Interfaces

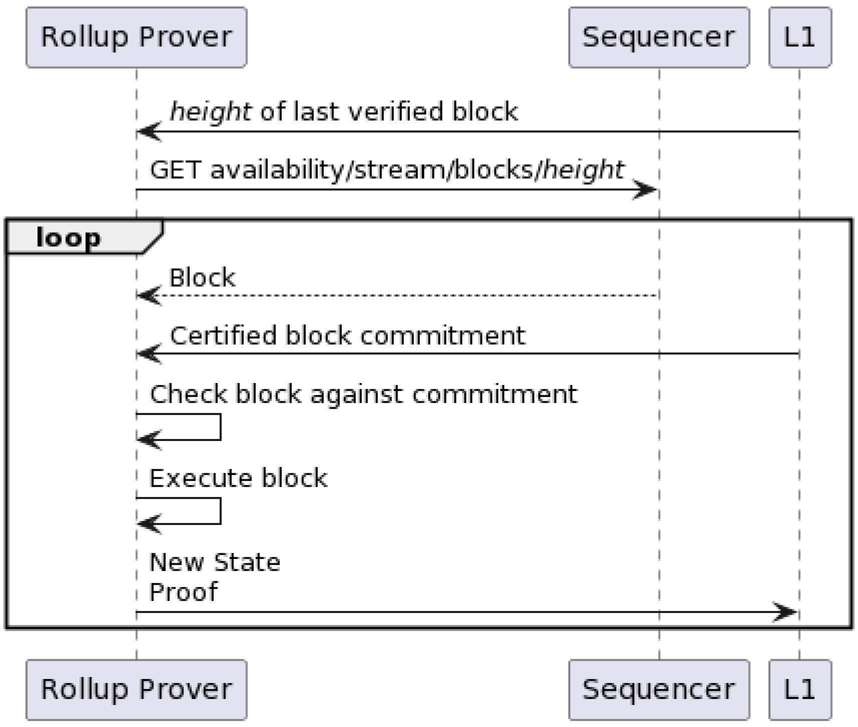
This section defines in detail the interfaces between each major component in the system architecture.

Espresso ↔ Rollup

In order to keep HotShot nodes themselves as generic and simple as possible, there is no rollup-specific logic in Espresso itself, and thus Espresso never actively communicates with any rollup. Instead, HotShot *query service* nodes present a public interface which rollups are expected to query in order to integrate with Espresso. This interface takes the form of a REST API. See the API reference for details.

Usage

Rollup nodes may use these APIs differently depending on the role they are playing in the system (e.g., prover, full node, etc.). A prover can use the API to stream block data from a node, so that it can execute blocks as they are finalized and generate proofs. The prover also interacts with the L1, since it can only verify a rollup proof on the L1 if the L1 has already verified the sequencing of the corresponding block.


Rollup Prover
Sequencer
L1
height of last verified block
GET availability/stream/blocks/*height*
loop
Block
Certified block commitment
Check block against commitment
Execute block
New State Proof
Rollup Prover
Sequencer
L1

Prover / Sequencer Integration

A rollup may also include full nodes which store and provide access to rollup-related state, but do not run a prover. Such a full node can stream blocks and verify consensus proofs (QCs) directly from the HotShot APIs, without interacting with the L1. Avoiding interaction with the L1 allows state updates to be computed faster.

Full Node / Sequencer Integration

The rollup also interacts with HotShot via the submit API. This interaction is completely independent of the streaming interaction illustrated above. It is simply used to add transactions to HotShot's mempool so that they may eventually be included in the sequence. Any rollup node which serves a rollup API (e.g. JSON-RPC) should be able to handle transaction submissions through HotShot's submit API.

The body of submit requests includes the transaction to submit as well as a rollup-specific numeric identifier. This identifier is associated with the transaction in the final sequence, so rollup proofs can use the ID to easily exclude transactions intended for other rollups. Each rollup should have its own protections against cross-rollup replay attacks, such as an EVM chain ID, in addition to this rollup ID.

Espresso ↔ L1

Espresso interacts with the L1 via the sequencer contract, which validates HotShot consensus and provides a certified, trustless interface for other participants to check the sequence of blocks. Note that the contract only deals with short block commitments, not full blocks, in order to minimize the cost of sending data to the L1. Anyone who has verified a commitment against the contract can get the corresponding block—and authenticate it against the commitment—from the HotShot availability API.

Espresso interacts with the sequencer contract via an interface like:

// HotShot.sol

Copy

```
struct QC { /* Fields omitted */ }


// Root of a Merkle tree accumulating the verified sequence of block commitments.

bytes32 public commitmentsRoot;


// Event emitted when new blocks are sequenced.

event NewBlocks(uint firstBlockNumber, uint256[] commitments, bytes32[] frontier);


// Called to append a chain of new blocks, given proof that consensus has finalized them.

function newBlocks(QC[] calldata qcs, bytes calldata proof, bytes32[] calldata frontier) external;
```

The newBlocks method allows a sequencer node to append a list of newly sequenced block commitments to the log stored in the contract. It takes a list of quorum certificates, a validity proof, and a Merkle frontier corresponding to commitmentsRoot, and it validates that

- Each QC extends from the previous QC in the chain (starting with the previously sequenced QC)
- Each QC is properly signed (the contract will need to store and keep up-to-date the stake table)

- There are enough QCs to prove finality for one or more block commitments. HotShot consensus currently requires a chain of at least 3 QCs before the first QC in the chain is considered finalized (an upcoming version of HotShot will only require a 2-chain of QCs)

If validation succeeds, it updates commitmentsRoot, which can then be used by other contracts to validate proofs of inclusion of block commitments in the sequence. On success, newBlocks emits a NewBlocks event informing clients (e.g. rollup provers) that new blocks have been appended. Those clients can read the new block commitments from the event logs using an Ethereum client. The event logs also include a snapshot of the Merkle frontier just before the new blocks were appended. A client can construct a Merkle path for any given commitment by appending commitments to the snapshotted frontier.

newBlocks will fail if the given batch has already been sequenced, since qcs will fail the check that it must extend from the last sequenced QC. This ensures that each batch of blocks will only be sequenced once — whoever calls this method first will be the one to sequence it. It is an open question whether the contract will explicitly incentivize sequencer nodes to call newBlocks.

To learn more about the sequencer contract, how it stores data, and how it validates QCs, read the section on its internal functionality.

Rollup ↔ L1

Each rollup communicates with the L1 via its own rollup contract, which can have a unique public interface. In order to verify state updates sent from the rollup (either proactively with validity proofs in the case of ZK-rollups, or when presented with a fraud proof in the case of optimistic rollups), each rollup contract must have access to the certified sequence of blocks which led to the claimed state update.

When using Espresso, the authoritative sequence of blocks is the output of HotShot, which is replicated to the L1 and certified by the HotShot contract. Therefore, rollup contracts will interface with the HotShot contract. The interface which allows rollup contracts to query the certified sequence of block commitments is very simple: the sequencer contract provides a public sequencedCommitments field, which is an array of block commitments which have been verified by the contract.