The Espresso Network

The global confirmation layer

Espresso's Global Confirmation Network is a shared source of truth that provides secure confirmations for transaction ordering and data across chains. It leverages a BFT consensus protocol called HotShot. Rollup transactions are processed by HotShot consensus and confirmed within a few seconds. By sharing the same Global Confirmation Network, chains have certainty within seconds about what state will eventually be finalized on the L1. This is achieved by preventing sequencers from changing a transaction sequence after it has been confirmed by HotShot, which can happen within only a few seconds (long before settlement on Ethereum).

Compared to relying on a disparate set of centralized sequencers for pre-confirmations, sharing a decentralized consensus protocol as a Global Confirmation Network bolsters the L2 ecosystem as a whole against equivocation of pre-confirmations that may occur as a result of hacks or bad actors. Applications that interact between multiple chains, such as bridges, are especially sensitive to the quality of confirmations of individual chains they are involved with.

This is reflected by the fact that for many of the largest bridge providers today, the waiting time for moving funds away from a rollup is around 15 minutes; the time it takes for the bridging transaction to be finalized inside a blob or as calldata on Ethereum. With HotShot, bridging between rollups can be done in a manner of seconds, as opposed to waiting 15 minutes for transactions to finalize on the Ethereum L1, with similar security guarantees.

The confirmation guarantees of HotShot mirror that of Ethereum L1. Once HotShot confirms a transaction reached, an adversary would need to control at least 1/3rd of the overall stake to revert said transaction. Combined with restaking, this means that HotShot's safety guarantee can approach that of Ethereum L1 over time. In the event that safety is broken, the adversary's stake will be slashed, making it very expensive to undo confirmations in HotShot.

System Overview

Chains (e.g. L2 rollups) interact with Espresso and with the layer 1 blockchain to facilitate trustless state checkpoints. Specifically, when we refer to L1 and L2s we mean the following:

- **The L1 (layer 1) blockchain:** This is the blockchain that chains using Espresso post state checkpoints to. HotShot must also checkpoint its state and history on the layer-1, which will serve as an interface to the rollups. Initially Espresso will run a smart contract on Ethereum that tracks HotShot's state commitments. Other projects can recycle these state commitments to reuse on other chains if they wish.
- **The L2 (layer 2) chains:** These are the chains that use Espresso for selling sequencing rights and confirmations. They could be anything from app-specific chains to fully-featured smart contract systems in their own right (like EVM rollups). Each chain is assumed here to be structured as a rollup: after receiving an ordered sequence of transactions, the transactions are executed off-chain in some deterministic VM, and periodically state updates are posted to the layer-1, along with a proof of validity (for ZK-rollups) or a potential fraud proof (for optimistic rollups).

This diagram shows the flow of information through the entire system, starting with transaction submission through clients to various integrated rollups and finally to being certified and checkpointed on the layer-1.

The next sections dive into these components and their interactions in more detail.

Rollup Components

The internal architecture of each rollup will vary greatly depending on the type of rollup (e.g., ZK/optimistic; EVM/app-specific) but all rollups will have a few similar basic components. The internal structure of Rollup 1 is shown in detail in the diagram (for simplicity, the other rollups are abstracted).

A rollup must provide an interface for clients to interact with it. This can be any kind of **API**, although Ethereum-compatible **JSON-RPC** will be common. The API responds to queries about the rollup state by reading from a **state database**, which is populated by the **executor**, a component which executes every block provided by the sequencer. Finally, a **prover** (which may be part of a decentralized prover network) is triggered by updates to the state and is responsible for justifying those state updates. For ZK-rollups, the prover will be triggered by every block and produce a validity proof for the state update. For optimistic rollups, the prover will only be triggered if another node publishes an invalid state update, in which case the prover will generate a fraud proof.

In addition to answering state queries, the rollup API may also serve as an endpoint for clients to submit transactions. While clients can submit transactions directly to the Espresso Network's mempool, doing so may be inconvenient for a few reasons:

- The Espresso Network's transaction submission interface is not specific to any one rollup. Clients will have to wrap their rollup-specific transactions into a more generic kind of transaction before submitting.
- It requires clients to interact with two different services: the rollup API for state queries and the Espresso Network (HotShot) for transaction submissions. Depending on the client software, this may not even be possible. MetaMask, for example, requires a single URL for each chain that it can use for queries and transaction submission.

It is therefore recommended that rollup servers provide a transaction submission interface as part of their API, for those clients who are already using the rest of the API. Such an interface is actually required for conforming JSON-RPC implementations, since the eth_sendRawTransaction RPC method allows clients of the RPC to submit a transaction. Whatever its interface, the implementation of the rollup's submission API can be as simple as wrapping rollup transactions into generic transactions and forwarding them to the HotShot mempool.

Transaction Flow

Once a transaction is forwarded or submitted, it will be included in a block, and confirmed and made available by HotShot, after which blocks propagate back through the rollups' executors and provers, which in turn forward their blocks to the L1. Espresso also sends a block commitment to the layer 1 sequencer contract, along with a quorum certificate that the contract uses to authenticate the block. This allows layer 1 rollup contracts to compare the rollup state update proof against a block commitment which is certified as having been finalized by HotShot consensus.

The diagram below shows in more detail the flow of a single transaction from a client through the system.

Transaction Lifecycle

1. User sends a transaction to a chain's server (e.g. an RPC service).
2. The chain forwards the transaction, along with an identifier for that rollup, to the chain's sequencer.
3. The chain's sequencer includes the transaction in a block, which is broadcast to subscribers. One of these subscribers, the rollup node, executes the transaction (along with any other transactions in the block belonging to that rollup). In the case of a ZK rollup, the node may produce a proof of correct execution, which can be broadcast to clients to quickly convince them of the new state.
4. A commitment to the block containing the transaction is persisted in the L1 sequencer contract (along with a proof that the block has been finalized by consensus).
5. A rollup node which has executed the block sends the new rollup state to the L1. It may include a validity proof (for ZK rollups) or open a window for fraud proofs (optimistic rollups).
6. The L1 rollup contract verifies any proofs related to the state update, using the certified block commitment from the sequencer contract to check that the state update corresponds to the correct block.

Confirmations

Centralized sequencers may give rollup users a pre-confirmation that their transaction will eventually be included in the finalized rollup state. These pre-confirmations can be trusted via a combination of reputation, security bonds, and/or fraud proofs.

Clients can also opt to wait for a stronger confirmation provided by HotShot (step 3). As long as no adversary controls more than one third of the HotShot stake, the client's transaction can never be rolled back. This is especially usefull for bridging, as they are sensitive to reorgs in the source chain of a bridge transaction.

In the case that a rollup posts their tx data to the L1, clients can also wait for their tx to be finalized on the L1, though in the case of Ethereum, this guarantee can take 15 minutes to attain, as opposed to a few seconds in the case of HotShot.

Once a transaction is finalized, clients may want to read the updated rollup state, either to check the results of their transaction's execution or to prepare another transaction. They have several options (listed below) for doing this, depending on who they trust and how much work they are willing to do themselves. Depending on which option they choose, each client can get their preferred tradeoff between latency, trust, and the amount of work they are required to perform.

- They can rely on the pre-confirmation provided by the sequencer to compute the next rollup state.
- They can leverage Espresso's Global Confirmation Layer and immediately execute it themselves to compute the new state.

- They can get a state update, at the cost of additional trust assumptions, by trusting a rollup server who has executed the transaction to give them the new state, even before a state update proof is generated.
- In the case of a ZK rollup, they can wait for a state update proof to be generated (step 3) and check that proof. This requires less computation than executing the block, and it is still trustless.
- Finally, if a client does not want to do any computation on their own (or in the case of an optimistic rollup, where there is no validity proof for the client to check) and does not want to trust a rollup server, they can wait until a state update is certified by the L1 (step 6) to fetch the updated state with no trust or computation.

Multi-Chain Blocks

In the case that a sequencer has won the right to sequence for multiple chains simultaneously, blocks produced by that sequencer (step 3) may include transactions from many different rollups. Chains need some way to prove they have executed all of the transactions belonging to them, and only those transactions, preferably without searching exhaustively through the entire block. Without this, chain nodes could censor users by simply ignoring some sequenced transactions, or they could create confusing state updates by executing transactions that were intended for another chain.

To support this efficiently, each transaction submitted to the sequencer must be associated with a chain-specific identifier (step 2). Each block produced by the sequencer contains transactions organized by namespace. The block also has a short, unique *commitment*, which depends on the transactions in the block and other metadata, like its height and timestamp. It is possible to create proofs relative to this commitment which convince a verifier that a given namespace in the block contains a certain list of transactions. This allows someone who only cares about a particular namespace, like a rollup node, to download just the data they care about (plus a short proof) from an untrusted server, and verify it.

Namespace proofs are small and computationally cheap to check, which makes them suitable for verification by an L1 smart contract or in a ZK circuit. This functionality is essential for ZK rollups that need to prove they have executed all the relevant transactions in a certain block in order to compute a new state, and for optimistic rollups that need to arbitrate a fraud challenge about the execution of relevant transactions in a block.

This namespaced block commitment is the commitment that is certified and stored in the L1 sequencer contract (step 4). When rollup contracts need to verify an inclusion proof for a sequence of rollup-specific transactions in a particular block, they can read the corresponding block commitment directly from the sequencer contract and use it to verify the proof.

Note that the sequencer does not authenticate the mapping from chain IDs to transaction sequences. Nothing stops a bad actor from submitting a transaction for one chain with the ID of another, or from submitting a transaction which is invalid in a particular chain with the ID of that chain. As such, chains will still need the ability to detect and exclude invalid transactions in their execution layers. This is fundamental to chains with permissionless sequencing.

A well-behaved chain should choose an ID for itself which is distinct from any IDs which are in use by other well-behaved chains. This is similar to how, in the Ethereum ecosystem, different EVM blockchains choose different chain IDs, and the responsibility ultimately falls on users to

avoid using a malicious chain which copies the chain ID of some other system. In fact, EVM rollups could in theory use their EVM chain IDs as their Espresso chain identifiers.