Integrating an Optimistic Rollup

This section will develop a model of the architecture of a typical optimistic rollup and describe how this architecture can be adapted to use Espresso. It can serve as a guide both for adapting existing rollups to integrate with Espresso and for designing new rollups intended to use the Espresso from genesis.

If you've already familiarized yourself with the Espresso architecture and you just want a quick integration checklist, you can skip ahead to the summary of changes.

Optimistic Rollup Architecture

This section develops a model of the architecture of an optimistic rollup by abstracting away most of the internal complexity, and focusing mainly on the interactions between the sequencer and other components. Thus, this model should be sufficient to guide the integration of a wide variety of optimistic rollups with Espresso.

Background

In this model, an **optimistic rollup** (**OR**) is a distributed **state machine** in which one or more **proposers** propose new state roots - the result of applying transactions to the current state - and optional **challenger**s verify the state root and challenge it in case it is incorrect. The state transitions are determined by applying the deterministic function which defines the rollup to a sequence of transactions, or state transition requests, starting from a known initial state. This sequence is usually represented as a sequence of **blocks**, where each block consists of an ordered list of many transactions.

The architecture model will aim to isolate the **sequencer**, the component of the system which determines this sequence. Modeling the sequencer's interactions with the rest of the system can shed light on how to integrate with Espresso.

Components

We model the rollup as a collection of four abstract components:

- The **sequencer** is responsible for batching transactions from various users into ordered *blocks*, and then committing to an order of those blocks. The order can be arbitrary or subject to rollup-specific constraints. The important thing is that the order is public and immutable: all rollup users at all times should agree on the relative ordering of blocks.
- The **proposer**'s job is to evolve the VM state by applying the sequenced transactions and commit to the new state by posting the new state root to the rollup contract. The result of evolving the state machine is expected to be deterministic: any party evolving the same state through applying the same ordered list of transactions with the transition function of the rollup must arrive at the same evolved state.
- The **challenger**'s job is similar to the proposer but instead of posting new state roots to the rollup contract it will challenge the state root in the rollup contract if it computes a different VM state than the proposer. The details of how the challenge process is implemented will differ from rollup to rollup.
- The **rollup contract**'s primary functions are to store the state roots, provide an interface for storing sequenced transactions, and provide a challenge mechanism. In reality this functionality is usually split up over multiple contracts.

This model is designed to make it easy to understand what needs to change when an optimistic rollup integrates with Espresso. For a fully functional product other services such APIs for clients to connect to (e. g. a JsonRPC server for EVM rollups) are required.

Transaction Flow

The figure below shows the interactions between these components as a transaction flows through the system. The dashed arrows indicate places where more than one design is possible, and different optimistic rollups may make different choices.

Generalized Optimistic Rollup architecture

1. The user submits a transaction. This will often be submitted to a rollup-specific service of the user's choice (such as a JSON-RPC server) before being forwarded to the sequencer. However, many rollups also allow the user to submit directly to the sequencer. In some rollups, the sequencer may even be combined with the rollup service, so that it can execute transactions before sequencing them and filter out invalid transactions.
2. The sequencer eventually includes the transaction in a block.
3. The block is sent to the rollup contract, where it is stored. This serves two purposes:
    - The contract can act as a source of truth for the order of blocks, so that if the sequencer later tries to equivocate, and report a different order to different participants, all parties can take the order saved in the contract as authoritative.
    - The layer 1 blockchain hosting the contract provides **data availability** for the rollup. Even if the sequencer goes down, users, proposers and challengers can read the data for each sequenced block from storage in the layer 1 blockchain, which is presumed to be highly available. This means that the sequencer cannot stop any participant from reconstructing the state of the layer 2 blockchain.
4. The proposer is notified of the new block. They may get it from the rollup contract, or directly from the sequencer. In the latter case, they will authenticate the block provided by the sequencer against the ordering (or a succinct commitment to the ordering) stored in the contract.
5. The proposer executes the block and updates its local copy of the VM state.
6. The proposer posts a commitment to the new state to the rollup contract.
7. The challengers notice a new state being posted to the rollup contract.
8. The challengers fetch the block and new state root from the rollup contract and use it to re-compute the new state root themselves.
9. If a challenger finds the state root in the rollup contract to be incorrect it initiates a challenge process. If successful it will obtain a reward and the proposer at fault will be punished.

10. The user observes the result of their transaction by querying the server provided by the rollup API. If the user does not trust this API they can choose to recompute the state locally (akin to what the proposer does), or wait until the challenge period has expired.

Using Espresso

This section describes the recommended architecture for an optimistic rollup integrated with Espresso.

Here we describe the architecture for an optimistic rollup using Espresso for confirmations, data availability, and sequencing. Rollups may choose to use a separate sequencer (centralized or decentralized) while still using Espresso for confirmations, with a similar system architecture.

Additions, deletions, and changes are highlighted in comparison with the typical optimistic rollup architecture.

Components

Generalized optimistic rollup architecture, modified to use Espresso

Espresso brings a few new components into the picture:

- **HotShot** finalizes blocks produced by the Espresso Network in a few seconds. This ensures that rollup transactions are confirmed quickly and in a decentralized and secure fashion, owing to the many nodes participating in HotShot.
- **Tiramisu** provides data availability for all rollup blocks. This availability is guaranteed by the same operator set that guarantees finality, so it is reliable, and rollups have the option of using Tiramisu as their standalone data availability layer if they wish. Tiramisu is designed to provide higher throughput and lower fees than using Ethereum for data availability.
- The Light client contract runs a HotShot client on the layer 1 blockchain, verifying the finality of each block and storing a binding commitment to the order of blocks decided by HotShot. As described in the following sections, rollup contracts on the same layer 1 can use this contract to relate their state transitions to the committed block order.

Transaction Flow

The transaction flow with Espresso is very similar to the transaction flow without. The main difference is in how confirmed blocks are disseminated. When not integrated with Espresso, rollup nodes are free to choose from where they fetch sequenced blocks: from the rollup contract on layer 1, or directly from the sequencer. Espresso, however, does not send sequenced blocks to each rollup contract: doing so would be expensive and would not meet the needs of all rollups. Instead, Espresso only sends sequenced blocks to one place—the sequencer contract—and it only sends a *commitment* to each block, at that: since Espresso provides its own data availability solution, the default is *not* to store entire blocks on the layer 1.

With this new flow, rollup nodes may choose how they get *notified* of new blocks: either by streaming them directly from Espresso or streaming events from the sequencer contract. But in either case, they must retrieve the block *contents* from the Espresso Network (see Espresso's

data availability API). Furthermore, rollups that still wish to use the layer 1 for data availability are responsible for sending the necessary data to the layer 1 once it has been sequenced.

It is important to note that while the transaction flow changes in terms of how transactions get sequenced (if Espresso is used for sequencing) and how rollups consume that sequence, there is no change in what a rollup does with a block *after* it is sequenced. This means that the execution layer, which makes up the bulk of the complexity of many rollups, can be used completely unmodified with Espresso.

The following sections describe how the components of the rollup must be modified to support the updated transaction flow and retain security.

Fraud Proofs in a Shared-Sequencer World

The bulk of the changes from the basic optimistic rollup architecture relate to the fraud proofs, requiring additions to the proposer, the challenger, and the rollup contract's challenge mechanism. While the details of challenge mechanisms vary widely between optimistic rollup instantiations, the general concept is universal: a proposer has sent a state root to the contract, and a challenger has claimed that the correct state root is something different. The challenger must then provide evidence that the proposed state root is incorrect, and the proposer may be given a chance to produce evidence defending their proposal.

There are two ways a state root can be incorrect:

1. The proposer executed the wrong transactions.
2. The proposer executed the right transactions incorrectly.

Therefore, any challenge mechanism must first check that the proposer and challenger both agree on the correct transactions to execute, according to the ordering determined by the sequencer. If they do not, then whichever one *has* executed the right transactions trivially wins the challenge. Once the proposer and challenger have agreed on the sequence of transactions being executed, the challenger must prove that the proposed state root is not the correct result of applying the state transition function to those transactions.

The first part, agreeing on the sequence of transactions, is generally a fairly straightforward check against the committed sequence. The second part, checking the proposer's computation of the state root, is where most of the complexity lies, since it involves proofs or interactive challenge games that encompass the low-level semantics of a complex VM. Luckily, this part of the challenge remains completely unchanged when integrating with Espresso, and only the mechanism for agreeing on the sequence of transactions needs to change.

With a single, centralized sequencer, reaching this agreement may only consist of a simple lookup, since the sequencer is sending blocks (or block commitments) directly to the rollup contract, and those blocks contain only transactions that are relevant to this particular rollup. Therefore the contract simply reads the last sequenced block directly from its own storage and compares it with the blocks claimed by the proposer and the challenger (or compares commitments, in some rollups). If the proposer's claimed block does not match the expected block, the contract can immediately consider the challenge successful. If the challenger's claimed block does not match, the contract can immediately reject the challenge.

Things are a bit more complicated when the confirmation layer is

1. decentralized, and
2. can be shared among multiple rollups

as is the case in Espresso. Blocks processed by HotShot may contain transactions meant for other rollups, which must be filtered out, and the process of checking that a block has been confirmed by HotShot is more complicated than simply reading from the rollup contract.

Handling Multi-Rollup Blocks

Let's tackle the first problem first. The rollup contract must be able to check that the lists of transactions executed by the proposer and challenger match a certain block produced by the sequencer and confirmed by HotShot. However, these transactions may not be the entirety of any block, since HotShot will produce blocks that combine transactions for multiple rollups. Thus, the contract needs a way to check that the proposed list of transactions contains all transactions from this rollup in a particular block ("completeness"), only those transactions ("inclusion"), and respects the order dictated by the sequencer. It must do this without necessarily having access to the whole HotShot block.

Instead of the whole block, the contract will work with short, binding commitments to blocks. The proposer and challenger can each provide a commitment *C* to the block they have executed along with a proof of inclusion and completness relative to *C* for the list of transactions they executed. The contract then simply verifies that proof and checks that the commitment *C* was actually confirmed at the required position in the chain (see the next section).

To permit an efficient proof of completeness, *C* is computed according to the Savoiardi VID scheme described in Appendix A of The Espresso Sequencing Network. The Espresso SDK, also under development, will include predefined functions for working with these commitments and doing proofs about them, so the complexity will be abstracted away from rollup integrations.

There is a subtle point regarding ordering. The inclusion and completeness proofs provided by the block commitment scheme enforce an ordering, which prevents an attack where a malicious proposer executes the right transactions in the wrong order, arriving at the wrong state root which was nonetheless correctly computed, and cannot be challenged. However, some rollups may *want* to allow the proposer to execute transactions in a different order than the sequencer dictated. Such rollups can have the contract check the ordering proof, and then apply a deterministic *reordering function* before proceeding to the execution phase of the challenge protocol, thus ensuring that the transactions are executed in an order which respects VM-specific constraints. It may be beneficial to do so, because HotShot is agnostic to the semantics of any particular rollup and does not enforce ordering constraints at the consensus level.

However, most rollups can probably consider this step optional. In practice, the sequencers elected through Espresso (and sequencers in general) will work with builders that are aware of rollup-specific semantics. These builders will seek to create valuable blocks, either by requiring users to pay fees or via arbitrage, and thus they will have an economic incentive to fill their blocks with valid transactions. In most cases, then, it will be sufficient for rollups to reject transactions that are "out of order" (by executing them as no-ops), which can be somewhat simpler than unconditionally sorting or reordering a whole block.

Checking That a Block Has Been Finalized

The last unsolved problem is how the rollup contract can check that a block commitment *C*, which has been proven to correspond to a certain list of rollup transactions, has actually been finalized at a given position in the chain.

With a centralized sequencer, this check is usually straightforward. The sequencer may have a known public key, for which it can produce a signature on any given block. Or, the sequencer may own the only Ethereum account authorized to call the function on the rollup contract which stores a new block on the layer 1. In most cases this smart contract is used as the single source of truth, even if multiple parties are allowed to sequence transactions via escape hatches.

With a decentralized system, things are a bit more complicated. Rollups that integrate with Espresso have their blocks processed by a decentralized **consensus protocol**, where thousands of nodes act as peers, and no one node has the privilege of unilaterally determining the status of sequenced rollup blocks. A block is considered *finalized* if this network of peers reaches consensus on the decision to include that rollup block at the next available position in the chain. Luckily, the process of reaching consensus produces artifacts which can be independently verified by non-participants of consensus, including smart contracts. These artifacts are called **quorum certificates**, or **QCs**.

A quorum certificate shows that a vote took place to include a certain block in a certain view, and that consensus nodes controlling a sufficient fraction of the total stake voted yes. The certificate contains an aggregated signature from those that voted. Due to the nature of distributed consensus, it actually requires three (in an upcoming version of HotShot this will be reduced to two) consecutive rounds of voting to finalize a block, so a chain of three consecutive valid QCs is definitive evidence that a block (and all previous blocks) has been finalized. Thus, any client, such as a rollup state transition verifier, wishing to verify that a rollup block has been sequenced must obtain and validate a chain of three consecutive QCs.

Luckily, the work required to verify finality for each block is the same across all rollups using Espresso, and can be shared. This is where the **sequencer contract** comes in. It is a single contract that receives commitments to sequenced blocks along with QCs proving the finality of those commitments, verifies the QCs, and stores a commitment to the finalized order of blocks. Anyone can append a commitment to a newly sequenced block to the contract, simply by providing a valid chain of QCs, which can be obtained from any honest consensus node.

Once the sequencer contract has done the hard work of checking QCs to verify that a block is finalized, anyone else can check finality simply by checking that the block is included in the sequence committed to by the contract. The contract uses a Merkle tree to commit to the sequence of finalized blocks, storing the root of this tree, so this check is usually done by Merkle proof.

In a typical optimistic rollup, there are several parties who need to verify that a certain block has been sequenced:

- The proposer and challenger must verify that a block has been sequenced before using it to compute a new state root. They can do this easily by waiting for the sequencer contract to verify the block and emit an event confirming that the block has been finalized. However, they may opt to confirm the block faster than the sequencer contract (thus providing *preconfirmations* to users) by downloading and verifying the QCs themselves. There are two options for verifying QCs:

    - Use the Espresso SDK to run the same QC verification algorithm that HotShot consensus uses.
    - Participate in consensus as a HotShot node. The HotShot node interface exposes a stream of verified blocks that the proposer and challenger can then consume.

- The rollup contract can read the sequence commitment directly from the sequencer contract. It can then require as part of the challenge protocol that the proposer and challenger provide Merkle proofs showing that the block commitment *C* from their inclusion/completeness proofs exists at a certain position in the Merkle tree.

Transaction Format

From HotShot's perspective, a transaction is just an array of bytes with an integer attached to identify the rollup that the transaction belongs to. Therefore, rollups using Espresso can keep their existing transaction format. The only change required is that, if the rollup provides a service like JSON-RPC that accepts transaction submissions, it must be modified to attach the rollup identifier when forwarding the transaction to Espresso.

The rollup identifier works much like an EVM chain ID. Each rollup is completely free to choose whatever identifier they want. However, it is strongly recommended to choose an identifier that no other rollup is using, because the rollup identifier determines which transactions are included in the completeness proof when filtering a multi-rollup block. Therefore, if you choose a rollup identifier which is already in use, your rollup will be forced to execute not only its own transactions but also all of those intended for the rollup with the same ID.

Downloading Data

Once a block has been finalized, various rollup participants will need to download it or a subset of it from the Tiramisu data availability layer. We consider three main use cases:

- A node wants to get notified when a new block is finalized
- An end user wants a proof that a particular transaction has been included in a block, but they don't want to download the entire block. This is a way of obtaining fast finality, because once a transaction is included in a finalized block, it is guaranteed that the rollup will eventually execute it. (This follows from completeness proofs.)
- A proposer or challenger wants to download just the subset of a block pertaining to the relevant rollup, with a proof that the server has provided the correct transactions in the correct order.

All of these use cases can make use of the **availability API**. Any HotShot node or client can provide this API by plugging in the modular HotShot query service.

New Block Notifications

The availability API provides several streaming endpoints, which a client can connect to using a WebSockets client. These endpoints allow clients to receive information when a new block is finalized or becomes available in the DA layer, without excessive polling. The streaming endpoints are:

- /availability/stream/leaves/:height

Stream blocks as soon as they are finalized, starting from :height (use 0 to start from genesis). The stream yields leaves, which contain metadata about finalized blocks such as the identity of the node that proposed them, the signature from nodes that voted for the block, and so on. This is the fastest way to learn of new blocks, but because Tiramisu disseminates data asynchronously, the actual *contents* of the block may or may not be included in this stream.

- /availability/stream/blocks/:height

This endpoint is similar to the leaves stream, but it waits until a block is fully available for download from Tiramisu before notifying the client. Each entry in the stream is a block with its full contents.

- /availability/stream/headers/:height

This endpoint is similar to the blocks stream, in that it will not notify the client of a new block until the full contents of that block are ready for download. However, it will not *send* the full contents of the block. It will only send the block header, which contains metadata like the block height and timestamp. This is a good way to wait until a block is finalized, at which time you can use some of the finer-grained endpoints discussed below to download only a subset of the block contents, saving bandwidth.

In the following sub-sections, it is assumed that clients of the availability API will use one of these streams to wait for more blocks to be sequenced before querying for the specific data they are interested in.

Single-Transaction Finality

The typical flow for this use case is

1. A user builds a transaction using rollup-specific client software.
2. The user saves the hash of their transaction and then submits it.
3. The user queries the availability API for proof that a transaction with the same hash has been included in a block.
4. The user checks that the resulting block has in fact been finalized.
5. The user verifies the proof, at which point it is guaranteed that the rollup will eventually execute the transaction.

The query for a proof uses the endpoint GET /availability/transaction/hash/:hash, replacing :hash with the tagged base 64 encoding of their transaction hash. If the requested transaction has in fact been sequenced, the response is a JSON object with a key proof, containing a proof of inclusion in a block, as well as metadata about the block, such as height and block_hash. It does not include the full block contents, so the bandwidth usage is minimal.

The user can check that block_hash has been sequenced as described above: either by checking for the corresponding event from the sequencer contract, or by downloading the relevant QCs and verifying them manually. The QCs can be obtained using the endpoint GET /availability/leaf/:height, and the Espresso SDK will include functionality for verifying them.

Once the user has confirmed that the block is finalized, the only thing left to do is to verify the proof that the transaction of interest was included in that block. This is a namespace KZG

inclusion proof just like the ones used in the state transition proofs, and the SDK will include functionality for verifying it.

Rollup's Subset of a Block

Rollup proposers and challengers must download transactions relevant to their rollup in order to compute new state roots, but it would be wasteful to download entire blocks, which may contain many transactions from other rollups. However, they do not generally want to trust the availability service to provide the correct subset of transactions. The desired flow is:

1. A proposer or challenger queries for the relevant subset of the next block.
2. The availability service responds with the desired transactions and a proof of completeness and inclusion.
3. The node verifies that the block has been finalized, verifies the completeness/inclusion proof, and then executes the transactions.

The node's query has the form GET /availability/block/:height/namespace/:rollup-id. On success, this returns a JSON object with two keys:

- block, a commitment to the desired block
- proof, which includes within it a list of transactions and proves their inclusion and completeness in block.

The node checks that block has been finalized at :height as described above: either by checking for the corresponding event from the sequencer contract, or by downloading the relevant QCs and verifying them manually. The QCs can be obtained using the endpoint GET /availability/leaf/:height, and the Espresso SDK will include functionality for verifying them.

proof is a KZG namespace proof just like the ones used in the state transition proofs, and the SDK will include functionality for verifying it. After verifying the proof, the node is assured that block :height includes all the returned transactions, in the correct order, and no other transactions with ID :rollup-id. It can then execute the transactions to compute the next rollup state.

Data Availability

Tiramisu is a scalable and secure data availability solution which ensures that all sequenced blocks will be available for rollup participants to download and execute. This ensures that any participant can reconstruct the state of the rollup.

While Tiramisu with ETH restaking can be just as secure as Ethereum DA, using it as the only source of data availability technically makes a rollup into a **validium**. Some rollups in the Ethereum ecosystem place a high value on persisting all of their data to Ethereum. Espresso supports both approaches. Any rollup may continue to use Ethereum for DA in addition to Tiramisu simply by having a rollup node send each block produced via Espresso to a layer 1 contract. If your existing rollup already uses Ethereum DA, this is actually one less change you have to make!

Using Espresso

This section describes the recommended architecture for an optimistic rollup integrated with Espresso.

Here we describe the architecture for an optimistic rollup using Espresso for confirmations, data availability, and sequencing. Rollups may choose to use a separate sequencer (centralized or decentralized) while still using Espresso for confirmations, with a similar system architecture.

Additions, deletions, and changes are highlighted in comparison with the typical optimistic rollup architecture.

Components

Generalized optimistic rollup architecture, modified to use Espresso

Espresso brings a few new components into the picture:

- **HotShot** finalizes blocks produced by the Espresso Network in a few seconds. This ensures that rollup transactions are confirmed quickly and in a decentralized and secure fashion, owing to the many nodes participating in HotShot.

- **Tiramisu** provides data availability for all rollup blocks. This availability is guaranteed by the same operator set that guarantees finality, so it is reliable, and rollups have the option of using Tiramisu as their standalone data availability layer if they wish. Tiramisu is designed to provide higher throughput and lower fees than using Ethereum for data availability.
- The Light client contract runs a HotShot client on the layer 1 blockchain, verifying the finality of each block and storing a binding commitment to the order of blocks decided by HotShot. As described in the following sections, rollup contracts on the same layer 1 can use this contract to relate their state transitions to the committed block order.

Transaction Flow

The transaction flow with Espresso is very similar to the transaction flow without. The main difference is in how confirmed blocks are disseminated. When not integrated with Espresso, rollup nodes are free to choose from where they fetch sequenced blocks: from the rollup contract on layer 1, or directly from the sequencer. Espresso, however, does not send sequenced blocks to each rollup contract: doing so would be expensive and would not meet the needs of all rollups. Instead, Espresso only sends sequenced blocks to one place—the sequencer contract—and it only sends a *commitment* to each block, at that: since Espresso provides its own data availability solution, the default is *not* to store entire blocks on the layer 1.

With this new flow, rollup nodes may choose how they get *notified* of new blocks: either by streaming them directly from Espresso or streaming events from the sequencer contract. But in either case, they must retrieve the block *contents* from the Espresso Network (see Espresso's data availability API). Furthermore, rollups that still wish to use the layer 1 for data availability are responsible for sending the necessary data to the layer 1 once it has been sequenced.

It is important to note that while the transaction flow changes in terms of how transactions get sequenced (if Espresso is used for sequencing) and how rollups consume that sequence, there is no change in what a rollup does with a block *after* it is sequenced. This means that the execution layer, which makes up the bulk of the complexity of many rollups, can be used completely unmodified with Espresso.

The following sections describe how the components of the rollup must be modified to support the updated transaction flow and retain security.

Fraud Proofs in a Shared-Sequencer World

The bulk of the changes from the basic optimistic rollup architecture relate to the fraud proofs, requiring additions to the proposer, the challenger, and the rollup contract's challenge mechanism. While the details of challenge mechanisms vary widely between optimistic rollup instantiations, the general concept is universal: a proposer has sent a state root to the contract, and a challenger has claimed that the correct state root is something different. The challenger must then provide evidence that the proposed state root is incorrect, and the proposer may be given a chance to produce evidence defending their proposal.

There are two ways a state root can be incorrect:

1. The proposer executed the wrong transactions.
2. The proposer executed the right transactions incorrectly.

Therefore, any challenge mechanism must first check that the proposer and challenger both agree on the correct transactions to execute, according to the ordering determined by the sequencer. If they do not, then whichever one *has* executed the right transactions trivially wins the challenge. Once the proposer and challenger have agreed on the sequence of transactions being executed, the challenger must prove that the proposed state root is not the correct result of applying the state transition function to those transactions.

The first part, agreeing on the sequence of transactions, is generally a fairly straightforward check against the committed sequence. The second part, checking the proposer's computation of the state root, is where most of the complexity lies, since it involves proofs or interactive challenge games that encompass the low-level semantics of a complex VM. Luckily, this part of the challenge remains completely unchanged when integrating with Espresso, and only the mechanism for agreeing on the sequence of transactions needs to change.

With a single, centralized sequencer, reaching this agreement may only consist of a simple lookup, since the sequencer is sending blocks (or block commitments) directly to the rollup contract, and those blocks contain only transactions that are relevant to this particular rollup. Therefore the contract simply reads the last sequenced block directly from its own storage and compares it with the blocks claimed by the proposer and the challenger (or compares commitments, in some rollups). If the proposer's claimed block does not match the expected block, the contract can immediately consider the challenge successful. If the challenger's claimed block does not match, the contract can immediately reject the challenge.

Things are a bit more complicated when the confirmation layer is

1. decentralized, and
2. can be shared among multiple rollups

as is the case in Espresso. Blocks processed by HotShot may contain transactions meant for other rollups, which must be filtered out, and the process of checking that a block has been confirmed by HotShot is more complicated than simply reading from the rollup contract.

Handling Multi-Rollup Blocks

Let's tackle the first problem first. The rollup contract must be able to check that the lists of transactions executed by the proposer and challenger match a certain block produced by the sequencer and confirmed by HotShot. However, these transactions may not be the entirety of any block, since HotShot will produce blocks that combine transactions for multiple rollups. Thus, the contract needs a way to check that the proposed list of transactions contains all transactions from this rollup in a particular block ("completeness"), only those transactions ("inclusion"), and respects the order dictated by the sequencer. It must do this without necessarily having access to the whole HotShot block.

Instead of the whole block, the contract will work with short, binding commitments to blocks. The proposer and challenger can each provide a commitment *C* to the block they have executed along with a proof of inclusion and completness relative to *C* for the list of transactions they executed. The contract then simply verifies that proof and checks that the commitment *C* was actually confirmed at the required position in the chain (see the next section).

To permit an efficient proof of completeness, *C* is computed according to the Savoiardi VID scheme described in Appendix A of The Espresso Sequencing Network. The Espresso SDK, also

under development, will include predefined functions for working with these commitments and doing proofs about them, so the complexity will be abstracted away from rollup integrations.

There is a subtle point regarding ordering. The inclusion and completeness proofs provided by the block commitment scheme enforce an ordering, which prevents an attack where a malicious proposer executes the right transactions in the wrong order, arriving at the wrong state root which was nonetheless correctly computed, and cannot be challenged. However, some rollups may *want* to allow the proposer to execute transactions in a different order than the sequencer dictated. Such rollups can have the contract check the ordering proof, and then apply a deterministic *reordering function* before proceeding to the execution phase of the challenge protocol, thus ensuring that the transactions are executed in an order which respects VM-specific constraints. It may be beneficial to do so, because HotShot is agnostic to the semantics of any particular rollup and does not enforce ordering constraints at the consensus level.

However, most rollups can probably consider this step optional. In practice, the sequencers elected through Espresso (and sequencers in general) will work with builders that are aware of rollup-specific semantics. These builders will seek to create valuable blocks, either by requiring users to pay fees or via arbitrage, and thus they will have an economic incentive to fill their blocks with valid transactions. In most cases, then, it will be sufficient for rollups to reject transactions that are "out of order" (by executing them as no-ops), which can be somewhat simpler than unconditionally sorting or reordering a whole block.

Checking That a Block Has Been Finalized

The last unsolved problem is how the rollup contract can check that a block commitment *C*, which has been proven to correspond to a certain list of rollup transactions, has actually been finalized at a given position in the chain.

With a centralized sequencer, this check is usually straightforward. The sequencer may have a known public key, for which it can produce a signature on any given block. Or, the sequencer may own the only Ethereum account authorized to call the function on the rollup contract which stores a new block on the layer 1. In most cases this smart contract is used as the single source of truth, even if multiple parties are allowed to sequence transactions via escape hatches.

With a decentralized system, things are a bit more complicated. Rollups that integrate with Espresso have their blocks processed by a decentralized **consensus protocol**, where thousands of nodes act as peers, and no one node has the privilege of unilaterally determining the status of sequenced rollup blocks. A block is considered *finalized* if this network of peers reaches consensus on the decision to include that rollup block at the next available position in the chain. Luckily, the process of reaching consensus produces artifacts which can be independently verified by non-participants of consensus, including smart contracts. These artifacts are called **quorum certificates**, or **QCs**.

A quorum certificate shows that a vote took place to include a certain block in a certain view, and that consensus nodes controlling a sufficient fraction of the total stake voted yes. The certificate contains an aggregated signature from those that voted. Due to the nature of distributed consensus, it actually requires three (in an upcoming version of HotShot this will be reduced to two) consecutive rounds of voting to finalize a block, so a chain of three consecutive valid QCs is definitive evidence that a block (and all previous blocks) has been finalized. Thus,

any client, such as a rollup state transition verifier, wishing to verify that a rollup block has been sequenced must obtain and validate a chain of three consecutive QCs.

Luckily, the work required to verify finality for each block is the same across all rollups using Espresso, and can be shared. This is where the **sequencer contract** comes in. It is a single contract that receives commitments to sequenced blocks along with QCs proving the finality of those commitments, verifies the QCs, and stores a commitment to the finalized order of blocks. Anyone can append a commitment to a newly sequenced block to the contract, simply by providing a valid chain of QCs, which can be obtained from any honest consensus node.

Once the sequencer contract has done the hard work of checking QCs to verify that a block is finalized, anyone else can check finality simply by checking that the block is included in the sequence committed to by the contract. The contract uses a Merkle tree to commit to the sequence of finalized blocks, storing the root of this tree, so this check is usually done by Merkle proof.

In a typical optimistic rollup, there are several parties who need to verify that a certain block has been sequenced:

- The proposer and challenger must verify that a block has been sequenced before using it to compute a new state root. They can do this easily by waiting for the sequencer contract to verify the block and emit an event confirming that the block has been finalized. However, they may opt to confirm the block faster than the sequencer contract (thus providing *preconfirmations* to users) by downloading and verifying the QCs themselves. There are two options for verifying QCs:
    - Use the Espresso SDK to run the same QC verification algorithm that HotShot consensus uses.
    - Participate in consensus as a HotShot node. The HotShot node interface exposes a stream of verified blocks that the proposer and challenger can then consume.
- The rollup contract can read the sequence commitment directly from the sequencer contract. It can then require as part of the challenge protocol that the proposer and challenger provide Merkle proofs showing that the block commitment *C* from their inclusion/completeness proofs exists at a certain position in the Merkle tree.

Transaction Format

From HotShot's perspective, a transaction is just an array of bytes with an integer attached to identify the rollup that the transaction belongs to. Therefore, rollups using Espresso can keep their existing transaction format. The only change required is that, if the rollup provides a service like JSON-RPC that accepts transaction submissions, it must be modified to attach the rollup identifier when forwarding the transaction to Espresso.

The rollup identifier works much like an EVM chain ID. Each rollup is completely free to choose whatever identifier they want. However, it is strongly recommended to choose an identifier that no other rollup is using, because the rollup identifier determines which transactions are included in the completeness proof when filtering a multi-rollup block. Therefore, if you choose a rollup identifier which is already in use, your rollup will be forced to execute not only its own transactions but also all of those intended for the rollup with the same ID.

Downloading Data

Once a block has been finalized, various rollup participants will need to download it or a subset of it from the Tiramisu data availability layer. We consider three main use cases:

- A node wants to get notified when a new block is finalized
- An end user wants a proof that a particular transaction has been included in a block, but they don't want to download the entire block. This is a way of obtaining fast finality, because once a transaction is included in a finalized block, it is guaranteed that the rollup will eventually execute it. (This follows from completeness proofs.)
- A proposer or challenger wants to download just the subset of a block pertaining to the relevant rollup, with a proof that the server has provided the correct transactions in the correct order.

All of these use cases can make use of the **availability API**. Any HotShot node or client can provide this API by plugging in the modular HotShot query service.

New Block Notifications

The availability API provides several streaming endpoints, which a client can connect to using a WebSockets client. These endpoints allow clients to receive information when a new block is finalized or becomes available in the DA layer, without excessive polling. The streaming endpoints are:

- /availability/stream/leaves/:height

Stream blocks as soon as they are finalized, starting from :height (use 0 to start from genesis). The stream yields leaves, which contain metadata about finalized blocks such as the identity of the node that proposed them, the signature from nodes that voted for the block, and so on. This is the fastest way to learn of new blocks, but because Tiramisu disseminates data asynchronously, the actual *contents* of the block may or may not be included in this stream.

- /availability/stream/blocks/:height

This endpoint is similar to the leaves stream, but it waits until a block is fully available for download from Tiramisu before notifying the client. Each entry in the stream is a block with its full contents.

- /availability/stream/headers/:height

This endpoint is similar to the blocks stream, in that it will not notify the client of a new block until the full contents of that block are ready for download. However, it will not *send* the full contents of the block. It will only send the block header, which contains metadata like the block height and timestamp. This is a good way to wait until a block is finalized, at which time you can use some of the finer-grained endpoints discussed below to download only a subset of the block contents, saving bandwidth.

In the following sub-sections, it is assumed that clients of the availability API will use one of these streams to wait for more blocks to be sequenced before querying for the specific data they are interested in.

Single-Transaction Finality

The typical flow for this use case is

1. A user builds a transaction using rollup-specific client software.
2. The user saves the hash of their transaction and then submits it.
3. The user queries the availability API for proof that a transaction with the same hash has been included in a block.
4. The user checks that the resulting block has in fact been finalized.
5. The user verifies the proof, at which point it is guaranteed that the rollup will eventually execute the transaction.

The query for a proof uses the endpoint GET /availability/transaction/hash/:hash, replacing :hash with the tagged base 64 encoding of their transaction hash. If the requested transaction has in fact been sequenced, the response is a JSON object with a key proof, containing a proof of inclusion in a block, as well as metadata about the block, such as height and block_hash. It does not include the full block contents, so the bandwidth usage is minimal.

The user can check that block_hash has been sequenced as described above: either by checking for the corresponding event from the sequencer contract, or by downloading the relevant QCs and verifying them manually. The QCs can be obtained using the endpoint GET /availability/leaf/:height, and the Espresso SDK will include functionality for verifying them.

Once the user has confirmed that the block is finalized, the only thing left to do is to verify the proof that the transaction of interest was included in that block. This is a namespace KZG inclusion proof just like the ones used in the state transition proofs, and the SDK will include functionality for verifying it.

Rollup's Subset of a Block

Rollup proposers and challengers must download transactions relevant to their rollup in order to compute new state roots, but it would be wasteful to download entire blocks, which may contain many transactions from other rollups. However, they do not generally want to trust the availability service to provide the correct subset of transactions. The desired flow is:

1. A proposer or challenger queries for the relevant subset of the next block.
2. The availability service responds with the desired transactions and a proof of completeness and inclusion.
3. The node verifies that the block has been finalized, verifies the completeness/inclusion proof, and then executes the transactions.

The node's query has the form GET /availability/block/:height/namespace/:rollup-id. On success, this returns a JSON object with two keys:

- block, a commitment to the desired block
- proof, which includes within it a list of transactions and proves their inclusion and completeness in block.

The node checks that block has been finalized at :height as described above: either by checking for the corresponding event from the sequencer contract, or by downloading the relevant QCs

and verifying them manually. The QCs can be obtained using the endpoint GET /availability/leaf/:height, and the Espresso SDK will include functionality for verifying them.

proof is a KZG namespace proof just like the ones used in the state transition proofs, and the SDK will include functionality for verifying it. After verifying the proof, the node is assured that block :height includes all the returned transactions, in the correct order, and no other transactions with ID :rollup-id. It can then execute the transactions to compute the next rollup state.

Data Availability

Tiramisu is a scalable and secure data availability solution which ensures that all sequenced blocks will be available for rollup participants to download and execute. This ensures that any participant can reconstruct the state of the rollup.

While Tiramisu with ETH restaking can be just as secure as Ethereum DA, using it as the only source of data availability technically makes a rollup into a **validium**. Some rollups in the Ethereum ecosystem place a high value on persisting all of their data to Ethereum. Espresso supports both approaches. Any rollup may continue to use Ethereum for DA in addition to Tiramisu simply by having a rollup node send each block produced via Espresso to a layer 1 contract. If your existing rollup already uses Ethereum DA, this is actually one less change you have to make!