Low Latency Parachains

Design

Authors: Robert Klotzner
Status: Draft
Approvers: Jan-Jan van der Vyver Andrei Sandu

Table of Contents

Introduction

For many use cases that go beyond pure speculation on price on exchanges, we have humans more or less directly interacting with the blockchain and here especially for more complex usage patterns (not just a single transfer), latency matters. Oftentimes perceived latency can be mitigated by UIs, but especially for highly interactive and dynamic use cases this is insufficient, in particular for situations where the success of the user initiated transaction depends on what others are doing. E.g. consider Defi or any form of bidding, where success is by no means guaranteed and depends on other bids coming in. For these scenarios we actually **need fast confirmation that a transaction was sequenced and achieved a particular result with respect to that sequence.** This implies having built an actual block.

Let's dive in.

Status Quo

Relay Chain

Parachains derive their security from the relay chain, but the relay chain has block times of 6s. On top of that getting a parachain block included takes 2 relay chain blocks. Together with the block building ahead of time (asynchronous backing), this results in a total latency of 18s, for any transaction that aims to be secured by Polkadot's economic security.

What makes matters worse is that at that point, all you got guaranteed was that your transaction sequence is valid, you don't yet have a guarantee that it will become canonical. The relay chain can experience forks and reorgs and a previously valid block can get reverted, resulting in collators having to rebuild their blocks, potentially arriving at a different (also valid) sequence.

Therefore, for our initial problem, the total perceived latency is even worse as one would need to wait for relay chain finality, which adds another 18s. Hence, the total perceived latency for a confirmed transaction sequence in a parachain block is around 36s. With improvements like moving from BABE to SASSAFRAS reorgs become way less likely, but nevertheless stay possible and even then the latency would still be 18s.

It becomes clear quite quickly that user-experience super-low latency confirmations can not be expected to benefit from the full economic security provided by Polkadot.

If we want to have a latency lower than 18s, we will need to provide user feedback earlier at a point that has reduced security guarantees, but still significant confidence. To determine the feasibility, let's look into collators.

Collators

Parachain blocks are initially produced by collators. Those are usually staked in the parachain's native token. These stakes can be rather low though as we only rely on collators for liveness. Meaning, a misbehaving collator can (in the current model) not cause any soundness issues. It can only effectively slow down the parachain.
To disincentivize this small risk, collators can have a much smaller stake than relay chain validators and having those collators lose out on rewards will be enough to ensure liveness.

Current model:

1. A single honest collator suffices to guarantee liveness of the parachain.
2. Collator based risk is only liveness, which is not much of an incentivization to misbehave. There is no security/soundness threat.

Together these two, justify to say that parachains derive economic security from the relay chain. For improving latency we will have to weaken at least one of them.

Goals

We would like to achieve:

1. Much lower latency. Instead of 36s, ideally we could get at least somewhat secured confirmations within seconds, for more centralized parachains even subseconds.
2. Still derive at least some security from the relay chain, even for those fast confirmations. Ideal: A single honest collator is enough to maintain low-latency security. Note how this differs from the existing model, where a single honest collator suffices to maintain liveness, which is an even weaker requirement.

Solution Overview

Observation: If we are aiming for super-low latency in confirmations, we can not allow forks!

Super-low 1-block latency means we need to have at least some "finality" guarantees that are economically enforced. Otherwise we are entirely relying on single actors being honest, which completely defeats our mission.

Consider Bitcoin as an illustrative example. It does allow forks, but they are expensive - like really expensive! Hence if someone created a fork, just to fool you, the gain to the block author from this fraud attempt would need to outweigh the cost for creating this fork. Users can then reason that they are very likely not cheated, as the cheater would lose money doing so. The confidence grows with time, as costs grow with the length of the fork. This is a common theme in blockchains: Confidence grows with time passed, which directly works against our low-latency plans.

Contrast this with our current system: Creating a fork (an equivocation) is factually free, thus there is no economic cost into trying to make some stakeholders believe something that won't become canonical.

Seeing a block containing some transaction **does not provide any cryptoeconomic guarantees!**

Thus our solution will work towards:

a. Avoid the need of honest parachain forks
b. Introduce cost (best case: slashes) for creating a fork

For this we are introducing:

1. Acknowledgement signatures
2. Some decoupling from the relay chain
3. Fast speculative messaging to make up for (2).

Acknowledgment Signatures

For the following sections to make sense, let's quickly introduce acknowledgement signatures. We will demand (and enforce) that any block produced by a collator is only allowed to be provided to backers if the author gathered acknowledging signatures from at least 50% + 1 of the collators.

In particular when a block author is done, it will ensure to have shared the entire block + header + an acknowledgement of his own (created in a way that it is different from the seal), but omitting its own authoring signature (the seal) in order to prevent fraudulent submission to the relay chain by others.

The other collators will then sign the header with their key and send that back to the block producer. Once the block producer has gathered enough signatures it will submit the block and reveal the seal.

To describe the process in a bit more detail: We assume set reconciliation/tx streaming to be in place here, all collators build in parallel. Steps happening roughly:

1. Block building with live set reconciliation/tx streaming: All collators put transactions in the block as they arrive.
2. Block author shares any additional information needed for a full block (but not the seal).
3. Once collators have all the data, they provide an acknowledgement signature, signing off on the state transition. These signatures are gossiped and also sent to other network participants.
4. Once the block author receives enough acknowledgements it shares the seal and publishes the full block to the network: This is where users get their low-latency

confirmation: They received the full block (or block data they are interested in) + all the acknowledging signatures.
5. Block author sends POV to relay chain validators.

Nodes will acknowledge a block if (and only if):

1. They deem the block valid, assuming an accompanying seal will be provided.
2. They received an acknowledgement of the alleged block producer. (So we have proof that the block is actually coming from the block producer.)
3. They have enough acknowledgements for all ancestry blocks, back until a parent that has been included on a finalized relay chain block.
4. No conflicting block (block with same parent, but different resulting state root) has already been acknowledged. (Checked back similarly until to the latest parent that has been included on a finalized relay chain block).
5. All those pending ancestry blocks are still valid. (Relay parent in scope, can still be submitted to the relay chain.) Any blocks and their signatures that fail to be seen on the relay chain in time will be discarded, together with all descendent blocks.
6. If the built block is a sibling of a previously retracted block (e.g. because of a relay chain fork), only acknowledge if the newly built block includes the transaction of that retracted block in the same order, as much as possible. Specifically: Removal of transactions that have become invalid is of course allowed. With this rule in place we also get some resilience even if not building on finalized relay chain blocks as this will be the case at least in early phases of the implementation.

Note that these acknowledgement signatures are off-chain data. They are provided to clients in addition to the block data they requested to convince them of some level of canonicalization with low-latency and are only put on chain to prove an equivocation.

These acknowledging signatures serve a few purposes:

1. They make handling of block withholding and “block not submitted” attacks possible.
2. They help avoiding accidental equivocations (e.g. an operator running two collator nodes).
3. By their very nature, they increase confidence in canonicality.

Opportunities:
The set reconciliation we are implementing for the transaction pool could potentially alleviate the privileged MEV position the block producer is in: The other collators can refuse to sign off the block if the produced block does not match their expectation.
Update: Actual set reconciliation seems much less important now, simple tx streaming should suffice for our immediate goals.

Equivocations

With acknowledgement signatures out of the way, let’s look into our definition of equivocations.

Definition

Usually equivocations are defined as: *Same author* authoring two blocks in the same slot, e.g. here. Our definition for this document is a bit different to account for elastic scaling, asynchronous backing and more importantly also other attacks than simple direct signing of multiple conflicting blocks.

Given two blocks, they should be considered equivocating if:

1. They have the same parent block.
2. They result in a different state root or different logs.
3. At least one of them got acknowledged via signatures by at least 50% + 1 of the collators and those acknowledgements are still valid.

For the requirement (2): By requiring block authors to never submit a parachain block to the relay chain if they have not gathered enough supporting signatures, we essentially made the whole network the “author” of the block and the network is guaranteed to be aware of any existing conflicting blocks: It is impossible to get 50% + 1 of the signatures for both blocks, without at least 1 of those signers having seen both. This allows specifically for settling block withholding and missed block submission “disputes”.

Reporting

If a collator were malicious and would present to a victim a different block than what they propose on the relay chain, then the victim can defend itself, by submitting an equivocation proof, consisting of:

1. Sealed header of block submitted to the relay chain.
2. Sealed header of block received.
3. Acknowledging signatures on one of these blocks.
4. Acknowledging signatures of the other block (if available).

The actual reporting happens in three phases:

1. Reporting: Someone reports two equivocating block headers. This now opens a challenge period (>=1 times the number of collators in slots).
2. Challenge: Within the challenge period, acknowledging votes for those two different state transitions are collected.
3. Punishment Decision

Punishments

For the punishment decision there are two scenarios:

1. Scenario: Two conflicting state transitions, each with enough (and still valid) acknowledging signatures.

2. Scenario: Two conflicting state transitions, one with enough acknowledging signatures, one without enough acknowledging signatures.

Scenario (1): All collators who acknowledged both (block authorship counts as implicit acknowledgement) get punished. Whether or not any of those blocks got included on the relay chain is irrelevant. The punishment here can be quite harsh as misbehavior due to force of circumstances or by accident can be ruled out with high probability: Slashes are feasible.

Scenario (2): A block that never received enough acknowledgements should never have been revealed to anybody, thus the very presence of such a header (with seal) can be punished. Useful exception to consider: If that losing block reported an equivocation itself (ideally autoconfirmed - already providing enough acknowledgements of one of these blocks), then we refrain from punishment. This is to allow even an honest minority to hold dishonest nodes accountable.

Punishments for scenario (2) have to be less severe and will come in the form of transferring era points: Era points of the offending collator are transferred to block authors coming after him (to make up for missed era points on their end).

Acknowledgement Validity

As explained in the acknowledgement signatures section, acknowledgements lose their validity at some point. This needs to also be enforced by the runtime:
Any provided acknowledgement votes for an out of scope state transition need to be discarded by the runtime: The runtime needs to be aware that they are out of scope, which should be trivial given that we have slot information of the block available in the header.

Covered Variants

Apart from a plain equivocation by a single block author in its slot also the following equivocation triggering behaviors are handled by above logic:

Avoid Block Submission

If a collator fails to submit an acknowledged block to the relay chain, this will also lead to an equivocation.

There are two ways to do this. One is in combination with tricking the user to believe it has synced the relay chain, when in fact he is on a very old state. If we then assume that equivocation reports can only come in up to a certain point, then this enable to get away with producing a fork based on that old state. There is an easy mitigation for this attack though: Clients should sanity check the relay chain block timestamps.

The other form is doing it live: Produce a block, show it to a client, but never submit:

Such a misbehavior can be detected once the not submitted block becomes invalid (relay parent out of scope) as then other collators will build and submit an alternative fork.

Once that alternative fork is included on the relay chain, an equivocation proof with the non submitted header can be submitted. Any acknowledgements for that non submitted proof will be invalid by now.

Threat model: A dishonest majority only has two choices going forward with avoiding block submission. They can either completely stall the chain or end up equivocating. Thus a single honest node is sufficient to ensure consequences for this kind of attack.

Block Withholding

A collator could choose to submit a block to the relay chain backers, but not reveal that block to his fellow collators. This would prevent those collators from building a canonical child block as the they can only either:

1. Don't build at all.
2. Build on the parent of the not revealed block.

In both scenarios they will miss out on rewards, because no block would become included on the relay chain. Block withholding is not directly an equivocation attack, but rather mere griefing. Acknowledging signatures do allow us to defend against these though: The next block author can once its slot starts, produce an equivocating block (preferably empty) and get that acknowledged by the network. With this in place the above punishment scheme can take care of the rest: Once availability can be recovered the collator can build a block that can be backed and will put an equivocation report in.

It has been raised to replace this scheme with just making acknowledgement signatures part of the block and indeed for this griefing attack this would seem to be a working solution too. Gains are low though as the actual equivocation attacks would still require the mechanisms described here.

Threat model: It is worth pointing out that this defense mechanism does require an honest majority as otherwise the collator getting attacked would not get enough signatures for its equivocating block.

As this is only a griefing attack it is also sensible to skip the equivocation part and just accept losing out on rewards, resorting to e.g. governance/social means to hold people accountable for such a behavior. This might be acceptable as for "avoiding block submission" handling we do actually consider being lenient with how long it can take to get a block submitted, but that results in similar consequences (reduced liveness) to the network. Not punishing here also improves censorship resistance.

Handle the Relay Chain

For maximum reliability we have to decouple parachain blocks as much as possible from the relay chain.
We need to fix two issues with the relay chain which force us to equivocate right now:

1. Forks: If the relay chain forks, we are also forced to fork in at least some way.
2. Collations becoming invalid/going out of scope and need to be rebuilt.

Forks

Requiring the parachain to not fork at all (without punishment) is a hard requirement, considering the forkyness of the relay chain itself. This is one reason why we restrict our punishments to transferring of era points: Equivocations due to relay chain forks would affect all collators similarly so those punishments should even out on average.

But in the above scheme relay chain forks will in any case reduce reliability of low-latency confirmations and will hamper throughput as collators will naturally avoid building on multiple forks to escape punishment. Thus on every relay chain fork we will see a downtime until acknowledgements become invalid.

Short Term Solution

Low-latency chains should build on “old” relay parents as this significantly reduces the chance of being affected by a relay chain fork. This ironically increases latency for messages between that chain and others and is limited by the max ancestry length restriction we impose on parachains. (Requirement: Block Based Collator)

Scheduling wise we can do that easily up to a depth of let’s say 5 and at that depth we are in a territory where finality usually will have kicked in. Thus we have very decent guarantees on canonicality. Nevertheless going to depths of 5 will short-term only be sensible to chains which don’t care too much about messaging latency.

Implementation:

- Block based collator
- Increase claim queue size
- Determine minimum depth to achieve good enough results

Long Term Solution

The short term solution is only applicable if the increased latency for communication is acceptable, if not we need to resolve that drawback: We do have ideas for speculative low-latency messaging, which works independently of the age of the relay parent.

Collations Going Out Of Scope

Even if we ignore the problem of added latency for messaging, the validity of a collation is limited by its used relay parent. If for some reason a collation is not included on the relay chain at the predicted block the collation will become invalid, causing performance degradation and enforces equivocations of some form as blocks need to be rebuilt.

Whether or not a collation can make it on the relay chain in time is largely outside of the control of a collator, which is an important reason for the rather mild punishment we were able to establish in the previous section. It is impossible to justify a slash, if there is a good chance of honest nodes getting slashed regularly.

Improvements in this area come in two forms:

1. Identify reasons for a collation not making it in time and improve/mitigate them as much as possible: This makes sense as it also ensures reliable throughput.
2. Make rescheduling possible, without forcing a new block: Decouple scheduling from blocks directly and give collators a chance to resubmit the same block in case something goes wrong.

Improve Submission Reliability

Ways to improve reliability, roughly sorted in terms of implementation complexity.

Speculative Availability

Reduce chances of availability timing out, by giving it more time. This is a relatively straightforward improvement explained here.

Backers getting DOSed

This is getting improved in 2025, by implementing the collator protocol revamp.

Fix Dropping of Candidates on Session Boundaries

Problem also explained here. The proposed solution there is hardly applicable for high reliability & low-latency chains. Just skipping an entire relay chain block, is a long time if you are promising 0.5s blocks. Thus the solution must be to remove the restriction that candidates can’t outlive a session. That being said, reliability trumps throughput. To get immediate reliability improvements we can consider the stop-gap solution of collators just not building at a session boundaries. This will at least halve the downtime on session boundaries and more importantly avoids any equivocations.

Offchain runtime upgrades + offchain messages

Reduce the chances of relay chain blocks getting too full, forcing the dropping of candidates. With those things offchain, the only reason remaining would be a dispute storm, which is very much an exceptional situation and can be ignored for practical matters.

Enable Rescheduling

If we separate scheduling on the relay chain from the actual parachain blocks, then we can allow for relay parents to stay valid longer, but more importantly we could resubmit the same block with updated scheduling information, provided the consensus algorithm gives us the time for that.

Wrapping Parachain Validation Function (PVF)

We gain quite a bit of flexibility if we add another layer. In particular we could have the PVF not be the actual state transition directly, but instead extend the existing wrapper function around it. The idea is to move out the following from the actual state transition:

1. Providing proofs for received messages.
2. Providing claim queue/backing group selection data: (Relay chain block used, core selector, claim queue pos)

Instead the state transition function would be modified to provide a list of (sender para id, message root) of messages that have been received to the outer PVF and the current block producer.

We will also need to pack the following additional data in the PoV (on top of block data and state proof):

1. Claim queue/backing group selection data (at least the relay chain block to use), core selector can come from the state transition, claim queue pos can be static. All of that signed by the collator..
2. Any proofs for received messages. Any message roots where no proofs are provided will cause the other PVF to trigger a “requires” UMP message, so the roots can be provided by those other chains via a “provides” UMP message at the same time.

With claim queue/backing group selection data being provided by the outer PVF we decoupled backing group assignments from the relay parent. This allows for a few things:

1. We can be more strict again and only allow most current leaves, instead of an implicit ancestry. Which means that we will only have one responsible backing group at any point in time, instead of multiple.
2. More importantly: It allows a parachain block to stay valid for longer, without causing issues. The relay parent used for the block no longer determines the backing group, nor the claim queue, hence it can be much older (e.g. finalized).

By factoring out proving of received messages we also make parachain block validity independent on how messages are provided. Live (via provides/requires), vs. via proving their existence in relay chain ancestry. In particular if we aimed for providing live, but then

the backing group did not provide our block in time, we can provide a new POV to the next backing group containing the exact same block (no equivocation), but instead of emitting a “requires” message, we provide a proof of existence of those messages.

Similarly by making backing group and core selection independent of the produced blocks, those things can change without the need to create a new block, thus again avoiding equivocations. This also solves the problem of parachains changing cores at Agile Coretime region boundaries.

Apart from changing the PVF itself, an additional element is the actual resubmission logic. We need to track our collation on the relay chain (or more specifically the earliest still pending collation, which could be any of our parents too). If we don’t see it becoming backed and included within `k` relay chain blocks, we need to rebuild our collation (using the same block again). This will be done by all collators having a pending collation, so a complete chain of blocks (whole prospective parachain) will get resubmitted. Rough flow:

1. Submit collation to backers (any problems here are handled immediately).
2. Follow relay chain and keep track of included candidates.
3. Once our candidate or any parent did time out (was not included in time), we rebuild our collation with updated scheduling information.
4. All collators will detect such an event and slow down block production/halt it until the unincluded segment has a reasonable size again.

Block Based Collator

With a variant of the block based collator parachains gain more flexibility in trading equivocation soundness vs liveness. In particular, we would change the formulas of the block based collator to the following:

1. `n >= s + 1 && n < s + l` -> `k := s + 1, p = s + i`
2. `n == s && i < v -> i := i + 1 ; i >= v -> k := s + 1`
3. `n < s` -> illegal
4. `n >= s + l` -> `k := s + l`

With a configurable parameter `l` for lenience, allowing a block producer to keep trying providing a collation for `` `l` `` relay chain blocks. Note, `l` should be chosen such that `l % num_collators != 0` to avoid that rule (3) leads to the same collator being in charge again.

`n` will be the relay parent chosen by the collator. To become immune to forks it should not be the most recent one, but rather an ancestor. Individual collators could choose to ignore that rule and build on the most recent relay parent regardless, but by doing so they risk equivocating and the mistake can be corrected by the collators coming afterwards (they can reuse the same relay parent, waiting for it to get at the desired depth if necessary). It is worth pointing out that this is a harmless behavior: The collator collating too early only risks equivocating himself, but for the collator coming after him they only need to do the “waiting” that should have been done by that misbehaving node. The net effect is exactly the same.

With large enough values for `l` it should be possible to slash for equivocations, which could drastically increase the usefulness of low-latency confirmations.

The block based collator achieves two things:

1. It actually makes resubmitting the same block feasible (as the next block author can't yet interfere). So it is necessary to make the wrapping PVF useful.
2. It is a requirement to make building on older relay parents feasible, so is also needed for the fork handling in the previous section.

Opportunities: While the block based collator is trading reliability for liveness, the effects on liveness can actually be minimized by introducing secondary block producers, which is "forklessly" possible with acknowledging signatures. Essentially acknowledging collators will prefer the primary producer, but will after some time also acknowledge the block of a secondary producer. We are not introducing forks here, as even with network latency 50% + 1 acknowledgements are only possible for one of those blocks, without anybody equivocating.

Note that this only helps for the honest scenario where a collator just goes down. A malicious collator could still simply refuse to submit its collation which already got acknowledged. Others could also not submit the block on his behalf, because it might not have shared the seal either. This kind of attack is unlikely though as it is pure griefing: Without revealing the seal the collator won't trick anybody into taking the block for any kind of confirmation, so its effect will only be causing a down time.

Threat model: As equivocation reporting blocks do not require acknowledging signatures, there is no change in the threat model. A single honest node can still submit an equivocation, despite the network maliciously preferring a secondary block. There is one caveat though: If the secondary block producer is not prevented by the runtime to immediately jump in, it might just be faster than the primary block producer to get its block backed. TL;DR: If we consider this opportunity worth it, some more thought should go into it.

Considerations

1. We assume collator networks are relatively small and well connected. All collators signing off every block has to be cheap & fast enough. This is a trade-off any parachain aiming for low-latency has to accept: Low-latency comes at a decentralization cost. Which in fact **should be carefully considered** whether that is an acceptable trade-off for something as crucial to Polkadot as **Polkadot Hub**!
2. We assume collators are staked and have something to lose. An assumption, that to my knowledge, **does not fully hold** for a system chain like **Polkadot Hub**. (Note: Most punishments are only reducing rewards by now, so this might be less of a concern.) More details here .
3. A low-latency parachain gets stalled as soon as more than 50% of the collators go offline. In general: We now need 50%+1 honest collators to keep the parachain alive: It might make sense to implement a fallback mechanism, where after some timeout

normal “high-latency” block production gets enabled, to resume a stuck chain. (Disabling of acknowledgement signatures and equivocations)

4. We are not only trading liveness, but also censorship resistance. If there is a dishonest majority they can refuse to sign off a particular collator’s block. It is not perfect censorship as the collator can submit its collation, but it will risk punishment doing so. This can be solved, by punting on punishing on block withholding (which is not an equivocation).
5. Crypto-economic guarantees provided are still on the low-end as long as we are not confident enough (yet) in relay chain inclusion to allow for slashes. Nevertheless, practically speaking, the acknowledgment signatures on their own add quite a bit of confidence: As long as we assume the collator set is sufficiently decentralized, which would need to be achieved via other means, such that you won’t have enough nodes to collude for acquiring malicious 50%+1 acknowledgements.