Collaborative Based rollups

This post outlines a simplistic mental model of blockchains and rollups and articulates the DA bottlenecks that exist for all types of based rollups. It presents 4 solutions that rollups can use to decrease their block time, two of which allow them to become based rollups.

If a rollup can overcome these bottlenecks, it should consider becoming a based rollup.

Shout out to @LHerskind who wrote 90% of this but is on a plane so I front-run his posting

Let's start with a definition:

A Collaborative Based Rollup:

a based rollup where the inputs to the state transition function are always derivable from published data, allowing anyone to produce the proofs and for there to be common trustless understanding of what is the canonical chain (even pending blocks).

For any decentralised rollup looking to become based, ensuring block building is collaborative is a pre-requisite. To understand why, we first need to go back to how blockchains and rollups work.

What is a blockchain?

ELI5 Version

- A blockchain is a replicated state machine
- State is replicated by agreeing on the genesis \mathcal{S}_{genesis}

, the >transition function \mathcal{T}

and history of the inputs \mathcal{I}

- State must be recoverable to synchronize new nodes to the network
- This is achieved by having history available or with verifiable snapshots
- We need the state \mathcal{S}

to compute the next state \mathcal{S}'

Detailed Version

A blockchain is a replicated state machine that allows a lot of computers to agree on some state by applying an agreed upon state transition function to agreed upon inputs (or blocks). We can call this state transition function for \mathcal{T}

and defined as \mathcal{T}(\mathcal{S}, \mathcal{I}) \mapsto \mathcal{S}'

, where \mathcal{I}

is the input, \mathcal{S}

the initial state and \mathcal{S}'

is the state after the execution.

graph LR I --> T S --> T T --> S'

To get the same state S'

replicated among all the machines in the network, we "simply" need to ensure that all the computers start with the same initial state \mathcal{S}

(genesis) and that they are all applying the same transition function \mathcal{T}

to the same input data \mathcal{I}

.

If we then keep doing this, feeding the output \mathcal{S}'

in as the start state for the next execution along with the next chunk of \mathcal{I}

, our state machine is progressing while being consistent across the set of machines.

While we can agree on \mathcal{S}

and \mathcal{T}

ahead of time and keep those constant (until hard forks

, but lets save those for another time), we need to continuously extend \mathcal{I}

with new data to have a system that is actually useful - a never changing state is not particularly interesting.

Extending \mathcal{I}

needs to be done in such a way that the network is consistent, e.g., its nodes all have the same \mathcal{I}

. In practice, this can be done by agreeing on new chunks and then adding them. Agreeing on extensions to \mathcal{I}

is what is referred to as consensus, we come to consensus on the history of the network.

Since we are computing the current state \mathcal{S}'

by applying all of \mathcal{I}

to the genesis, a new computer that wants to join the network (become a node) can simply download the history (\mathcal{I}

) and then apply it to the genesis, arriving to the same state as the rest of the network! This is what happens when a full-node is syncing.

We say that you can synchronize a new node to a blockchain if its history is available. That the history is available is sometimes referred to as data availability

and data ordering

. If the history was not ordered, it would be of little use to new nodes as they would not end up with the same state as the rest of the network. The same is the case if the history is only partially available.

Most blockchains will have nodes in their networks that are happy to provide newcomers with the necessary data.

With all this talk about the history we must not forget that it is just a means to an end - we use the history to agree on the state of the network!

Since consensus protocols adds overhead every time we have to run them, we have a bright idea

.

Instead of applying it for every small input (transaction), we can bundle them into a block and then come to consensus on the that instead. This way we reduce the number of consensus executions we have to make.

These blocks often have a bit of extra information such as producers and some reward, so there will in practice be two state transition functions, a larger \mathcal{T}

which takes a block, and it will then apply some checks and then run \mathcal{T}_{inner}

on each of its transactions.

Variations on \mathcal{T}

and \mathcal{I}

While we are always using a state transition function to progress state, there are some quite big variations to what exactly it is doing and what the input is.

To showcase the differences, lets take a look at Ethereum

and ZCash

where the transactions that make up the inputs are quite different.

Ethereum example

In Ethereum

, the input \mathcal{I}

is a execution request

(\mathcal{E}

) and its signature (\sigma

).

It is a message telling the network that a specific account would like to execute a piece of code on its virtual machine. Essentially, that it want to execute the state transition function \mathcal{T}_{inner}

with its execution request as input on top of its current state.

The execution will apply some changes to the state. These changes are referred to as the state-diff or the effects of the transaction.

Common effects could be decrease its own balance and increasing some other balance, as part of a transfer or paying some fee.

\mathcal{T}_{inner}\left(\mathcal{S}, (\mathcal{E}, \sigma)\right) \mapsto \mathcal{S}'

ZCash example

In ZCash

, the input \mathcal{I}

is quite different!

As the goal is to be private it does not make sense to publish the request directly - if you tell everyone what you are going to do it is not very private.

When the user wants to produce his transaction, he will use a prover \mathcal{P}

along with his execution request and signature.

If the execution request can be executed correctly, and the signature is valid, the prover produces some outputs \mathcal{O}

and a proof \pi

.

\mathcal{P}(\mathcal{E}, \sigma) \mapsto \cases{\mathcal{O},\pi \qquad \text{if correct execution} \ fail \qquad \text{otherwise}}

We simplified here to make it easier to follow, in practice public and private inputs are provided, but this is easier to understand (I hope).

A verifier \mathcal{V}

can then be used to check that the request was correctly executed without even knowing what the request was

.

\mathcal{V}(\mathcal{O}, \pi) \mapsto \cases{accept \qquad \text{if correct execution} \ reject \qquad \text{otherwise}}

As ZCash are privacy preserving, the output \mathcal{O}

will consist of the effects(state diff) (\Delta

) of the transaction. These effects will consist of commitments, nullifiers and some additional data that the recipient can use to learn about the content of the commitments from.

To anyone else than the recipient, the effects \Delta

looks like a collection of random bytes.

While the proof \pi

may convince us that the effects is valid in the sense that they are generated from the execution request, it might not be a valid state transition anyway!

The most well known reason could be that you are sending the same transaction multiple times - trying to perform a double-spend!

Double-spends are caught by making sure that nullifiers have not been seen before (no duplicates allowed).

\mathcal{T}_{inner}\left(\mathcal{S}, (\mathcal{O}, \pi)\right) \mapsto \mathcal{S}'

While these are still fairly similar at a high level, (inputs are thrown into a state machine), the differences in input \mathcal{I}

can greatly impact the properties of the system.

TL;DR:

- The inputs will generally be execution request
- signature or effects
- proof
- Multiple transactions are generally bundled into blocks to reduce overhead of consensus
- We need the state \mathcal{S}

to compute the next state \mathcal{S}'

Note

:

General programmability chains can implement state transitions similar to ZCash

as a smart contract to support some privacy features.

What is a Rollup

A rollup is essentially a blockchain that is posting its inputs \mathcal{I}

and occaisional state roots \chi(S)

to a parent blockchain (the baselayer).

In this way, it is paying the baselayer for handling the data availability and ordering concerns.

Similar to how blockchains grouped transactions into blocks for consensus efficiency, rollups group them to reduce overhead costs per transaction.

The overhead depends on what type of rollup we are dealing with, so lets take a look into the types in a few moments.

Beyond pushing data to the baselayer, most rollups also have a bridge which is essentially a validating rollup light node that allows the base-layer to learn about the state of the rollup - allowing the two blockchains (rollup and baselayer) to communicate.

From the view of the validating bridge, the state transition function is slightly altered as a root of the state is inputed instead of the full state. This is an optimization as we cannot provide the full state \mathcal{S}

at every execution. We denote the commitments to the states as \chi(\mathcal{S})

and \chi(\mathcal{S})'

.

Simplifying a bit, we alter the function to:

\mathcal{T}(\mathcal{S}, \mathcal{I}) \mapsto \mathcal{S}' \quad \Rightarrow \quad \mathcal{T}(\chi(\mathcal{S}), \mathcal{I}) \mapsto \chi(\mathcal{S})'

With the new formula at hand, we expand a bit to have a general setup. Let \chi(\mathcal{S})

be a value stored in the validating bridge. We can then describe a block \mathcal{B}

along the lines of.

\mathcal{B} = \left(\chi(\mathcal{S}),\chi(\mathcal{S})', \mathcal{I} \right)

With the bridge updating its state as:

\chi(\mathcal{S}) \stackrel{?}{=} \mathcal{B}*{\chi(\mathcal{S}))} \quad \wedge \quad \mathcal{T}\left(\chi(\mathcal{S}), \mathcal{B}*{\mathcal{I}} \right) \mapsto \mathcal{B}*{\chi(\mathcal{S}')} \quad\Rightarrow\quad \chi(\mathcal{S}) \mapsto \mathcal{B}*{\chi(\mathcal{S}')}

We don't need

to store \chi(\mathcal{S})

as part of every block as we can get it from the parent, but it is just a nice way to simplify the model mental slightly, using it as the reference itself.

Optimistic Rollups

Briefly described, an optimistic rollup is a rollup where new blocks are proposed, with the assumption that the blocks and state transitions are valid.

As blocks are published on the baselayer a challenge period is started and the rollup nodes pick it up and execute it on their local view of the state.

If the rollup node's local view disagree with what it saw purported on the baselayer, it can challenge the block with a "fraud-proof". This challenge is usually executed using some kind of bridge, and will "slash" the fraudster and roll back the pending chain.

As long as there is no fraud committed, the rollup operates with low overhead, and is mainly paying the baselayer for data available and not execution.

Progressing State

:

To progress the state of the optimistic chain, you collect a bunch of signed execution requests (transactions) in a block, simulate their impact, and then add the block to the end of the current pending chain.

If the block added is invalid, fraud proofs will be posted and the block is rolled back and the sequencer slashed.

If valid, the state of the chain is progressed, and the block will become unchallengable after the challenge period have ended.

From the point of view of the rollup nodes there is essentially no delay from the point of publication to state progression and they are able to build on top right ahead.

We could outline the individual block input like below

\mathcal{B}*{\mathcal{I} = \left{\mathcal{E}, \sigma\right}*}{tx \in txs}

Validity Rollups

A validity rollup (or pessimistic rollup) is different to the optimistic, in the sense that it want proof that something good happened before it will allow a block to become part of a canonical chain.

It is following the premise that lack of proof of fraud is not the same as lack of fraud.

Validity rollups are taking the proof idea that we saw earlier from ZCash, and using it to prove correct execution without the privacy. Essentially it "compresses" the amount of work needed to validate

that some state is really the state after applying the transaction inputs.

While we in the ZCash example were producing a proof for every single transaction, we could also produce a proof for a collection of transactions

(rolling them up).

As we will see in a moment, the exact value of \mathcal{O}

differs between variations of validity rollups.

Common for the variations however, is that a block will only be added to the canonical chain if the proof \pi

is accepted.

Since the proof \pi

needs to be produced and then validated on the baselayer to progress the view of the bridge it have a higher cost overhead than the optimistic from earlier.

Signed Execution Requests

The first case one might think of is using the signed execution requsts as the “output” \mathcal{O}

similar to optimistic rollups and Ethereum.

\mathcal{P}(\left{\mathcal{E}, \sigma\right}*{tx \in txs}) \mapsto \cases{\mathcal{O}=\left{\mathcal{E}, \sigma\right}*{tx \in txs} ,\pi \qquad \text{if correct execution}\ fail \qquad\qquad\qquad\qquad \text{otherwise}}

As the signed execution requsts and previous state is everything that is needed to generate the proofs but also to learn the new state, it is possible to produce a proof only from the data published, and anyone that wish to extend the chain can themselves check that it is valid before the build.

This allows for a clean separation of sequencing and proving!

If publishing just \mathcal{O}

any sequencer wanting to build on top of your block can entirely validate that it can be proven and don’t need to wait for the proof \pi

for that specific block to be generated.

This means that the proof can come later

, and even span multiple blocks

. As the proof validation where the extra overhead beyond data publication this amortizes the validation cost and can in turn reduce the cost of individual transactions.

\mathcal{B}\mathcal{I} = \left(\left{\mathcal{E}, \sigma\right}*{tx \in txs}, \pi \right)

Execution Requests

A variation of the above is to publish only the execution requsts (and not the signature) but require that the proof is only containing execution requsts that have valid signatures on them.

It might sound super weird, but is similar to what we had in the ZCash example with some inputs being hidden.

It is therefore possible to build the proof only if you have the valid signatures, but without needing to broadcast these on the baselayer - reducing the data overhead.

\mathcal{P}(\left{\mathcal{E}, \sigma\right}*{tx \in txs}) \mapsto \cases{\mathcal{O} = \left{\mathcal{E}\right}*{tx \in txs} ,\pi \qquad \text{if correct execution}\ fail \qquad\qquad\qquad\quad \text{otherwise}}

However, this pruned nature means that sequencing and proving is less independent!

Namely, while it is possible for someone to build on top of another block without it being proven, they will not know before the proof if it is actually

a valid block.

After all, they don’t know if the signatures are valid. A sequencer trusting an earlier sequencer and building on that chain might end up building on a fork that will never make it into the canonical chain - wasted work!

\mathcal{B}\mathcal{I} = \left(\left{\mathcal{E}\right}*{tx \in txs}, \pi \right)

Effects (StateDiff)

A potentially

more “compressed” version of the inputs is to have just the state differences (the effects) of the entire block as the public inputs and then have all the signed execution requsts as private inputs into the circuit.

This way, if the same applications are used a lot, continious changes to the same state variables will simply be “summed” into just one state update.

\mathcal{P}(\left{\mathcal{E}, \sigma\right}_{tx \in txs}) \mapsto \cases{\mathcal{O} = \Delta,\pi \qquad \text{if correct execution}\ fail \qquad\qquad\qquad\quad \text{otherwise}}

\mathcal{B}_\mathcal{I} = \left(\Delta, \pi \right)

This have the same issue as the execution requsts only case.

Privacy Rollups

Privacy rollups are quite interesting, the combination of execution requsts + signature can essentially be replaced with state-diff + proof.

\mathcal{E}, \sigma \quad\Rightarrow\quad \Delta, \pi

While the individual transactions have zero knowledge proofs, the rollup as a whole can fall in either the validity and optimistic camp.

So for every transaction the user themselves will be producing the privacy preserving proofs as we saw earlier:

\mathcal{P}(\mathcal{E}, \sigma) \mapsto \cases{\Delta,\pi \qquad \text{if correct execution} \ fail \qquad \text{otherwise}}

A collection of these transactions will then be used to produce a block, of which we will see some variations shortly.

txs = \left(\left{\Delta, \pi\right}_{tx \in txs} \right)

This could potentially be enough for the input in an optimisitc rollup, but posting all of these transactions can be a huge burden, after all, these can be significantly larger than stuff like a signature

Private Optimistic Rollup

This case is very similar to the signed execution request we saw from earlier.

\mathcal{B}*mathcal{I} = *left{\Delta, *pi*right}*{tx \in txs}

Private Validity Rollup - effects and proofs

This case is very similar to the signed execution request we saw from earlier.

\mathcal{P}(\left{\Delta, \pi\right}*{tx *in txs}) *mapsto *cases{*mathcal{O}=*left{\Delta, *pi*right}*{tx \in txs} ,\pi_{\mathcal{B}} \qquad \text{if correct execution}\ fail \qquad\qquad\qquad\qquad\quad \text{otherwise}}

As earlier, if I know \mathcal{O}

, I am able to build a block on top of the current block. Sequencing of transactions and proving of the aggregate proof can occur independently of each other.

\mathcal{B}*mathcal{I} = *left(*left{\Delta, *pi*right}*{tx \in txs}, \pi_{\mathcal{B}} \right)

Private Validity Rollup - effects and aggregate proof

This case is very similar to the validity rollup variations using the execution requests without the signatures - namely just the diff and an aggregated proof instead.

Since the individuasl proofs can be fairly large, we might wish to aggregate them into one larger proof that we are posting before we are really moving along.

\mathcal{P}(\left{\Delta, \pi\right}*{tx *in txs}) *mapsto *cases{*mathcal{O}=*left{\Delta*right}*{tx \in txs} ,\pi_{\mathcal{B}} \qquad \text{if correct execution}\ fail \qquad\qquad\qquad\qquad \text{otherwise}}

As earlier, we are unable to “trustlessly” build on top of the proposed block until the proof \pi_{\mathcal{B}}

have arrived.

\mathcal{B}*mathcal{I} = *left(*left{\Delta*right}*{tx \in txs}, \pi_{\mathcal{B}} \right)

Aggregated Diff + proof

As the private transactions will be consisting of unique data there is no “overlapping” state diff changes.

Trustless build ahead

Regardless of the type of rollup, data that needs to be published in order for decentralised sequencers to “build ahead” of the finalized state and fully seperate sequencing and proving.

This reduces the overhead costs of validating proofs, and considerable improves UX as users can act on yet to be proven state.

Danger

Centralized rollups (of any kind) get this separation for free as they only need to trust themselves to continue building.

However, they are boring

, and not really an option for privacy focused rollups where cencorship is a major threat.

For optimistic rollups, this requirement is satisfied via posting data to L1 as seen in production today. There are questions on the size of the required data to validate a transaction, but largely it is well understood. For optimistic private rollups the data will be larger than a public rollup.

For validity rollups, there are two paths forward:

1. Validate the transactions individually by posting individual transaction proofs (32kb).
2. Wait on an aggregated proof to be provided.

TLDR; private rollups (optimistic or validity) need to post more data to enable trustless build-ahead, or they will have to delay building ahead until a validity proof is constructed.

The following table compares the size of the data needed for public and private rollups.

NB we are using Aztec as our model for a private rollup as this allows us to give accurate estimates for client side transaction proof sizes.

Variation

Needed for build ahead

Approx size per TX

Txs in a blob

Public

\left{\mathcal{E}, \sigma\right}_{tx \in txs}

\mathcal{E} +\sigma \approx 400 \mathsf{B}

\approx 300

Private

\left{\Delta, \pi\right}_{tx \in txs}

\Delta \approx 1.5\mathsf{KB}, \pi \approx 32\mathsf{KB}

\approx 4

\mathsf{B}

is bytes, and \mathsf{KB}

kilo bytes.

Transaction sizes

Transaction sizes are based off a typical Ethereum transaction. E.g a USDC transfer is \approx 120\mathsf{B}

while uniswap V3 swaps can span from 700bytes to 1.5KB and v2 swaps around 400 bytes. We assume ~400 bytes for use here to cover generally.

For private transactions the size is based of a typical Aztec transaction. This transaction will create 4 notes with encrypted preimages, emitting 4 nullifiers and altering 4 public storage slots.

Required D/A for liveness

The data listed in the table is required for the sequencer of block n+1

to be able to build on block n

, but does not necessarily convince L1 of the validity of the state transition.

Thereby, it could be acceptable to use a separate DA layer to publish this information additionally to the baselayer as a way to provide quicker block building at an increased cost.

An example: a rollup uses Ethereum for DA and runs a validating bridge but also posts a portion of the data to Celestia such that sequencers can build ahead on the chain quickly.

In this way, the rollup is a pure ethereum rollup from a security perspecitve, but is using Celestia for some “off-chain” collaboration.

The Status Quo on Aztec

The status quo is that Aztec will adopt the Fernet sequencing algorithm alongside SideCar to facilitate decentralised block production, resulting in a ~10 minutes between state updates, and finality.

However below we outline options to improve our block time, 2 of which are compatible with becoming a based rollup.

Why the long block time?

As described previously, a rollup is effectively a blockchain that executes a state transition function over a given set of inputs. In Aztec’s case the inputs consist of:

1. User transaction proof
2. User transaction state diff (commitments and nullifiers)

Fernet and SideCar are designed to reduce the amount of data posted to D/A by recursively aggregating transaction proofs into a rollup proof. The rollup proof is verified on chain and proves the validity of a block.

This circumvents the need to publish individual transaction proof data, but requires that a rollup proof be constructed and verified before the chain can advance, resulting in the ~10 min block time.

Is it possible to advance the chain without aggregating and proving transaction proofs?

Maybe, but there are risks. A sequencer commits to a blocks contents before a state transition occurs, which enables a malicious sequencer to stall the chain if they commit to a blocks contents and don’t reveal sufficient data for the next sequencer to validate the state transition can be proven. This will stall the chain.

To give an example: a sequencer can commit to a a new state root for a block with 4 transactions blockStateRoot(Tx1,Tx2,Tx3,Tx4)

, but only reveal a subset Tx1, Tx2, Tx3

. Without the data of Tx4

(Proof + State Diff), the entire chain will stall as an aggregate proof can’t be generated.

Reducing Block Times for Rollups

There are four ways we know to reduce block times that apply to any rollup construction. Two of these ways result in a “collaborative” based rollup.

The goal of each solution is to make it harder or impossible for a malicious sequencer to not reveal data committed to in a block proposal, that is required for a state transition.

Non Based

1. L2 consensus network
2. Posting all the data to produce proofs to a D/A

Based

1. Realtime proving - Improving proof agregation speed to allow aggregated tx proofs to be posted to L1 faster
2. Economic consensus

L2 Consensus Network

This solution uses a staking network on the L2 running a BFT or similar consensus algorithm between participants. IMPORTANTLY

this consensus is not designed to agree on the result of the state transition but agree that sufficient data is available, such that the state transition function can be run.

Pros

State updates are possible as fast as consensus can agree that data is available (likely sub second).

Users receive fast pre-confirmations and state updates.

Cons

A lot of engineering work – the consensus algorithm needs to be proven inside the state transition function. For a zk rollup this requires proving consensus in ZK.

As transaction proofs are ~32kb, the networking requirements for consensus network participants are very high. This will centralise the staking set.

Real-time Proof Aggregation

This solution relies on advances in proving systems or proving hardware to reduce the time to aggregate proofs to less than an L1 block (12s).

Software Approach

Using a proving system that heavily utilises folding, it may

be possible to aggregate Aztec's Tx proof data sufficiently quickly to remove the need for it to be published.

Hardware Approach

Using a faster machines, GPU's or ASIC chips it is possible to reduce the block time with raw compute power, this requires several orders of magnitude advances before this is feasible.

Pros

It is certain that the state transition function can be run as a proof is already available.

Cons

Timeline / feasibility

Requires further cryptographic advances

Requires new un-trusted hardware (ASIC's)

New hardware adds additional centralisation concerns.

Posting all the data to D/A

This solution is the simplest but requires significant D/A bandwidth.

The solution is to ensure all of the data required to validate the state transition and build block N+1

is posted to the D/A layer. Sequencers can then check the data and decide if block N%

will be included and if it should be the base for block N+1

.

Where to post data

Blobs to the rescue?

Technically all the data Aztec requires can be posted to Ethereum. However as each individual TX proof is ~32kb. Posting the data would limit Aztec throughput to <1 tps and consume ALL of the current blob limit

This is not an acceptable outcome.

Alt-DA (e.g Celestia / Espresso)

It may be possilbe to post this data to an alternative D/A however that D/A would need to be able to convince L1 that the data is available at when the Aztec state transition is proven. (Similar to 4844). Given a goal of 10 minute finality this is costly.

Pros

Simple to build

Cons

Limited throughput using Ethereum

Slow finality if using AlT D/A as proof latency is usually hours.

Cost of D/A proof is unknown.

Applications security may reduce to that of the AlT D/A. (e.g a trading app issuing fast state updates via Alt-Da has worse security for un-finalised trades as the trade is acting on state that may never happen if the Alt D/A goes down).

Economic Consensus

This solution relies on a large slashable economic bond being posted to ensure that following sequencers have some mechanism to assume risk of building on the yet-to-be-proven chain.

Building the chain

To extend the chain, a sequencer needs to publish the public inputs required for proof verification (\Delta

from earlier) and put up an economic bond.

From the time of publication, the accompanying proof must be proposed within the next TIME_TO_PROVE

L1 blocks or the bond will be slashed.

As the \Delta

is insufficient to allow everyone to produce the proof in itself, the sequencer is either to prove it themselves, or share the information out of protocol. One way to share the information would be using only transactions that are disbursed publicly in the mem-pool already, allowing others to produce the proof as well.

A sequencer has two choices to make when they submit a block proposal:

1. What to build?
2. Extend the canonical fork
3. Extend a non-canonical fork
4. Create a non-canonical fork
5. Extend the canonical fork
6. Extend a non-canonical fork
7. Create a non-canonical fork
8. How much of the chain they assume proving responsibilities for:
9. The block they are proposing
10. The block they are proposing + N ancestors
11. The block they are proposing

12. The block they are proposing + N ancestors

There needs to be a protocol defined incentive for sequencers to assume proving risk. A sequencer will take on the “risk” or responsibility of another sequencers batch, if all the data is available. It also allows larger aggregate proofs to be submitted reducing costs.

The L1 rollup contract implements a simple fork choice rule:

- At any fork, pick the “oldest” block.
- “Oldest” here is defined as the block added first
- We greedily follow the oldest blocks
- “Oldest” here is defined as the block added first
- We greedily follow the oldest blocks
- The chain must not include a slashed block
- A block is slashed if it have not proven within TIME_TO_PROVE

and

it is in the canonical fork.

- A block is slashed if it have not proven within TIME_TO_PROVE

and

it is in the canonical fork.

The L1 contract keeps track of a FINALIZED_TIP

as well, which is the newest block that follow the fork choice and

is proven.

A bond can be reclaimed when:

- None of the blocks claimed responsibility for is slashed
- AND:
- All of the blocks claimed responsibilitiy for is proven
- Or the FINALIZED_TIP

have moved beyond the “fork”, e.g., you proposel did not make it into the canonical chain.

- All of the blocks claimed responsibilitiy for is proven
- Or the FINALIZED_TIP

have moved beyond the “fork”, e.g., you proposel did not make it into the canonical chain.

A proof of a “sub-chain” can be provided by anyone, and will, if accepted, mark its blocks as proven. If the “next” block in the pending canonical chain is proven, we can keep progressing the state until we get to unproven blocks.

Incentives

Instead of paying a sequencer a block reward if their block is proven and marked canonical within the allocated time, economic consensus stacks block rewards if a block is built on in a short amount of time (and thus the data is available). The incentives are as follows:

1. A sequencer has a cost to proposer a block in the form of DA fees etc
2. The sequencer will post a bond to be slashed if the block does not end up in the canonical chain within TIME_TO_PROVE

.

1. In the default case, the sequencer will only receive the fees within their block - no block reward.

2. The sequencer will receive a block reward if the next sequencer builds a block on top of theirs, faster than the TIME_TO_PROVE

. This reward decays each L1 block, and is cumaltive for each additional block that is built on top. * The sequencer can enter this case if they are also proposing the following blocks

- Or by posting data related to convincing around \Delta

separately, convincing the following sequencers to build on top of their block.

- Other sequencers might also do a risk assessment, and be convinced that they can build on top as you will likely not give up on your bond if sufficiently large.
- The sequencer can enter this case if they are also proposing the following blocks
- Or by posting data related to convincing around \Delta

separately, convincing the following sequencers to build on top of their block.

1. Other sequencers might also do a risk assessment, and be convinced that they can build on top as you will likely not give up on your bond if sufficiently large.

[

Screenshot 2024-03-29 at 14.38.14

2580×876 142 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/2X/f/f97cc8d199f48d748780f0b3800217b0eb417aab.png)

In the worst case, no sequencers are posting enough data for others to be convinced elsewhere, so one small block is proven every TIME_TO_PROVE

block with a high proof validation overhead (cost).

Conclusion

There are multiple ways to decrease Aztec block times from 10 minutes. Two of these solutions are compatible with the Ethereum Shared sequencing roadmap but need further analysis as to the trade-offs before committing to a path.