Appendix

Interacting with L1

Espresso is intended primarily for use with rollups, and an essential feature of rollups is that they periodically send verified checkpoints of their state to some other blockchain, the layer 1. In the case of Espresso we have two types of checkpoints:

- **HotShot checkpoints** record the history and state of HotShot. This includes commitments to all blocks that have been sequenced and the state that is needed to verify new blocks (such as the stake table) starting from the latest checkpoint. These checkpoints are recorded in the sequencer contract and shared by all applications using Espresso.
- **Rollup checkpoints** record the state of individual rollups, such as the state of an EVM instance. Each rollup records its own checkpoints in its rollup contract. When recording a new checkpoint, the rollup contract will validate the checkpoint against the committed sequence of blocks by reading the historical sequence of block commitments from a corresponding HotShot checkpoint.

Many rollups will not only write their state to the L1, but will also allow information from the L1 to flow back into the rollup. In this case, the interaction with the rollup contract becomes a bridge between the L1 and the rollup, allowing tokens and other information to be exchanged bidirectionally.

Espresso is interesting in that HotShot itself *also* allows this bidirectional flow of information to and from the L1, specifically regarding the *stake table*. This allows operations on the L1 to interact with the Espresso staking token, a crucial requirement for implementing restaking, which allows Espresso to share the security budget of the L1 and pass value generated by the sequencer back to L1 validators.

Since state updates are not only written to the L1 but also read back into the HotShot and rollup states, we will hereafter eschew the "checkpoint" terminology and instead refer to **HotShot state updates** and **rollup state updates**.

The following sections dive into the use cases for these state updates and explore some of the utility and security properties that L2s are able to derive from the L1 that they use for state updates.

Trustless Sync

One of the main reasons to use blockchains is to decentralize trust. However, the current Ethereum ecosystem often compromises on this principle by using trusted query services like Infura for clients to access the blockchain. This trades off trustlessness for convenience and scalability, since a client that trusts a query service does not need to verify any Ethereum block data.

However, Ethereum's switch to proof-of-stake and its rollup-centric roadmap offer the potential to break out of this tradeoff, enabling efficient and user-friendly clients that retain decentralized trust. When rollups post their state updates to Ethereum or a similar L1, each L1 validator independently validates the state transition for the rollup by executing the rollup's smart

contract. This means that any user who trusts the collective L1 validator set can quickly sync with the latest state of the rollup simply by reading a recent, verified state update from the L1 state.

Of course, this only pushes the problem of fast, trustless sync to the L1. This is nonetheless a substantial improvement. In the case of Ethereum, the proof-of-stake consensus protocol and the fixed block time enable the creation of L1 light clients which sync far faster than real time. For instance, Helios is an Ethereum light client which manages trustless sync in only 2 seconds. By verifying HotShot and rollup states on Ethereum, any such Ethereum client can be turned into a client for any rollup merely by syncing with Ethereum and then reading rollup state from the appropriate smart contract.

Fork Recovery

When we think of rollups checkpointing to an L1, one of the first things that comes to mind is allowing the rollups to inherit the security of the L1. This subject is subtle, and discussions of it often lack a clear accounting of the security properties we hope for the rollup to obtain and how exactly they are inherited. This section will detail how rollups that checkpoint to an L1 inherit an extra level of *finality* from the L1.

First, it is important to note that Espresso, on its own, even without any form of checkpointing, already offers extremely strong finality guarantees. In order to break finality, and adversary must cause a safety violation in the HotShot finality gadget, but HotShot is a secure BFT protocol—violating its security requires an adversary to control over 1/3 of the total stake, a massive investment once HotShot reaches a critical mass of adoption.

Nevertheless, we can consider what happens in the extremely unlikely event that HotShot does suffer a safety violation, and some block which was considered final according to the HotShot protocol is removed from the history of the ledger. In this case, we can maintain finality by using the HotShot checkpoints on L1 to decide on an immutable, canonical chain.

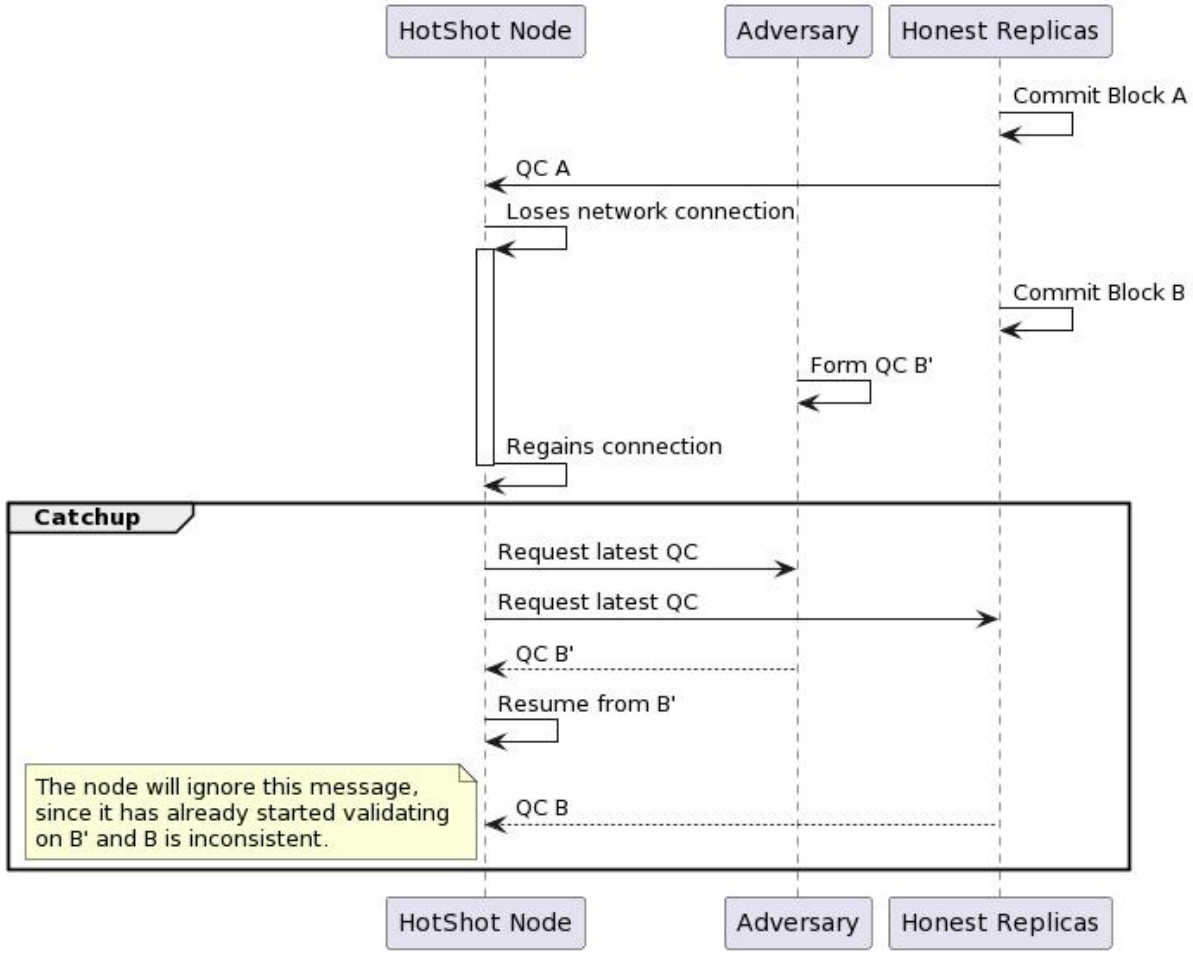
Let's look at how such a scenario could happen. There are two ways an adversary could compromise HotShot's finality.

First, if they control more than 1/3 of the total stake at any given time, they can cause or exploit a failure in the network connecting honest nodes, partitioning honest nodes controlling 1/3 of the stake each into two disjoint sub-networks. The adversary, being Byzantine, can then use their 1/3 of the stake to vote for conflicting blocks in each partition, committing both conflicting blocks with a 2/3 quorum each, and thus creating a fork.

Second, an adversary can create a fork without even controlling 1/3 of the current stake via a *long range attack*. In a long range attack, an adversary compromises some old private keys which were used to sign a quorum certificate many blocks ago. This may be substantially cheaper than compromising current private keys, as the owners of the old private keys may already have withdrawn their stake and thus may not be very invested in maintaining the security of their keys.

While compromising old keys does not directly enable the adversary to append new invalid blocks, they can build a new fork including signed QCs starting from the block at which they compromised the keys, and unilaterally grow this fork until it is similar in length to the canonical chain.

Once an adversary has created a fork, they can compromise finality by convincing honest nodes to switch from the canonical branch of the fork to the malicious branch. Blocks which were considered finalized on the canonical branch may not have been committed at all on the malicious branch.

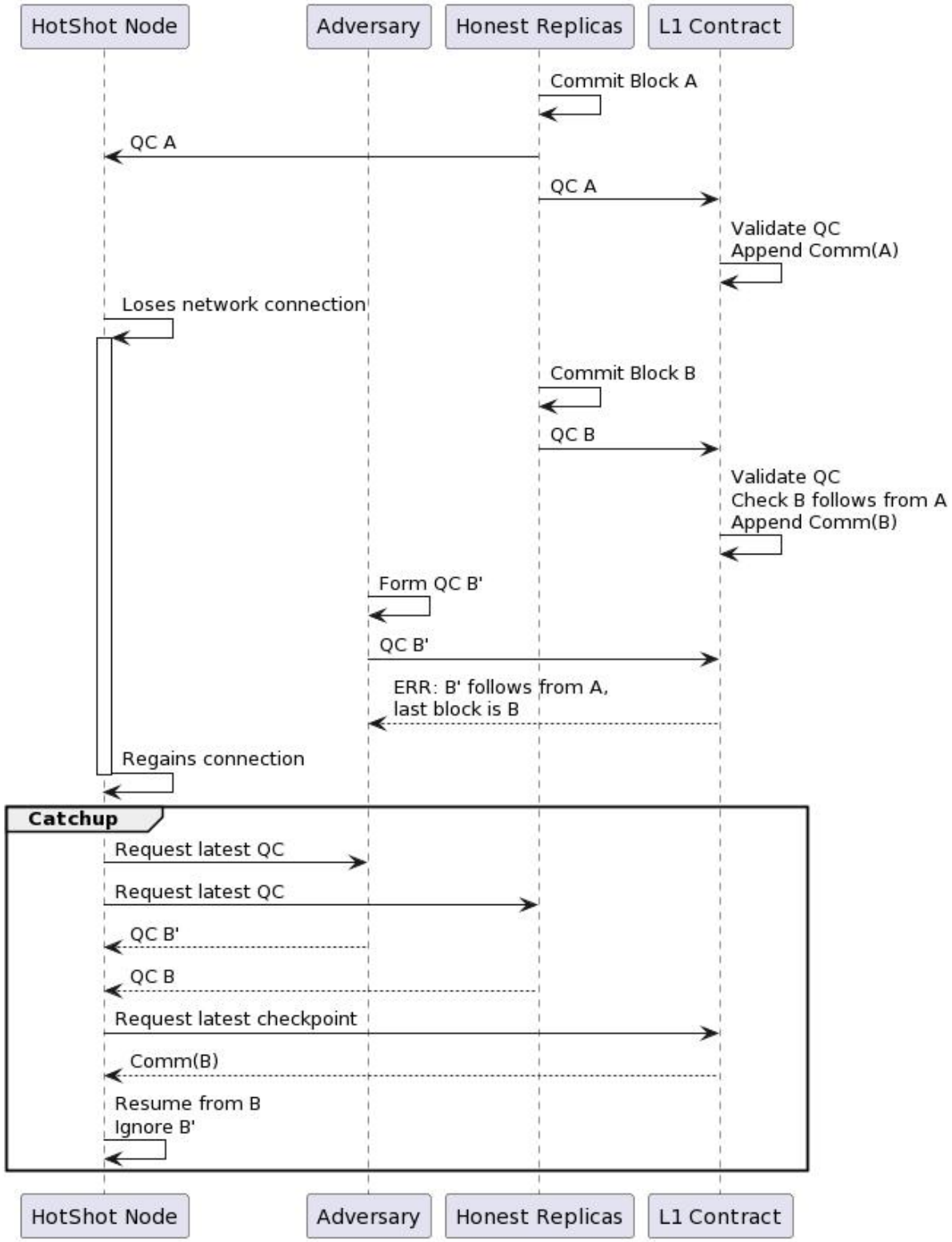


Fork choice between two conflicting QCs without L1 checkpoints

The adversary does this by taking advantage of an ambiguity which honest nodes must somehow resolve when they are joining the network for the first time or catching up after being offline for some time. In order to catch up, such nodes will look for signed QCs attesting to HotShot state changes, so they can figure out which state to sync with. Since they cannot tell the difference between honest peers and adversaries, and adversary who can provide QCs from the malicious fork faster than honest peers can provide QCs from the canonical fork can convince honest nodes in catchup to sync to the malicious fork. Over time, the adversary may convince enough honest nodes to switch that the malicious fork becomes the "canonical" one, and blocks which were committed in the old canonical fork are permanently lost.

By leveraging the L1 checkpoints, we can prevent an adversary from tricking honest nodes onto the wrong branch of a fork, even if the adversary can present signed QCs. We simply use the checkpointed state on L1 as a deterministic *fork choice rule*: when presented with conflicting QCs during catchup, an honest node will choose the one which is compatible with the state checkpointed on L1. In this way, the L1 checkpoint plays for the Espresso blockchain the same role that weak subjectivity checkpoints play for Ethereum.

Choosing a fork consistent with the latest L1 checkpoint only requires a node to check the history between the latest L1 checkpoint and the QCs justifying each branch of the fork, which should be a fairly small, bounded amount of work, assuming checkpoints are posted at a fixed rate and fairly frequently. This is far more practical than storing and validating the entire history, which a node would have to do in order to detect a fork arbitrarily far in the past without using a trusted checkpoint.



Fork choice between two conflicting QCs, facilitated by L1 checkpoints

In order to ensure that this fork choice rule uniquely specifies a branch, the L1 sequencer contract enforces a linear history. This is easy enough to do by storing the latest block commitment and checking that each new block specifies the previous block commitment as its parent (and has the correct height). This is difficult to do in a node because it requires the node to receive every single block in order, but in practice nodes are not always online, and when they recover from an outage they catch up immediately to the head of the chain without validating consensus for every intermediate block. The contract, on the other hand, does behave as if it is always online and directly validates the QC for every block consecutively, because **anyone** can drive the contract forward by providing a block and a QC to append, so we only require that **some** honest node is online for every block in order for the contract to remain live.

By forcing the L1 checkpoints to be linear and ensuring that honest nodes always work on a fork which is compatible with those checkpoints, we gain L1-level finality for any blocks which have been included in an L1 checkpoint. Note that this checkpointing scheme does *not* prevent an adversary from causing a fork—the L1 HotShot contract will not distinguish between "malicious" and "canonical" forks when appending a new block, so long as that block has a valid QC and follows from its parent. It will take whichever block it sees first. However, it will ensure that once a block is added to the contract, its fork *becomes* canonical, and the block will forever remain in the canonical chain.

Bridging

Rollup state updates facilitate interoperability between the layer 1 and the rollup. If the state of the rollup is verified and stored by the layer 1, then the layer 1 can also validate claims against that state, such as a claim that some tokens have been deposited into a bridge contract on the rollup. The L1 can also *write* to the state which is maintained by the L1, and the rollup can thus receive messages and tokens from the L1.

This is the idea used by the LX-to-LY bridge, which Polygon zkEVM uses to bridge ETH between the layer 1 and layer 2. In this design, part of the L1 state, a Merkle tree of messages to be sent to the L2, is represented directly in the L2 VM semantics. Since the canonical execution of L2 transactions happens in a smart contract on the L1, this executor is able to read from the appropriate L1 state when executing operations in the L2 VM.

Glossary of Key Terms

Data Availability Layer - Storage nodes that guarantee the availability and retrievability of transaction raw data and in charge of the eventual dissemination of the full data across the network.

Data Availability Proposal - The DA proposal is a proposal on a block for data availability. The consensus on a data availability proposal makes sure that enough parties have the block data. The DA proposal consists of the block and the corresponding view number.

Data Availability Votes - A DA vote is a vote indicating the receipt of either a DA proposal or a valid VID share. The vote contains the justify QC commitment, view number, block commitment, signature, and vote token. **Data Availability Certificate (DAC)** - DAC is a certificate that the proposed data is available to a quorum of distinct parties in the random small data availability (DA) committee. A DA leader, after receiving a sufficient number of votes on its DA proposal, assembles the votes into an optimistic DAC. **Data Retrievability Certificate**

- A retrievability certificate is a certificate that valid VID shares are available to a quorum of replicas. This ensures the liveness/data availability of the HotShot protocol even if the random small DA committee is bribed by an adversary. A DA leader assembles the retrievability certificate after receiving enough votes from the entire node/replica set.

Optimistic Data Availability Certificate - A optimistic DA certificate is a certificate that the proposed data is available to a quorum of distinct parties in the random small data availability (DA) committee. A DA leader, after receiving a sufficient number of votes on its DA proposal, assembles the votes into an optimistic DAC.

Commitment Proposal - A commitment proposal is a proposal on a block commitment. A block can only be applied to the chain if the consensus on the commitment proposal is reached. A commitment proposal is used as follows:

- A (consensus) leader, upon receiving a DAC for this view or sufficient quorum votes for the previous view, will construct a commitment proposal and multicast it to replicas. The commitment proposal will contain the block commitment, view number, height, justify QC, and the proposer ID.
- A node, after receiving the commitment proposal, if also gets either the DAC or the DA proposal, will validate the set and send the quorum vote back to the leader.\

Quorum Vote - quorum vote is a vote on the commitment proposal representing the status of the node/replica's decision on the proposal.

Yes Vote: is sent if the replica successfully verifies the commitment proposal. It consists of the justify QC commitment, view number, leaf commitment, signature, and vote token. *No Vote:* on the contrary, indicates the replica's rejection. A no vote has the same contents as a yes vote.

Timeout Vote: means the replica cannot decide due to timeout. Therefore, it does not include the leaf commitment as the yes or no vote does, but has all the other fields.

The node is not only signing on the data that is voted on but also the vote type, preventing a dishonest node from altering the vote. E.g., it is impossible to use the signature associated with a no vote to create a yes vote. **DA Vote** - A node vote on the data commitment, indicating the receipt of either the corresponding data proposal or a valid VID share.