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.

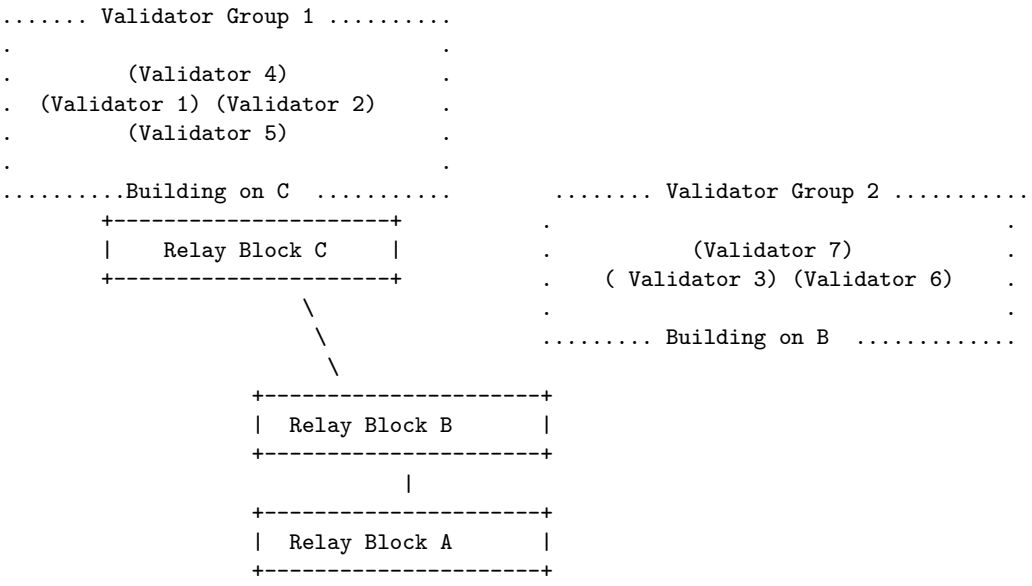
These two pipelines sum up the sequence of events necessary to extend and acquire full security on a Parablock. Note that the Inclusion Pipeline must conclude for a specific parachain before a new block can be accepted on that parachain. After inclusion, the Approval Process kicks off, and can be running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

Reiterating the lifecycle of a candidate:

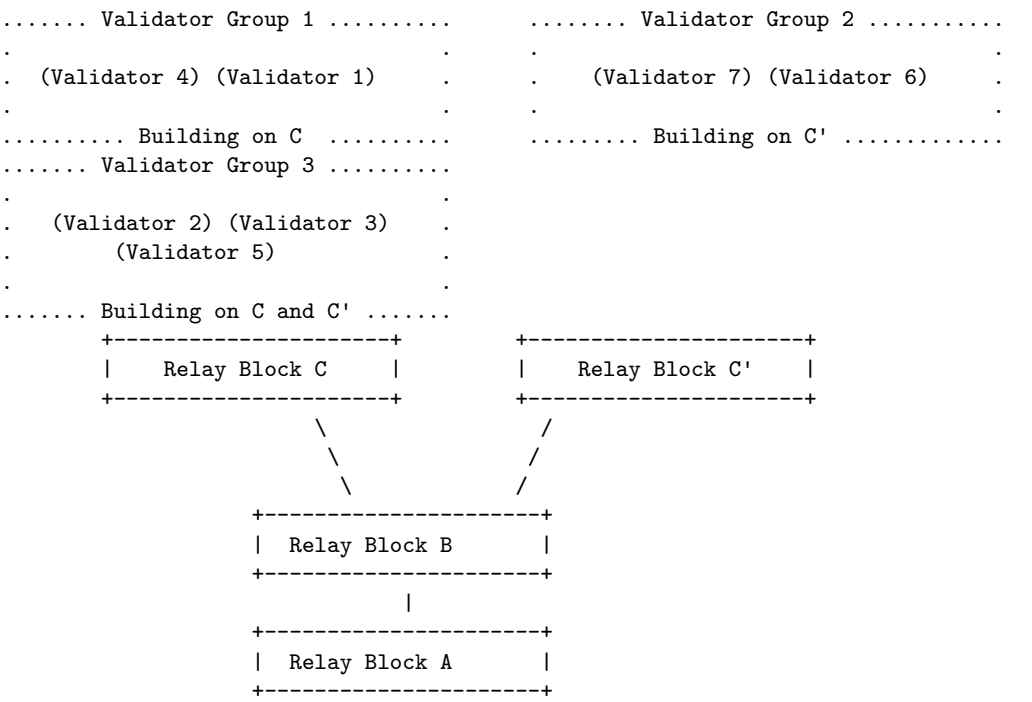
1. Candidate: put forward by a collator to a validator.
2. Seconded: put forward by a validator to other validators.
3. Backable: validity attested to by a majority of assigned validators.
4. Backed: Backable & noted in a fork of the relay-chain.
5. Pending availability: Backed but not yet considered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



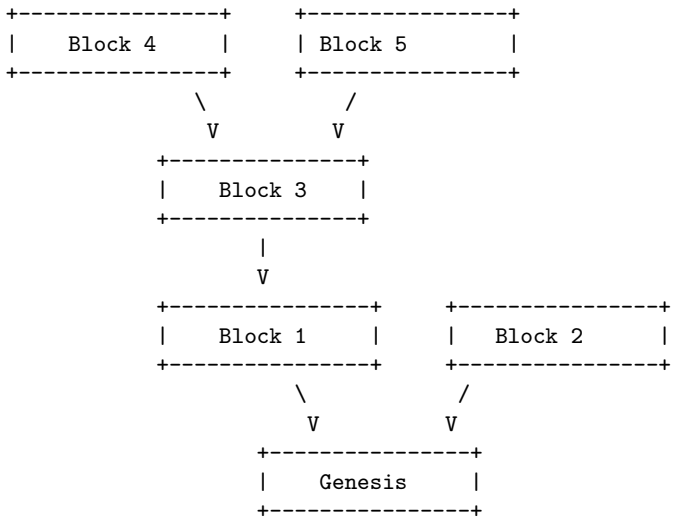
In this example, group 1 has received block C while the others have not due to network asynchrony. Now, a validator from group 2 may be able to build another block on top of B, called C'. Assume that afterwards, some validators become aware of both C and C', while others remain only aware of one.



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.

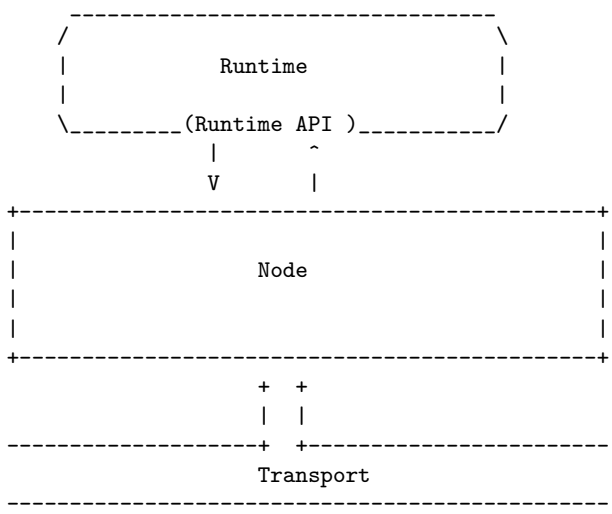
So in specifying an architecture to carry out the functionality of a Parachain Host, we have to answer two categories of questions:

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?

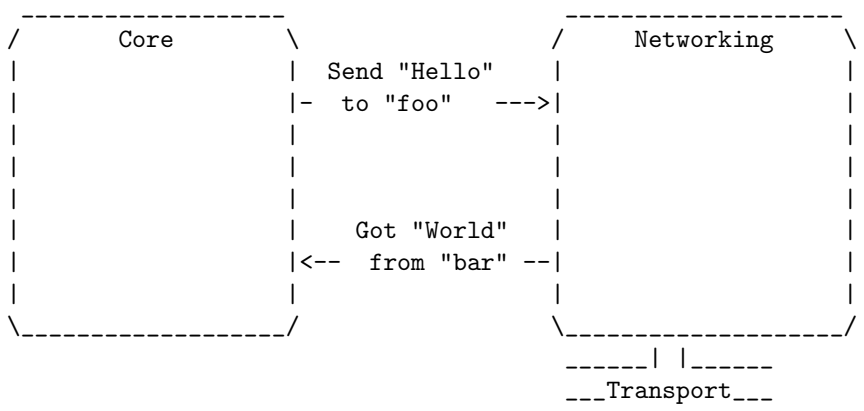
The first category of questions will be addressed by the Runtime, which defines the state-transition logic of the chain. Runtime logic only has to focus on the perspective of one chain, as

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.

2.5.3 The Configuration Module

Description

This module is responsible for managing all configuration of the parachain host in-flight. It provides a central point for configuration updates to prevent races between configuration changes and parachain-processing logic. Configuration can only change during the session change routine, and as this module handles the session change notification first it provides an invariant that the configuration does not change throughout the entire session. Both the scheduler and inclusion modules rely on this invariant to ensure proper behavior of the scheduler.

The configuration that we will be tracking is the `HostConfiguration` struct. TODO: @lamafab: link to type

Storage

The configuration module is responsible for two main pieces of storage.

```
/// The current configuration to be used.
Configuration: HostConfiguration;
/// A pending configuration to be applied on session change.
PendingConfiguration: Option<HostConfiguration>;
```

Session change

The session change routine for the Configuration module is simple. If the `PendingConfiguration` is `Some`, take its value and set `Configuration` to be equal to it. Reset `PendingConfiguration` to `None`.

Routines

```
/// Get the host configuration.
pub fn configuration() -> HostConfiguration {
  Configuration::get()
}
/// Updating the pending configuration to be applied later.
fn update_configuration(f: impl FnOnce(&mut HostConfiguration)) {
  PendingConfiguration::mutate(|pending| {
    let mut x = pending.unwrap_or_else(Self::configuration);
    f(&mut x);
    *pending = Some(x);
  })
}
```

Entry-points

The Configuration module exposes an entry point for each configuration member. These entry-points accept calls only from governance origins. These entry-points will use the `update_configuration` routine to update the specific configuration field.

2.5.4 The Paras Module

Description

The Paras module is responsible for storing information on parachains and parathreads. Registered parachains and parathreads cannot change except at session boundaries. This is primarily to ensure that the number of bits required for the availability bitfields does not change except at session boundaries.

It's also responsible for managing parachain validation code upgrades as well as maintaining availability of old parachain code and its pruning.

Storage

Utility structs:

```
// the two key times necessary to track for every code replacement.
pub struct ReplacementTimes {
/// The relay-chain block number that the code upgrade was expected to be activated.
/// This is when the code change occurs from the para's perspective - after the
/// first parablock included with a relay-parent with number >= this value.
expected_at: BlockNumber,
/// The relay-chain block number at which the parablock activating the code upgrade was
/// actually included. This means considered included and available, so this is the time at which
/// that parablock enters the acceptance period in this fork of the relay-chain.
activated_at: BlockNumber,
}
/// Metadata used to track previous parachain validation code that we keep in
/// the state.
pub struct ParaPastCodeMeta {
// Block numbers where the code was expected to be replaced and where the code
// was actually replaced, respectively. The first is used to do accurate lookups
// of historic code in historic contexts, whereas the second is used to do
// pruning on an accurate timeframe. These can be used as indices
// into the `PastCode` map along with the `ParaId` to fetch the code itself.
upgrade_times: Vec<ReplacementTimes>,
// This tracks the highest pruned code-replacement, if any.
last_pruned: Option<BlockNumber>,
}
enum UseCodeAt {
// Use the current code.
Current,
// Use the code that was replaced at the given block number.
ReplacedAt(BlockNumber),
}
struct ParaGenesisArgs {
  /// The initial head-data to use.
  genesis_head: HeadData,
  /// The validation code to start with.
  validation_code: ValidationCode,
  /// True if parachain, false if parathread.
  parachain: bool,
}
```

Storage layout:

```
/// All parachains. Ordered ascending by ParaId. Parathreads are not included.
Parachains: Vec<ParaId>,
/// The head-data of every registered para.
Heads: map ParaId => Option<HeadData>;
/// The validation code of every live para.
ValidationCode: map ParaId => Option<ValidationCode>;
/// Actual past code, indicated by the para id as well as the block number at which it became outdated.
```

```
PastCode: map (ParaId, BlockNumber) => Option<ValidationCode>;
/// Past code of parachains. The parachains themselves may not be registered anymore,
/// but we also keep their code on-chain for the same amount of time as outdated code
/// to keep it available for secondary checkers.
PastCodeMeta: map ParaId => ParaPastCodeMeta;
/// Which paras have past code that needs pruning and the relay-chain block at which the code was replaced.
/// Note that this is the actual height of the included block, not the expected height at which the
/// code upgrade would be applied, although they may be equal.
/// This is to ensure the entire acceptance period is covered, not an offset acceptance period starting
/// from the time at which the parachain perceives a code upgrade as having occurred.
/// Multiple entries for a single para are permitted. Ordered ascending by block number.
PastCodePruning: Vec<(ParaId, BlockNumber)>;
/// The block number at which the planned code change is expected for a para.
/// The change will be applied after the first parablock for this ID included which executes
/// in the context of a relay chain block with a number >= `expected_at`.
FutureCodeUpgrades: map ParaId => Option<BlockNumber>;
/// The actual future code of a para.
FutureCode: map ParaId => Option<ValidationCode>;
/// Upcoming paras (chains and threads). These are only updated on session change. Corresponds to an
/// entry in the upcoming-genesis map.
UpcomingParas: Vec<ParaId>;
/// Upcoming paras instantiation arguments.
UpcomingParasGenesis: map ParaId => Option<ParaGenesisArgs>;
/// Paras that are to be cleaned up at the end of the session.
OutgoingParas: Vec<ParaId>;
```

Session Change

1. Clean up outgoing paras. This means removing the entries under `Heads`, `ValidationCode`, `FutureCodeUpgrades`, and `FutureCode`. An according entry should be added to `PastCode`, `PastCodeMeta`, and `PastCodePruning` using the outgoing `ParaId` and removed `ValidationCode` value. This is because any outdated validation code must remain available on-chain for a determined amount of blocks, and validation code outdated by de-registering the para is still subject to that invariant.

2. Apply all incoming paras by initializing the `Heads` and `ValidationCode` using the genesis parameters.

3. Amend the Parachains list to reflect changes in registered parachains.

Initialization

1. Do pruning based on all entries in `PastCodePruning` with `BlockNumber <= now`. Update the corresponding `PastCodeMeta` and `PastCode` accordingly.

Routines

- `schedule_para_initialize(ParaId, ParaGenesisArgs)`: schedule a para to be initialized at the next session.

- `schedule_para_cleanup(ParaId)`: schedule a para to be cleaned up at the next session.
- `schedule_code_upgrade(ParaId, ValidationCode, expected_at: BlockNumber)`: Schedule a future code upgrade of the given parachain, to be applied after inclusion of a block of the same parachain executed in the context of a relay-chain block with number $\leq$ `expected_at`.
- `note_new_head(ParaId, HeadData, BlockNumber)`: note that a para has progressed to a new head, where the new head was executed in the context of a relay-chain block with given number. This will apply pending code upgrades based on the block number provided.
- `validation_code_at(ParaId, at: BlockNumber, assume_intermediate: Option<BlockNumber>)`: Fetches the validation code to be used when validating a block in the context of the given relay-chain height. A second block number parameter may be used to tell the lookup to proceed as if an intermediate parablock has been included at the given relay-chain height. This may return past, current, or (with certain choices of `assume_intermediate`) future code. `assume_intermediate`, if provided, must be before `at`. If the validation code has been pruned, this will return `None`.

Finalization

No finalization routine runs for this module.

2.5.5 The Scheduler Module

Description

TODO: this section is still heavily under construction. key questions about availability cores and validator assignment are still open and the flow of the the section may be contradictory or inconsistent.

The Scheduler module is responsible for two main tasks:

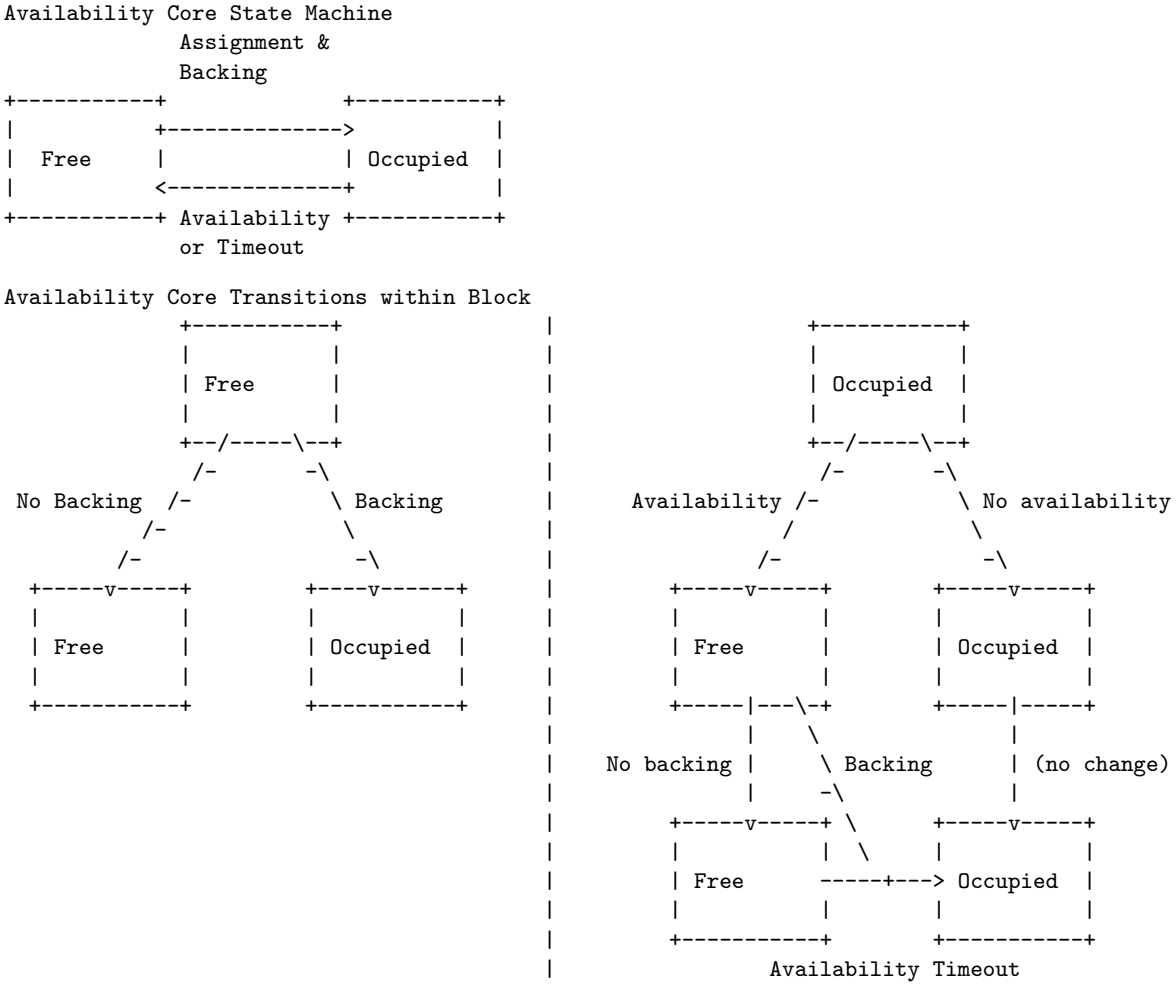
- Partitioning validators into groups and assigning groups to parachains and parathreads.
- Scheduling parachains and parathreads

It aims to achieve these tasks with these goals in mind:

- It should be possible to know at least a block ahead-of-time, ideally more, which validators are going to be assigned to which parachains.
- Parachains that have a candidate pending availability in this fork of the chain should not be assigned.
- Validator assignments should not be gameable. Malicious cartels should not be able to manipulate the scheduler to assign themselves as desired.
- High or close to optimal throughput of parachains and parathreads. Work among validator groups should be balanced.

The Scheduler manages resource allocation using the concept of "Availability Cores". There will be one availability core for each parachain, and a fixed number of cores used for multiplexing parathreads. Validators will be partitioned into groups, with the same number of groups as availability cores. Validator groups will be assigned to different availability cores over time.

An availability core can exist in either one of two states at the beginning or end of a block: free or occupied. A free availability core can have a parachain or parathread assigned to it for the potential to have a backed candidate included. After inclusion, the core enters the occupied state as the backed candidate is pending availability. There is an important distinction: a core is not considered occupied until it is in charge of a block pending availability, although the implementation may treat scheduled cores the same as occupied ones for brevity. A core exits the occupied state when the candidate is no longer pending availability - either on timeout or on availability. A core starting in the occupied state can move to the free state and back to occupied all within a single block, as availability bitfields are processed before backed candidates. At the end of the block, there is a possible timeout on availability which can move the core back to the free state if occupied.



Validator group assignments do not need to change very quickly. The security benefits of fast rotation is redundant with the challenge mechanism in the Validity module. Because of this, we only divide validators into groups at the beginning of the session and do not shuffle membership during the session. However, we do take steps to ensure that no particular validator group has dominance over a single parachain or parathread-multiplexer for an entire session to provide better guarantees of liveness.

Validator groups rotate across availability cores in a round-robin fashion, with rotation occurring at fixed intervals. The i'th group will be assigned to the (i+k)%n'th core at any point in time, where k is the number of rotations that have occurred in the session, and n is the number of cores. This makes upcoming rotations within the same session predictable. TODO: @lamafab: adjust math

When a rotation occurs, validator groups are still responsible for distributing availability pieces for any previous cores that are still occupied and pending availability. In practice, rotation and availability-timeout frequencies should be set so this will only be the core they have just been rotated from. It is possible that a validator group is rotated onto a core which is currently occupied. In this case, the validator group will have nothing to do until the previously-assigned group finishes their availability work and frees the core or the availability process times out. Depending on if the core is for a parachain or parathread, a different timeout `t` from the `HostConfiguration` will apply. Availability timeouts should only be triggered in the first $t - 1$ blocks after the beginning of a rotation.

Parathreads operate on a system of claims. Collators participate in auctions to stake a claim on authoring the next block of a parathread, although the auction mechanism is beyond the scope of the scheduler. The scheduler guarantees that they'll be given at least a certain number of attempts to author a candidate that is backed. Attempts that fail during the availability phase are not counted, since ensuring availability at that stage is the responsibility of the backing validators, not of the collator. When a claim is accepted, it is placed into a queue of claims, and each claim is assigned to a particular parathread-multiplexing core in advance. Given that the current assignments of validator groups to cores are known, and the upcoming assignments are predictable, it is possible for parathread collators to know who they should be talking to now and how they should begin establishing connections with as a fallback.

With this information, the Node-side can be aware of which parathreads have a good chance of being includable within the relay-chain block and can focus any additional resources on backing candidates from those parathreads. Furthermore, Node-side code is aware of which validator group will be responsible for that thread. If the necessary conditions are reached for core reassignment, those candidates can be backed within the same block as the core being freed.

Parathread claims, when scheduled onto a free core, may not result in a block pending availability. This may be due to collator error, networking timeout, or censorship by the validator group. In this case, the claims should be retried a certain number of times to give the collator a fair shot.

Cores are treated as an ordered list of cores and are typically referred to by their index in that list.

Storage

Utility structs:

```
// A claim on authoring the next block for a given parathread.
struct ParathreadClaim(ParaId, CollatorId);
// An entry tracking a claim to ensure it does not pass the maximum number of retries.
struct ParathreadEntry {
  claim: ParathreadClaim,
  retries: u32,
}
// A queued parathread entry, pre-assigned to a core.
struct QueuedParathread {
claim: ParathreadEntry,
core: CoreIndex,
}
struct ParathreadQueue {
queue: Vec<QueuedParathread>,
// this value is between 0 and config.parathread_cores
next_core: CoreIndex,
}
enum CoreOccupied {
  Parathread(ParathreadEntry), // claim & retries
  Parachain,
}
struct CoreAssignment {
  core: CoreIndex,
  para_id: ParaId,
  collator: Option<CollatorId>,
  group_idx: GroupIndex,
}
```

Storage layout:

```
/// All the validator groups. One for each core.
ValidatorGroups: Vec<Vec<ValidatorIndex>>;
/// A queue of upcoming claims and which core they should be mapped onto.
ParathreadQueue: ParathreadQueue;
/// One entry for each availability core. Entries are `None` if the core is not currently occupied. Can be
/// temporarily `Some` if scheduled but not occupied.
/// The i'th parachain belongs to the i'th core, with the remaining cores all being
/// parathread-multiplexers.
AvailabilityCores: Vec<Option<CoreOccupied>>;
/// An index used to ensure that only one claim on a parathread exists in the queue or is
/// currently being handled by an occupied core.
ParathreadClaimIndex: Vec<ParaId>;
/// The block number where the session start occurred. Used to track how many group rotations have occurred.
SessionStartBlock: BlockNumber;
/// Currently scheduled cores - free but up to be occupied. Ephemeral storage item that's wiped on finalization.
Scheduled: Vec<CoreAssignment>, // sorted ascending by CoreIndex.
```

Session Change

Session changes are the only time that configuration can change, and the configuration module's session-change logic is handled before this module's. We also lean on the behavior of the inclusion module which clears all its occupied cores on session change. Thus we don't have to worry about cores being occupied across session boundaries and it is safe to re-size the `AvailabilityCores` bitfield.

Actions:
TODO: @lamafab: adjust subitems

1. Set `SessionStartBlock` to current block number.
2. Clear all `Some` members of `AvailabilityCores`. Return all parathread claims to queue with retries un-incremented.
3. Set `configuration = Configuration::configuration()` (see HostConfiguration) TODO: @lamafab: set link
4. Resize `AvailabilityCores` to have length `Paras::parachains().len() + configuration.parathread_cores with all None` entries.
5. Compute new validator groups by shuffling using a secure randomness beacon.
6. We need a total of `N = Paras::parathreads().len() + configuration.parathread_cores` validator groups.
7. The total number of validators `V` in the `SessionChangeNotification`'s `validators` may not be evenly divided by `V`.
8. First, we obtain "shuffled validators" `SV` by shuffling the validators using the `SessionChangeNotification`'s random seed.
9. The groups are selected by partitioning `SV`. The first V % N groups will have (V / N) + 1 members, while the remaining groups will have (V / N) members each. TODO: @lamafab: adjust math
10. Prune the parathread queue to remove all retries beyond `configuration.parathread_retries`.
11. All pruned claims should have their entry removed from the parathread index.
12. Assign all non-pruned claims to new cores if the number of parathread cores has changed between the `new_config` and `old_config` of the `SessionChangeNotification`.
13. Assign claims in equal balance across all cores if rebalancing, and set the `next_core` of the `ParathreadQueue` by incrementing the relative index of the last assigned core and taking it modulo the number of parathread cores.

Initialization

1. Schedule free cores using the `schedule(Vec::new())`.

Finalization

Actions:

1. Free all scheduled cores and return parathread claims to queue, with retries incremented.

Routines

TODO: @lamafab: adjust subitems

- `add_parathread_claim(ParathreadClaim)`: Add a parathread claim to the queue.
- Fails if any parathread claim on the same parathread is currently indexed.
- Fails if the queue length is ¿= `config.scheduling_lookahead * config.parathread_cores`.
- The core used for the parathread claim is the `next_core` field of the `ParathreadQueue` and adding `Paras::parachains().len()` to it.
- `next_core` is then updated by adding 1 and taking it modulo `config.parathread_cores`.
- The claim is then added to the claim index.
- `schedule(Vec<CoreIndex>)`: schedule new core assignments, with a parameter indicating previously-occupied cores which are to be considered returned.
- All freed parachain cores should be assigned to their respective parachain
- All freed parathread cores should have the claim removed from the claim index.
- All freed parathread cores should take the next parathread entry from the queue.
- The i'th validator group will be assigned to the (i+k)%n'th core at any point in time, where `k` is the number of rotations that have occurred in the session, and `n` is the total number of cores. This makes upcoming rotations within the same session predictable. TODO: @lamafab: adjust math
- scheduled() -¿ Vec¡CoreAssignment¿: Get currently scheduled core assignments.
- occupied(Vec). Note that the given cores have become occupied.
- Fails if any given cores were not scheduled.
- Fails if the given cores are not sorted ascending by core index.
- This clears them from Scheduled and marks each corresponding core in the AvailabilityCores as occupied.
- Since both the availability cores and the newly-occupied cores lists are sorted ascending, this method can be implemented efficiently.
- `core_para(CoreIndex) -> ParaId`: return the currently-scheduled or occupied ParaId for the given core.

- `group_validators(GroupIndex) -> Option<Vec<ValidatorIndex>>`: return all validators in a given group, if the group index is valid for this session.
- `availability_timeout_predicate() -> Option<impl Fn(CoreIndex, BlockNumber) -> bool>`: returns an optional predicate that should be used for timing out occupied cores. If `None`, no timing-out should be done. The predicate accepts the index of the core, and the block number since which it has been occupied. The predicate should be implemented based on the time since the last validator group rotation, and the respective parachain and parathread timeouts, i.e. only within `max(config.chain_availability_period, config.thread_availability_period)` of the last rotation would this return `Some`.

2.5.6 The Inclusion Module

Description

The inclusion module is responsible for inclusion and availability of scheduled parachains and parathreads.

Storage

Helper structs:

```
struct AvailabilityBitfield {
  bitfield: BitVec, // one bit per core.
  submitted_at: BlockNumber, // for accounting, as meaning of bits may change over time.
}
struct CandidatePendingAvailability {
  core: CoreIndex, // availability core
  receipt: AbridgedCandidateReceipt,
  availability_votes: Bitfield, // one bit per validator.
  relay_parent_number: BlockNumber, // number of the relay-parent.
  backed_in_number: BlockNumber,
}
```

Storage Layout:

```
/// The latest bitfield for each validator, referred to by index.
bitfields: map ValidatorIndex => AvailabilityBitfield;
/// Candidates pending availability.
PendingAvailability: map ParaId => CandidatePendingAvailability;
```

TODO: CandidateReceipt and AbridgedCandidateReceipt can contain code upgrades which make them very large. the code entries should be split into a different storage map with infrequent access patterns.

Session Change

1. Clear out all candidates pending availability.
2. Clear out all validator bitfields.

Routines

All failed checks should lead to an unrecoverable error making the block invalid.

- `process_bitfields(Bitfields, core_lookup: Fn(CoreIndex) -> Option<ParaId>)`:
  1. Check that the number of bitfields and bits in each bitfield is correct.
  2. Check that there are no duplicates.
  3. Check all validator signatures.
  4. Apply each bit of bitfield to the corresponding pending candidate. looking up parathread cores using the `core_lookup`. Disregard bitfields that have a `1` bit for any free cores.
  5. For each applied bit of each availability-bitfield, set the bit for the validator in the `CandidatePendingAvailability`'s `availability_votes` bitfield. Track all candidates that now have $> 2 \div 3$ of bits set in their `availability_votes`. These candidates are now available and can be enacted.
  6. For all now-available candidates, invoke the `enact_candidate` routine with the candidate and relay-parent number.
  7. TODO: pass it onwards to Validity module.
  8. Return a list of freed cores consisting of the cores where candidates have become available.
- `process_candidates(BackedCandidates, scheduled: Vec<CoreAssignment>)`:
  1. Check that each candidate corresponds to a scheduled core and that they are ordered in ascending order by `ParaId`.
  2. Ensure that any code upgrade scheduled by the candidate does not happen within `config.validation_upgrade_frequency` of the currently scheduled upgrade, if any, comparing against the value of `Paras::FutureCodeUpgrades` for the given para ID.
  3. Check the backing of the candidate using the signatures and the bitfields.
  4. create an entry in the `PendingAvailability` map for each backed candidate with a blank `availability_votes` bitfield.
  5. Return a `Vec<CoreIndex>` of all scheduled cores of the list of passed assignments that a candidate was successfully backed for, sorted ascending by CoreIndex.
- `enact_candidate(relay_parent_number: BlockNumber, AbridgedCandidateReceipt)`:
  1. If the receipt contains a code upgrade, Call `Paras::schedule_code_upgrade(para_id, code, relay_parent_number + config.validationl_upgrade_delay)`. TODO: Note that this is safe as long as we never enact candidates where the relay parent is across a session boundary. In that case, which we should be careful to avoid with contextual execution, the configuration might have changed and the para may de-sync from the host's understanding of it.
  2. Call `Paras::note_new_head` using the `HeadData` from the receipt and `relay_parent_number`.
- `collect_pending`

```
fn collect_pending(f: impl Fn(CoreIndex, BlockNumber) -> bool) -> Vec<u32> {
  // sweep through all paras pending availability. if the predicate returns true, when given the c
  // the block number the candidate has been pending availability since, then clean up the corresp
  // return a vector of cleaned-up core IDs.
}
```

2.5.7 The InclusionInherent Module

Description

This module is responsible for all the logic carried by the `Inclusion` entry-point. This entry-point is mandatory, in that it must be invoked exactly once within every block, and it is also "inherent", in that it is provided with no origin by the block author. The data within it carries its own authentication. If any of the steps within fails, the entry-point is considered as having failed and the block will be invalid.

This module does not have the same initialization/finalization concerns as the others, as it only requires that entry points be triggered after all modules have initialized and that finalization happens after entry points are triggered. Both of these are assumptions we have already made about the runtime's order of operations, so this module doesn't need to be initialized or finalized by the `Initializer`.

Storage

`Included: Option<()>,`

Finalization

1. Take (get and clear) the value of `Included`. If it is not `Some`, throw an unrecoverable error.

Entry Points

TODO: @lamafab: Link sections

- `inclusion`: This entry-point accepts two parameters: `Bitfields` and `BackedCandidates`.
    1. The `Bitfields` are first forwarded to the `process_bitfields` routine, returning a set of freed cores. Provide a `Scheduler::core_para` as a core-lookup to the `process_bitfields` routine.
    2. If `Scheduler::availability_timeout_predicate` is `Some`, invoke `Inclusion::collect_pending` using it, and add timed-out cores to the free cores.
    3. Invoke `Scheduler::schedule(freed)`.
    4. Pass the `BackedCandidates` along with the output of `Scheduler::scheduled` to the `Inclusion::process_candidates` routine, getting a list of all newly-occupied cores.
    5. Call `Scheduler::occupied` for all scheduled cores where a backed candidate was submitted.
    6. If all of the above succeeds, set `Included` to `Some(())`.

2.5.8 The Validity Module

After a backed candidate is made available, it is included and proceeds into an acceptance period during which validators are randomly selected to do (secondary) approval checks of the parablock. Any reports disputing the validity of the candidate will cause escalation, where even more validators are requested to check the block, and so on, until either the parablock is determined to be invalid or valid. Those on the wrong side of the dispute are slashed and, if the parablock is deemed invalid, the relay chain is rolled back to a point before that block was included.

However, this isn't the end of the story. We are working in a forkful blockchain environment, which carries three important considerations:

1. For security, validators that misbehave shouldn't only be slashed on one fork, but on all possible forks. Validators that misbehave shouldn't be able to create a new fork of the chain when caught and get away with their misbehavior.
2. It is possible that the parablock being contested has not appeared on all forks.
3. If a block author believes that there is a disputed parablock on a specific fork that will resolve to a reversion of the fork, that block author is better incentivized to build on a different fork which does not include that parablock.

This means that in all likelihood, there is the possibility of disputes that are started on one fork of the relay chain, and as soon as the dispute resolution process starts to indicate that the parablock is indeed invalid, that fork of the relay chain will be abandoned and the dispute will never be fully resolved on that chain.

Even if this doesn't happen, there is the possibility that there are two disputes underway, and one resolves leading to a reversion of the chain before the other has concluded. In this case we want to both transplant the concluded dispute onto other forks of the chain as well as the unconcluded dispute.

We account for these requirements by having the validity module handle two kinds of disputes.

1. Local disputes: those contesting the validity of the current fork by disputing a parablock included within it.
2. Remote disputes: a dispute that has partially or fully resolved on another fork which is transplanted to the local fork for completion and eventual slashing.

Local Disputes

TODO: store all included candidate and attestations on them here. accept additional backing after the fact. accept reports based on VRF. candidate included in session S should only be reported on by validator keys from session S. trigger slashing. probably only slash for session S even if the report was submitted in session S+k because it is hard to unify identity.

One first question is to ask why different logic for local disputes is necessary. It seems that local disputes are necessary in order to create the first escalation that leads to block producers

abandoning the chain and making remote disputes possible.

Local disputes are only allowed on parablocks that have been included on the local chain and are in the acceptance period.

For each such parablock, it is guaranteed by the inclusion pipeline that the parablock is available and the relevant validation code is available.

Disputes may occur against blocks that have happened in the session prior to the current one, from the perspective of the chain. In this case, the prior validator set is responsible for handling the dispute and to do so with their keys from the last session. This means that validator duty actually extends 1 session beyond leaving the validator set.

Validators self-select based on the BABE VRF output included by the block author in the block that the candidate became available. TODO: some more details from Jeff's paper. After enough validators have self-selected, the quorum will be clear and validators on the wrong side will be slashed. After concluding, the dispute will remain open for some time in order to collect further evidence of misbehaving validators, and then issue a signal in the header-chain that this fork should be abandoned along with the hash of the last ancestor before inclusion, which the chain should be reverted to, along with information about the invalid block that should be used to blacklist it from being included.

Remote Disputes

When a dispute has occurred on another fork, we need to transplant that dispute to every other fork. This poses some major challenges.

There are two types of remote disputes. The first is a remote roll-up of a concluded dispute. These are simply all attestations for the block, those against it, and the result of all (secondary) approval checks. A concluded remote dispute can be resolved in a single transaction as it is an open-and-shut case of a quorum of validators disagreeing with another.

The second type of remote dispute is the unconcluded dispute. An unconcluded remote dispute is started by any validator, using these things:

- A candidate.
- The session that the candidate has appeared in.
- Backing for that candidate.
- The validation code necessary for validation of the candidate. TODO: optimize by excluding in case where code appears in $Paras :: CurrentCode$ of this fork of relay-chain.
- Secondary checks already done on that candidate, containing one or more disputes by validators. None of the disputes are required to have appeared on other chains. TODO: validator-dispute could be instead replaced by a fisherman w/ bond.

When beginning a remote dispute, at least one escalation by a validator is required, but this validator may be malicious and desires to be slashed. There is no guarantee that the para is registered on this fork of the relay chain or that the para was considered available on any fork of the relay chain.

TODO: @lamafab: link reference So the first step is to have the remote dispute proceed through an availability process similar to the one in the Inclusion Module, but without worrying about core assignments or compactness in bitfields.

We assume that remote disputes are with respect to the same validator set as on the current fork, as BABE and GRANDPA assure that forks are never long enough to diverge in validator set TODO: this is at least directionally correct. handling disputes on other validator sets seems useless anyway as they wouldn't be bonded.

As with local disputes, the validators of the session the candidate was included on another chain are responsible for resolving the dispute and determining availability of the candidate.

If the candidate was not made available on another fork of the relay chain, the availability process will time out and the disputing validator will be slashed on this fork. The escalation used by the validator(s) can be replayed onto other forks to lead the wrongly-escalating validator(s) to be slashed on all other forks as well. We assume that the adversary cannot censor validators from seeing any particular forks indefinitely TODO: set the availability timeout for this accordingly - unlike in the inclusion pipeline we are slashing for unavailability here!

If the availability process passes, the remote dispute is ready to be included on this chain. As with the local dispute, validators self-select based on a VRF. Given that a remote dispute is likely to be replayed across multiple forks, it is important to choose a VRF in a way that all forks processing the remote dispute will have the same one. Choosing the VRF is important as it should not allow an adversary to have control over who will be selected as a secondary approval checker.

After enough validator self-select, under the same escalation rules as for local disputes, the Remote dispute will conclude, slashing all those on the wrong side of the dispute. After concluding, the remote dispute remains open for a set amount of blocks to accept any further proof of additional validators being on the wrong side.

2.5.9 Slashing and Incentivization

The goal of the dispute is to garner a $2/3 + (2f + 1)$ supermajority either in favor of or against the candidate.

For remote disputes, it is possible that the parablock disputed has never actually passed any availability process on any chain. In this case, validators will not be able to obtain the PoV of the parablock and there will be relatively few votes. We want to disincentivize voters claiming validity of the block from preventing it from becoming available, so we charge them a small distraction fee for wasting the others' time if the dispute does not garner a 2/3+ supermajority on either side. This fee can take the form of a small slash or a reduction in rewards.

When a supermajority is achieved for the dispute in either the valid or invalid direction, we will penalize non-voters either by issuing a small slash or reducing their rewards. We prevent censorship of the remaining validators by leaving the dispute open for some blocks after resolution in order to accept late votes.

2.6 Architecture: Node-side

Design Goals

- Modularity: Components of the system should be as self-contained as possible. Communication boundaries between components should be well-defined and mockable. This is key to creating testable, easily reviewable code.
- Minimizing side effects: Components of the system should aim to minimize side effects and to communicate with other components via message-passing.
- Operational Safety: The software will be managing signing keys where conflicting messages can lead to large amounts of value to be slashed. Care should be taken to ensure that no messages are signed incorrectly or in conflict with each other.

The architecture of the node-side behavior aims to embody the Rust principles of ownership and message-passing to create clean, isolatable code. Each resource should have a single owner, with minimal sharing where unavoidable.

Many operations that need to be carried out involve the network, which is asynchronous. This asynchrony affects all core subsystems that rely on the network as well. The approach of hierarchical state machines is well-suited to this kind of environment.

We introduce a hierarchy of state machines consisting of an overseer supervising subsystems, where Subsystems can contain their own internal hierarchy of jobs. This is elaborated on in the next section on Subsystems.

2.6.1 Subsystems and Jobs

In this section we define the notions of Subsystems and Jobs. These are guidelines for how we will employ an architecture of hierarchical state machines. We'll have a top-level state machine which oversees the next level of state machines which oversee another layer of state machines and so on. The next sections will lay out these guidelines for what we've called subsystems and jobs, since this model applies to many of the tasks that the Node-side behavior needs to encompass, but these are only guidelines and some Subsystems may have deeper hierarchies internally.

Subsystems are long-lived worker tasks that are in charge of performing some particular kind of work. All subsystems can communicate with each other via a well-defined protocol. Subsystems can't generally communicate directly, but must coordinate communication through an Overseer, which is responsible for relaying messages, handling subsystem failures, and dispatching work signals.

Most work that happens on the Node-side is related to building on top of a specific relay-chain block, which is contextually known as the "relay parent". We call it the relay parent to explicitly denote that it is a block in the relay chain and not on a parachain. We refer to the parent because when we are in the process of building a new block, we don't know what that new block is going to be. The parent block is our only stable point of reference, even though it is usually only useful when it is not yet a parent but in fact a leaf of the block-DAG expected to soon become a parent (because validators are authoring on top of it). Furthermore, we are assuming a forkful blockchain-extension protocol, which means that there may be multiple possible children of the relay-parent. Even if the relay parent has multiple children blocks, the parent of those children is the same, and the context in which those children is authored should be the same. The parent block is the best and most stable reference to use for defining the scope of work items and messages, and is typically referred to by its cryptographic hash.

Since this goal of determining when to start and conclude work relative to a specific relay-parent is common to most, if not all subsystems, it is logically the job of the Overseer to distribute those signals as opposed to each subsystem duplicating that effort, potentially being out of synchronization with each other. Subsystem A should be able to expect that subsystem B is working on the same relay-parents as it is. One of the Overseer's tasks is to provide this heartbeat, or synchronized rhythm, to the system.

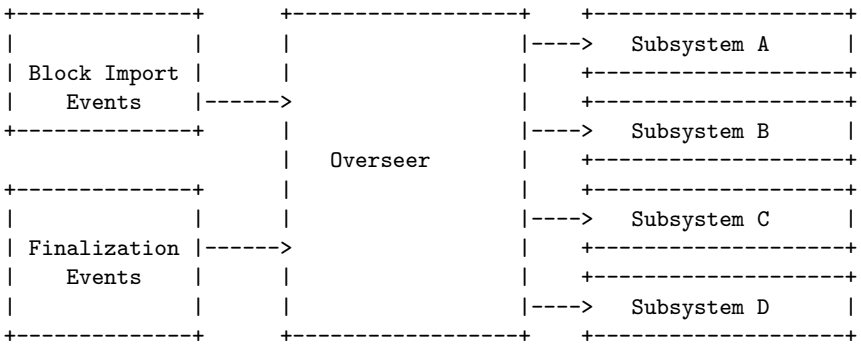
The work that subsystems spawn to be done on a specific relay-parent is known as a job. Subsystems should set up and tear down jobs according to the signals received from the overseer. Subsystems may share or cache state between jobs.

2.6.2 Overseer

The overseer is responsible for these tasks:

1. Setting up, monitoring, and handing failure for overseen subsystems.
2. Providing a "heartbeat" of which relay-parents subsystems should be working on.
3. Acting as a message bus between subsystems.

The hierarchy of subsystems:



The overseer determines work to do based on block import events and block finalization events. It does this by keeping track of the set of relay-parents for which work is currently being done. This is known as the "active leaves" set. It determines an initial set of active leaves on startup based on the data on-disk, and uses events about blockchain import to update the active leaves. Updates lead to `OverseerSignal::StartWork` and `OverseerSignal::StopWork` being sent according to new relay-parents, as well as relay-parents to stop considering. Block import events inform the overseer of leaves that no longer need to be built on, now that they have children, and inform us to begin building on those children. Block finalization events inform us when we can stop focusing on blocks that appear to have been orphaned.

The overseer's logic can be described with these functions:

On Startup

- Start all subsystems.
- Determine all blocks of the blockchain that should be built on. This should typically be the head of the best fork of the chain we are aware of. Sometimes add recent forks as well.
- For each of these blocks, send an `OverseerSignal::StartWork` to all subsystems.
- Begin listening for block import and finality events.

On Block Import Event

- Apply the block import event to the active leaves. A new block should lead to its addition to the active leaves set and its parent being deactivated.
- For any deactivated leaves send an `OverseerSignal::StopWork` message to all subsystems.
- For any activated leaves send an `OverseerSignal::StartWork` message to all subsystems.
- Ensure all `StartWork` messages are flushed before resuming activity as a message router.

TODO: in the future, we may want to avoid building on too many sibling blocks at once. the notion of a "preferred head" among many competing sibling blocks would imply changes in our "active leaves" update rules here.

On Finalization Event

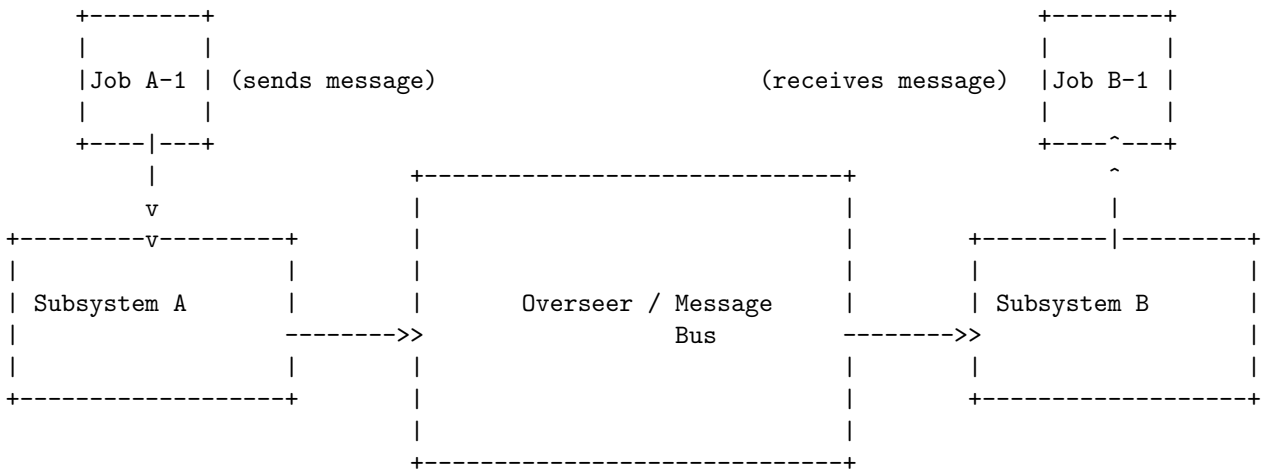
- Note the height `h` of the newly finalized block `B`.
- Prune all leaves from the active leaves which have height $\leq h$ and are not `B`.
- Issue OverseerSignal::StopWork for all deactivated leaves.

On Subsystem Failure

Subsystems are essential tasks meant to run as long as the node does. Subsystems can spawn ephemeral work in the form of jobs, but the subsystems themselves should not go down. If a subsystem goes down, it will be because of a critical error that should take the entire node down as well.

Communication Between Subsystems

When a subsystem wants to communicate with another subsystem, or, more typically, a job within a subsystem wants to communicate with its counterpart under another subsystem, that communication must happen via the overseer. Consider this example where a job on subsystem A wants to send a message to its counterpart under subsystem B. This is a realistic scenario, where you can imagine that both jobs correspond to work under the same relay-parent.



First, the subsystem that spawned a job is responsible for handling the first step of the communication. The overseer is not aware of the hierarchy of tasks within any given subsystem and is only responsible for subsystem-to-subsystem communication. So the sending subsystem must pass on the message via the overseer to the receiving subsystem, in such a way that the receiving subsystem can further address the communication to one of its internal tasks, if necessary.

This communication prevents a certain class of race conditions. When the Overseer determines that it is time for subsystems to begin working on top of a particular relay-parent, it will dispatch a `StartWork` message to all subsystems to do so, and those messages will be handled asynchronously by those subsystems. Some subsystems will receive those messsages before others, and it is important that a message sent by subsystem A after receiving `StartWork` message will arrive at subsystem B after its `StartWork` message. If subsystem A maintaned an independent channel with subsystem B to communicate, it would be possible for subsystem B to handle the side message before the `StartWork` message, but it wouldn't have any logical course of action to take with the side message - leading to it being discarded or improperly handled. Well-architectured state machines should have a single source of inputs, so that is what we do here.

One exception is reasonable to make for responses to requests. A request should be made via the overseer in order to ensure that it arrives after any relevant `StartWork` message. A subsystem

issuing a request as a result of a `StartWork` message can safely receive the response via a side-channel for two reasons:

1. It's impossible for a request to be answered before it arrives, it is provable that any response to a request obeys the same ordering constraint.

2. The request was sent as a result of handling a `StartWork` message. Then there is no possible future in which the `StartWork` message has not been handled upon the receipt of the response.

So as a single exception to the rule that all communication must happen via the overseer we allow the receipt of responses to requests via a side-channel, which may be established for that purpose. This simplifies any cases where the outside world desires to make a request to a subsystem, as the outside world can then establish a side-channel to receive the response on.

It's important to note that the overseer is not aware of the internals of subsystems, and this extends to the jobs that they spawn. The overseer isn't aware of the existence or definition of those jobs, and is only aware of the outer subsystems with which it interacts. This gives subsystem implementations leeway to define internal jobs as they see fit, and to wrap a more complex hierarchy of state machines than having a single layer of jobs for relay-parent-based work. Likewise, subsystems aren't required to spawn jobs. Certain types of subsystems, such as those for shared storage or networking resources, won't perform block-based work but would still benefit from being on the Overseer's message bus. These subsystems can just ignore the overseer's signals for block-based work.

Furthermore, the protocols by which subsystems communicate with each other should be well-defined irrespective of the implementation of the subsystem. In other words, their interface should be distinct from their implementation. This will prevent subsystems from accessing aspects of each other that are beyond the scope of the communication boundary.

2.6.3 Candidate Backing Subsystem

Description

The Candidate Backing subsystem is engaged in by validators in to contribute to the backing of parachain candidates submitted by other validators.

TODO: @lamafab: link reference Its role is to produce backable candidates for inclusion in new relay-chain blocks. It does so by issuing signed Statements and tracking received statements signed by other validators. Once enough statements are received, they can be combined into backing for specific candidates.

It also detects double-vote misbehavior by validators as it imports votes, passing on the misbehavior to the correct reporter and handler.

When run as a validator, this is the subsystem which actually validates incoming candidates.

Protocol

TODO: @lamafab: link reference This subsystem receives messages of the type CandidateBacking-SubsystemMessage.

Functionality

The subsystem should maintain a set of handles to Candidate Backing Jobs that are currently live, as well as the relay-parent to which they correspond.

On Overseer Signal

- If the signal is an `OverseerSignal::StartWork(relay_parent)`, spawn a Candidate Backing Job with the given relay parent, storing a bidirectional channel with the Candidate Backing Job in the set of handles.
- If the signal is an `OverseerSignal::StopWork(relay_parent)`, cease the Candidate Backing Job under that relay parent, if any.

On CandidateBackingSubsystemMessage

- If the message corresponds to a particular relay-parent, forward the message to the Candidate Backing Job for that relay-parent, if any is live.

TODO: "contextual execution". 1.) At the moment we only allow inclusion of new parachain candidates validated by current validators, 2.) Allow inclusion of old parachain candidates validated by current validators, 3.) Allow inclusion of old parachain candidates validated by old validators.

This will probably blur the lines between jobs, will probably require inter-job communication and a short-term memory of recently backable, but not backed candidates.

Candidate Backing Job

The Candidate Backing Job represents the work a node does for backing candidates with respect to a particular relay-parent.

TODO: @lamafab: link reference The goal of a Candidate Backing Job is to produce as many backable candidates as possible. This is done via signed Statements by validators. If a candidate receives a majority of supporting Statements from the Parachain Validators currently assigned, then that candidate is considered backable.

on startup

- Fetch current validator set, validator -¿ parachain assignments from runtime API.
- Determine if the node controls a key in the current validator set. Call this the local key if so.
- If the local key exists, extract the parachain head and validation function for the parachain the local key is assigned to.

on receiving new signed Statement

```
if let Statement::Seconded(candidate) = signed.statement {
  if candidate is unknown and in local assignment {
    spawn_validation_work(candidate, parachain head, validation function)
  }
}
```

spawning validation work

```
fn spawn_validation_work(candidate, parachain head, validation function) {
  asynchronously {
    let pov = (fetch pov block).await
    // dispatched to sub-process (OS process) pool.
    let valid = validate_candidate(candidate, validation function, parachain head, pov).await;
    if valid {
      // make PoV available for later distribution.
      // sign and dispatch `valid` statement to network if we have not seconded the given candidate.
    } else {
      // sign and dispatch `invalid` statement to network.
    }
  }
}
```

fetch pov block

Create a `(sender, receiver)` pair. Dispatch a `PovFetchSubsystemMessage(relay_parent, candidate_hash, sender)` and listen on the receiver for a response.

on receiving CandidateBackingSubsystemMessage

- If the message is a `CandidateBackingSubsystemMessage::RegisterBackingWatcher`, register the watcher and trigger it each time a new candidate is backable. Also trigger it once initially if there are any backable candidates at the time of receipt.
- If the message is a `CandidateBackingSubsystemMessage::Second`, sign and dispatch a `Seconded` statement only if we have not seconded any other candidate and have not signed a `Valid` statement for the requested candidate. Signing both a `Seconded` and `Valid` message is a double-voting misbehavior with a heavy penalty, and this could occur if another validator has seconded the same candidate and we've received their message before the internal seconding request.

TODO: send statements to Statement Distribution subsystem, handle shutdown signal from candidate backing subsystem.
TODO: subsystems for gathering data necessary for block authorship, for networking, for misbehavior reporting, etc.

2.7 Data Structures and Types

TODO: ...

- CandidateReceipt
- CandidateCommitments
- AbridgedCandidateReceipt
- GlobalValidationSchedule
- LocalValidationData (should commit to code hash too - see Remote disputes section of validity module)

Block Import Event

```
/// Indicates that a new block has been added to the blockchain.
struct BlockImportEvent {
  /// The block header-hash.
  hash: Hash,
  /// The header itself.
  header: Header,
  /// Whether this block is considered the head of the best chain according to the
  /// event emitter's fork-choice rule.
  new_best: bool,
}
```

Block Finalization Event

```
/// Indicates that a new block has been finalized.
struct BlockFinalizationEvent {
  /// The block header-hash.
  hash: Hash,
  /// The header of the finalized block.
  header: Header,
}
```

Statement Type

```
/// A statement about the validity of a parachain candidate.
enum Statement {
  /// A statement about a new candidate being seconded by a validator. This is an implicit validity vote.
  Seconded(CandidateReceipt),
  /// A statement about the validity of a candidate, based on candidate's hash.
  Valid(Hash),
  /// A statement about the invalidity of a candidate.
  Invalid(Hash),
}
```

Signed Statement Type

The actual signed payload should reference only the hash of the CandidateReceipt, even in the `Seconded` case and should include a relay parent which provides context to the signature. This prevents against replay attacks and allows the candidate receipt itself to be omitted when checking a signature on a `Seconded` statement.

```
/// A signed statement.
struct SignedStatement {
  statement: Statement,
  signed: ValidatorId,
  signature: Signature
}
```

Overseer Signal

Signals from the overseer to a subsystem to request change in execution that has to be obeyed by the subsystem.

```
enum OverseerSignal {
  /// Signal to start work localized to the relay-parent hash.
  StartWork(Hash),
  /// Signal to stop (or phase down) work localized to the relay-parent hash.
  StopWork(Hash),
}
```

Candidate Backing subsystem Message

```
enum CandidateBackingSubsystemMessage {
  /// Registers a stream listener for updates to the set of backable candidates that could be ba
  /// in a child of the given relay-parent, referenced by its hash.
  RegisterBackingWatcher(Hash, TODO),
  /// Note that the Candidate Backing subsystem should second the given candidate in the context
  /// given relay-parent (ref. by hash). This candidate must be validated.
  Second(Hash, CandidateReceipt)
}
```

Host Configuration

The internal-to-runtime configuration of the parachain host. This is expected to be altered only by governance procedures.

```
struct HostConfiguration {
  /// The minimum frequency at which parachains can update their validation code.
  pub validation_upgrade_frequency: BlockNumber,
  /// The delay, in blocks, before a validation upgrade is applied.
  pub validation_upgrade_delay: BlockNumber,
  /// The acceptance period, in blocks. This is the amount of blocks after availability that validators
  /// and fishermen have to perform secondary approval checks or issue reports.
  pub acceptance_period: BlockNumber,
  /// The maximum validation code size, in bytes.
  pub max_code_size: u32,
  /// The maximum head-data size, in bytes.
  pub max_head_data_size: u32,
  /// The amount of availability cores to dedicate to parathreads.
  pub parathread_cores: u32,
```

```
  /// The number of retries that a parathread author has to submit their block.
  pub parathread_retries: u32,
  /// How often parachain groups should be rotated across parachains.
  pub parachain_rotation_frequency: BlockNumber,
  /// The availability period, in blocks, for parachains. This is the amount of blocks
  /// after inclusion that validators have to make the block available and signal its availability to
  /// the chain. Must be at least 1.
  pub chain_availability_period: BlockNumber,
  /// The availability period, in blocks, for parathreads. Same as the `chain_availability_period`,
  /// but a differing timeout due to differing requirements. Must be at least 1.
  pub thread_availability_period: BlockNumber,
  /// The amount of blocks ahead to schedule parathreads.
  pub scheduling_lookahead: u32,
}
```

Signed Availability Bitfield

A bitfield signed by a particular validator about the availability of pending candidates.

```
struct SignedAvailabilityBitfield {
  validator_index: ValidatorIndex,
  bitfield: Bitvec,
  signature: ValidatorSignature, // signature is on payload: bitfield ++ relay_parent ++ validator index
}
struct Bitfields(Vec<(SignedAvailabilityBitfield)>), // bitfields sorted by validator index, ascending
```

Validity Attestation

An attestation of validity for a candidate, used as part of a backing. Both the `Seconded` and `Valid` statements are considered attestations of validity. This structure is only useful where the candidate referenced is apparent.

```
enum ValidityAttestation {
  /// Implicit validity attestation by issuing.
  /// This corresponds to issuance of a `Seconded` statement.
  Implicit(ValidatorSignature),
  /// An explicit attestation. This corresponds to issuance of a
  /// `Valid` statement.
  Explicit(ValidatorSignature),
}
```

Backed Candidate

A `CandidateReceipt` along with all data necessary to prove its backing. This is submitted to the relay-chain to process and move along the candidate to the pending-availability stage.

```
struct BackedCandidate {
  candidate: AbridgedCandidateReceipt,
  validity_votes: Vec<ValidityAttestation>,
  // the indices of validators who signed the candidate within the group. There is no need to include
```

```
  // bit for any validators who are not in the group, so this is more compact.
  validator_indices: BitVec,
}
struct BackedCandidates(Vec<BackedCandidate>); // sorted by para-id.
```