Extrinsics which trigger the correct entry points. The Node-side behavior relies on Runtime APIs to extract information necessary to determine which actions to take.

2.5 Architecture: Runtime

2.5.1 Broad Strokes

It's clear that we want to separate different aspects of the runtime logic into different modules. Modules define their own storage, routines, and entry-points. They also define initialization and finalization logic.

Due to the (lack of) guarantees provided by a particular blockchain-runtime framework, there is no defined or dependable order in which modules' initialization or finalization logic will run. Supporting this blockchain-runtime framework is important enough to include that same uncertainty in our model of runtime modules in this guide. Furthermore, initialization logic of modules can trigger the entry-points or routines of other modules. This is one architectural pressure against dividing the runtime logic into multiple modules. However, in this case the benefits of splitting things up outweigh the costs, provided that we take certain precautions against initialization and entry-point races.

We also expect, although it's beyond the scope of this guide, that these runtime modules will exist alongside various other modules. This has two facets to consider. First, even if the modules that we describe here don't invoke each others' entry points or routines during initialization, we still have to protect against those other modules doing that. Second, some of those modules are expected to provide governance capabilities for the chain. Configuration exposed by parachain-host modules is mostly for the benefit of these governance modules, to allow the operators or community of the chain to tweak parameters.

The runtime's primary roles to manage scheduling and updating of parachains and parathreads, as well as handling misbehavior reports and slashing. This guide doesn't focus on how parachains or parathreads are registered, only that they are. Also, this runtime description assumes that

validator sets are selected somehow, but doesn't assume any other details than a periodic session change event. Session changes give information about the incoming validator set and the validator set of the following session.

The runtime also serves another role, which is to make data available to the Node-side logic via Runtime APIs. These Runtime APIs should be sufficient for the Node-side code to author blocks correctly.

There is some functionality of the relay chain relating to parachains that we also consider beyond the scope of this document. In particular, all modules related to how parachains are registered aren't part of this guide, although we do provide routines that should be called by the registration process.

We will split the logic of the runtime up into these modules:

- Initializer: manage initialization order of the other modules.
- Configuration: manage configuration and configuration updates in a non-racy manner.
- Paras: manage chain-head and validation code for parachains and parathreads.
- Scheduler: manages parachain and parathread scheduling as well as validator assignments.
- Inclusion: handles the inclusion and availability of scheduled parachains and parathreads.
- Validity: handles secondary checks and dispute resolution for included, available parablocks.

The Initializer module is special - it's responsible for handling the initialization logic of the other modules to ensure that the correct initialization order and related invariants are maintained. The other modules won't specify a on-initialize logic, but will instead expose a special semi-private routine that the initialization module will call. The other modules are relatively straightforward and perform the roles described above.

The Parachain Host operates under a changing set of validators. Time is split up into periodic sessions, where each session brings a potentially new set of validators. Sessions are buffered by one, meaning that the validators of the upcoming session are fixed and always known. Parachain Host runtime modules need to react to changes in the validator set, as it will affect the runtime logic for processing candidate backing, availability bitfields, and misbehavior reports. The Parachain Host modules can't determine ahead-of-time exactly when session change notifications are going to happen within the block (note: this depends on module initialization order again - better to put session before parachains modules). Ideally, session changes are always handled before initialization. It is clearly a problem if we compute validator assignments to parachains during initialization and then the set of validators changes. In the best case, we can recognize that re-initialization needs to be done. In the worst case, bugs would occur.

There are 3 main ways that we can handle this issue:

1. Establish an invariant that session change notifications always happen after initialization. This means that when we receive a session change notification before initialization, we call the initialization routines before handling the session change.

2. Require that session change notifications always occur before initialization. Brick the chain if session change notifications ever happen after initialization.

3. Handle both the before and after cases.

Although option 3 is the most comprehensive, it runs counter to our goal of simplicity. Option 1 means requiring the runtime to do redundant work at all sessions and will also mean, like option 3, that designing things in such a way that initialization can be rolled back and reapplied under the new environment. That leaves option 2, although it is a "nuclear" option in a way and requires us to constrain the parachain host to only run in full runtimes with a certain order of operations.

So the other role of the initializer module is to forward session change notifications to modules in the initialization order, throwing an unrecoverable error if the notification is received after initialization. Session change is the point at which the configuration module updates the configuration. Most of the other modules will handle changes in the configuration during their session change operation, so the initializer should provide both the old and new configuration to all the other modules alongside the session change notification. This means that a session change notification should consist of the following data:

```
struct SessionChangeNotification {
// The new validators in the session.
validators: Vec<ValidatorId>,
// The validators for the next session.
queued: Vec<ValidatorId>,
// The configuration before handling the session change.
prev_config: HostConfiguration,
// The configuration after handling the session change.
new_config: HostConfiguration,
// A secure randomn seed for the session, gathered from BABE.
random_seed: [u8; 32],
}
```

TODO: REVIEW: other options? arguments in favor of going for options 1 or 3 instead of 2. we could do a "soft" version of 2 where we note that the chain is potentially broken due to bad initialization order

TODO: Diagram: order of runtime operations (initialization, session change)

2.5.2 The Initializer Module

Description

This module is responsible for initializing the other modules in a deterministic order. It also has one other purpose as described above: accepting and forwarding session change notifications.

Storage

```
HasInitialized: bool
```

Initialization

The other modules are initialized in this order:

1. Configuration
2. Paras
3. Scheduler
4. Inclusion
5. Validity

The configuration module is first, since all other modules need to operate under the same configuration as each other. It would lead to inconsistency if, for example, the scheduler ran first and then the configuration was updated before the Inclusion module.

Set `HasInitialized` to true.

Session Change

If `HasInitialized` is true, throw an unrecoverable error (panic). Otherwise, forward the session change notification to other modules in initialization order.

Finalization

Finalization order is less important in this case than initialization order, so we finalize the modules in the reverse order from initialization.

Set `HasInitialized` to false.