Internal Functionality

This section describes in detail the internal workings of each component in Espresso.

Contracts Overview

Core Contracts:

- LightClient
- FeeContract
- StakeTable (not used for Mainnet 0)

Espresso Node

An *Espresso node* is a participant in the HotShot consensus protocol which also makes available some services to support L2 clients. The Espresso node is L2 agnostic: it does not provide services specifically tailored to any particular L2. It merely exposes all available information about HotShot and the log of blocks which have been sequenced. Any L2 may query this information and interpret the blocks according to its own execution rules in order to implement a prover, executor, RPC service, etc.

Internal components of an Espresso node and their possible usage by various participants in an L2

The main internal components of the Espresso node and their respective functions are:

- **HotShot node:** The component which actually runs consensus and communicates with other nodes.
- **HotShot query service:** The query service maintains a database containing the history and current state of HotShot, including all committed blocks and QCs, consensus-specific data like view numbers and stake tables, and status information like validator uptime. It populates this data using events provided by the HotShot validator and provides a REST API for querying the data. There is also a WebSockets-based API which allows clients to subscribe to notifications when new blocks and QCs are produced. The query service does not provide any L2-specific information. The contents of the blocks it provides are the generic transactions that HotShot itself understands.
- **Submit API**: The Espresso node also provides an interface allowing clients to submit transactions to the HotShot mempool. It takes as input a transaction serialized into bytes and an L2 identifier, and it wraps these into HotShot's generic transaction type.

Light Client Contract

The source code for the light client contract can be found on GitHub.

The *light client contract* is responsible for maintaining state about HotShot consensus on the layer 1 blockchain that Espresso (and L2 rollups) checkpoint to.

LightClient validates the HotShot state through cryptographic proofs (SNARK proofs). Rollups can use the new finalizedState from the contract to confirm the validity and finality of transactions that have been bundled and processed.

Copy

```
LightClientState public finalizedState;
```

Light Client State

This LightClientState includes information such as:

- the latest view number, viewNum, and block height, blockHeight, of the finalized HotShot chain
- the Merkle root of finalized block commitments, blockCommRoot

Copy

```
struct LightClientState {
  uint64 viewNum;
  uint64 blockHeight;
  BN254.ScalarField blockCommRoot;
}
```

The most important part of the LightClientState is the blockCommRoot, which is the root of an append-only Merkle tree of all blocks sequenced by Espresso. This root is public, allowing other contracts to use succinct Merkle proofs to verify the inclusion of a certain block commitment at a certain height and to update the root when new blocks are appended.

At any given time, the LightClientState contains Espresso block commitments from height 0 up to (but not including) the current block height, blockHeight. All commitments for heights greater or equal to blockHeight are not present — the corresponding leaf is empty in the tree.

There's another struct, StakeTableState, which is used to store the initial stake table commitments. The stake table is currently fixed, meaning it is initialized once and stored in the genesisStakeTableState variable when the Light Client contract is first deployed. The initial stake table commitments include the BLS and Schnorr verification keys and the amounts the validators staked.

Copy

```
  struct StakeTableState {
    uint256 threshold;
    BN254.ScalarField blsKeyComm;
    BN254.ScalarField schnorrKeyComm;
    BN254.ScalarField amountComm;
  }
```

However, in future versions, the stake table is will become dynamic. This will allow validators' stake commitments and key information to be updated in real-time, allowing the system to adjust to changes in stake amounts, the registration of new validators, or the withdrawal of existing ones.

Rollups and the Light Client Contract

Rollup contracts on L1 must use the blockCommRoot when validating a state transition to ensure that the rollup block claimed to have been executed is indeed the next block in the canonical sequence.

The proposer of a rollup state transition must provide a proof, relative to blockCommRoot, showing that the Espresso state commitment at a specified height is consistent with the rollup block commitment.

For rollups to integrate with Espresso, they need to modify their contract on L1 to prove that their state is derived from the Espresso state. For instance, consider a scenario where the Espresso prover pushes a new Espresso state commitment to the light client contract every 1 minute, and the rollup prover submits a new rollup block commitment every 10 minutes to their rollup contract. The rollup prover needs to prove that the block commitment it publishes to the rollup contract corresponds to all its rollup transactions contained in the Espresso blocks during the 10-minute period.

Each Espresso block commitment also commits to a list of rollup transactions (among other metadata) which facilitates lightweight proofs with transaction granularity for arbitrarily old rollup blocks. This approach allows for the verification of these rollups blocks on Espresso's chain and enables the light client contract to operate with a constant amount of storage, irrespective of the HotShot chain's length.

Data Availability

Since the actual block commitments (let alone the full blocks) are not stored on-chain, it is important to understand the data availability properties ensuring that clients can always retrieve an old block, block commitment, or a block's Merkle proof.

Clients can fetch a Merkle proof for any block from an archival query service and authenticate it against the block root in the light client state. Failing that, they can fetch the individual blocks from HotShot DA and extract the proof themselves.

ZK Proofs, SNARK Proofs, and ZK Circuits

A circuit defines the computation to be proven. Zero-knowledge proof (ZKP) systems can generate cryptographic proofs attesting to the validity of statements described by the circuit without revealing underlying witness. In our context, we rely on SNARK proofs (a special kind of ZKP), whose succinct nature is especially valuable for verifying computation in smart contracts where gas costs are a critical concern. Irrespective of the number of signatures/consensus votes, the size and verification cost of the proof remain constant.

In a zero-knowledge protocol there are two main roles: (i) the *prover* and (ii) the *verifier*.

The *prover* uses a combination of secret inputs (HotShot nodes' Schnorr signatures), also called witnesses, public inputs (the LightClientState), and a circuit description in order to generate a SNARK proof.

The *verifier* uses the public inputs and the SNARK proof to verify that the rules defined by the ZK circuit are satisfied. In this case, the LightClient contract acts as the verifier for this ZK proof via the verifyProof method which is invoked within the newFinalizedState function.

Finally, both the prover and verifier use some public parameters. These public parameters are derived from the circuit and a structured reference string (SRS) that requires a trusted setup to be generated but can be reused for other circuits.

The Light Client Circuit

Let the following circuit CqcCqc over prime field FpF*p*, the corresponding Jubjub curve group G=⟨g⟩G=⟨*g*⟩ whose scalar field is FrF*r* and base field is exactly FpF*p*, so that each group element ∈Fp×Fp∈F*p*×F*p*.

- Public input:
  - stake table commitment: cmT∈FpcmT∈F*p*
    - A stake table entry now composes of a triple (bls_ver_key,schnorr_ver_key,stake_amount)(*bls_ver_key*,*schnorr_ver_key*,*stake_amount*). BLS verification key is under ark_bn254::Fq, Schnorr verification key is under ark_bn254::Fr, and the stake amount is within the range of ark_bn254::Fr. The commitment should be computed in the following way: first serialize all BLS keys into elements of ark_bn254::Fr, follows by a list of Schnorr keys and then the stake amount. The commitment is the rescue hash of this list.
  - quorum Threshold: T∈Fp*T*∈F*p*
  - attested new finalized hotshot state: m:=(v,h,rootcm,cmledger,cmstake_table)∈Fp5*m*:=(*v*,*h*,rootcm,cmledger,cmstake_table)∈F*p*5
    - the merkle tree for block commitments can use any hash function (e.g. SHA2) and best if there is an injective mapping between the root value to a FpF*p* element.
- Secret witness:
  - signers indicator vector: v⃗Sv*S*
  - stake table vector (consists of public key, weight pair): T=[(pki∈G,wi∈Fp)]i∈[n]T=[(*pki*∈G,*wi*∈F*p*)]*i*∈[*n*]
  - list of schnorr signatures: {σi=(si,Ri)∈Fr×G}i∈[n]{*σi*=(*si*,*Ri*)∈F*r*×G}*i*∈[*n*]
- Relation:
  - the input signers indicator vector v⃗Sv*S* is a bit vector
  - correct stake table commitment: cmT=commit(T)cmT=commit(T)
    - we use Rescue-based commitment, thus all operations are native
  - accumulated weighted sum exceeds threshold: ∑i∈Swi>T∑*i*∈*Swi*>*T*
    - assumption: there's no overflow!! NOTE: outside the circuit, the client software that's in charge of stake table management needs to check the accumulated sum does not exceed modulus p*p* AT ALL TIMES! ⚠️

- signature verification (on each): Vfy(pki,m,σi)=1Vfy(*pki*,*m*,*σi*)=1 for all ∀i∈[n]∀*i*∈[*n*].
    - c=H(R,m,..)*c*=*H*(*R*,*m*,..): rescue-based hash to get challenge
    - x=gs*x*=*gs*: a fixed-base scalar mul
        - internally, involves bit-decomposition of s*s* and elliptic curve addition based on the bits.
    - y=R+pkc*y*=*R*+*pkc*: a variable-base scalar multiplication + an elliptic curve addition.
    - x=?y*x*=?*y*: point equality check

Updating and Verifying LightClientState

The LightClientState is updated by any state prover that submits valid updates to the LightClient contract via the newFinalizedState method which sets the latest finalizedState.

Copy

```
function newFinalizedState(
   LightClientState memory newState,
   IPlonkVerifier.PlonkProof memory proof
) external {
  //...
}
```

We assume an *altruistic, honest prover* for now and leave the design of prover market to future work.

For replicas:

- upon receiving new QC from the leader, generate a Schnorr signature over the updated finalized HotShot state, and send it over to the CDN *and* store it locally for a while.
- the local storage for the Schnorr signatures (for each block) can be a sliding window of a fixed size where older signatures got pruned. The window size can be set based on the expected interval for on-chain update plus some buffer accounting for temporarily failing prover.

For the altruistic prover:

- will continuously listen passively for the HotShot state changes, e.g. when a new block has been decided
- periodically requests the Schnorr signatures from the DA layer by sending a request on a specific view v*v*. This will be the signature for the new finalized state from the consensus nodes

    - For convenience of signature collection, prover will first fetch from CDN (we refer to as the relay server) for the list of Schnorr signatures, if failed, then ask each individual replica (note: not small DA committee, or VID, but each node individually).
- Once a sufficient amount of valid signatures is collected, some provers can then generate a SNARK proof which is submitted alongside the new finalized state to the light client contract.

The contract has the ability to be in permissioned mode where it only accepts proofs from one prover. For the next release, only a permissioned prover, doing the computations, will call this function.

Replica nodes update the snapshot of the stake table at the beginning of an epoch and this snapshot is used to define the set of stakers for the next epoch. The light client state must be updated at least once per epoch.

For the next release, we are not using epochs so numBlockPerEpoch is set to type(uint32).max during deployment.

HotShot state authentication via Schnorr signatures

When a set of HotShot nodes reach consensus and the finalized HotShot state has been updated, they each sign a Schnorr signature on this updated HotShot state. These signatures assert that the signer agrees with the state of each proposed block. The signatures are stored locally on the DA layer, and to save space, older signatures are pruned using a sliding window mechanism. The window size can be set based on the expected interval for on-chain update plus some buffer accounting for a temporarily failing prover.

When a prover (an entity that confirms the truth of a claim) retrieves these signatures, a SNARK proof is then generated. This proof, is used by the LightClient contract to efficiently verify these Schnorr signatures. The state of the sequencer contract can be updated only if a correct SNARK proof is provided. This is a critical step that ensures the validity and security of state updates in Espresso's consensus protocol.

The proof of the Schnorr signatures is sent to the newFinalizedState function of the LightClient contract.

Verifying the Signatures and Light Client State

The LightClient contract also does the work of verifying the proof that is sent by the prover on L1. The verifyProof method accepts the proof and a set of public inputs (the LightClientState) to check whether the proof correctly verifies the new state being submitted.

Copy

```
function verifyProof(LightClientState memory state, IPlonkVerifier.PlonkProof memory proof)

    internal

    virtual

  {

    IPlonkVerifier.VerifyingKey memory vk = VkLib.getVk();
```

```
        uint256[] memory publicInput = preparePublicInput(state);


        if (!PlonkVerifier.verify(vk, publicInput, proof, bytes(""))) {
            revert InvalidProof();
        }
    }
```

Verifying a SNARK proof requires a constant amount of space and computation, no matter how many HotShot node signatures are involved. This is unlike verifying the signatures directly, which would require space and computation proportional to the number of signers.

The proof itself contains the HotShot state, the stake table info and the list of Schnorr signatures of the HotShot nodes that formed a Quorum and came to consensus on that state.

This verifyProof method is executed when the newFinalizedState method is called so that the new state is accepted only if the proof succeeds.

Escape Hatch Functionality

Rollup contracts keep track of their rollup VM states and depend on our Light Client contract for finalized consensus states. They usually support escape hatches in case of liveness failures on L2. Since different rollups impose different predicates when deciding whether L2 is down, our Light Client contract provides some helper functions to detect delays in HotShot updates. The Rollup contracts can then implement their escape hatch logic based on this info.

Two such helper functions are:

- function getHotShotCommitment(uint256 hotShotBlockHeight)
    - Returns the root of the HotShot block commitment tree, where each leaf contains the HotShot block commitment at a new height
- function lagOverEscapeHatchThreshold(uint256 blockNumber, uint256 threshold)
    - Returns whether there has been delay between updates
    - Checks if the HotShot state updates lag behind the specified threshold based on the provided L1 block number.
    - The rollup chooses the threshold based on their liveness criteria.
    - HotShot would be considered down if the gap between two consecutive updates where the provided L1 block number should have been recorded, exceeds the specified threshold

Periodically, the light client contract is updated with the latest validated HotShot state. We store a 10 day sliding window of historical Hotshot state roots, stateHistoryCommitments, so that optimistic rollups' dispute handling contracts can access required HotShot commitment data during disputes (which is usually during 7 day windows).

Public Write Methods

newFinalizedState

This method updates the latest finalized light client state. It is updated per epoch. An update for the last block for every epoch has to be submitted before any newer state can be accepted since the stake table commitments of that block become the snapshots used for vote verifications later on.

The contract has the ability to be in permissioned mode where there is only one prover that has the ability to call this function. In the next release, only a permissioned prover doing the computations will call this function

Copy

```
function newFinalizedState(LightClientState memory newState, IPlonkVerifier.PlonkProof memory proof)
  external;
```

Parameters

Name

Type

Description

newState

LightClientState

new light client state

proof

IPlonkVerifier.PlonkProof

Plonk proof

computeStakeTableComm

Given the light client state, compute the short commitment of the stake table

Copy

```
function computeStakeTableComm(LightClientState memory state) public pure returns (bytes32);
```

Fee Token Contract

The source code for the fee token contract can be found on GitHub.

The fee token contract enables builders to deposit ETH which allows them to pay for a data processing fee associated with HotShot. While the fee token contract facilitates deposits, HotShot itself manages and tracks the working balance of builder deposits. Initially, the fee token contract only supports deposits. Withdrawals are planned to be enabled in a future release.

Copy

```
function deposit(address user) public payable {
  //...
}
```

The contract defines a minimum and maximum amount for deposits to avoid errors and prevent the fee table from being filled with dust.

Copy

```
uint256 public immutable MAX_DEPOSIT_AMOUNT = 1 ether;

uint256 public immutable MIN_DEPOSIT_AMOUNT = 0.001 ether;
```

In the Mainnet 0 release, withdrawals are not enabled and thus the MAX_DEPOSIT_AMOUNT aims to minimize how much ETH is locked by a builder.

Public Write Methods

deposit

Allows anyone to deposit an ETH balance for any user

the deposit amount is less than a specified threshold to prevent accidental errors

Copy

```
function deposit(address user) public payable;
```

Stake Table

The s*take table contract* maintains the canonical stake table for HotShot and enables nodes to stake their tokens to participate in HotShot. This contract will be responsible for:

- staking
- unstaking
- delegating

How the Stake Table Contract Works

The HotShot contract, which verifies and stores Espresso state updates, also stores the entirety of the latest HotShot stake table. By taking this to be not just a mirror of the stake table but the canonical version of it, we gain a number of advantages, including the potential for the L1 itself to make changes to the stake table, a requirement for implementing restaking.

Restaking

Restaking is a critical part of the Espresso roadmap, since it allows Espresso to share a security budget with Ethereum and aids in incentive alignment between the Ethereum operator set and rollups integrated with Espresso. But to enable restaking using a system like Eigenlayer, the restaking contracts must be able to change the state of the Espresso stake table, such that new entries can be added when new L1 validators opt into restaking for Espresso.

By treating the stake table stored on the L1 as the canonical stake table, we allow L1 smart contracts to do exactly this, merely by writing to another contract on the same L1. Espresso nodes will then read the updated stake table back from the L1 at the start of the next epoch and begin using the stake table with the restaker's entry for validating future consensus decisions.

This does require every HotShot consensus participant to run an L1 light client in order to make trustless reads from the HotShot smart contract. However, Ethereum's proof-of-stake consensus enables very efficient, lightweight, trustless clients, and this Ethereum client can be bundled directly into the HotShot client executable for a seamless user experience.

Consensus Sync

In order to verify consensus decisions, a HotShot client must know the stake table that was used to authenticate each decision, so that it can accurately account for the number of votes endorsing each decision. This means that, naively, if a new HotShot client or consensus participant wanted to sync with the current state of consensus, it would have to replay from genesis at least every block which updated the stake table. By storing verified snapshots of the stake table on a trusted L1, new HotShot nodes can instead read the latest snapshot from the L1 and start replaying blocks from there, trusting the L1 validator set to have already verified each block which led up to the snapshot. This is very similar to how rollups can enable fast, trustless sync for their clients by leveraging L1 state updates.

Smart Contract Upgradeability

The following smart contracts are upgradeable:

- LightClient
- FeeContract

These contracts use the *universally upgradeable proxy pattern (UUPS)* to make it possible to upgrade functionality in the contract, e.g., adding a new method for a future launch.

How it works

A proxy contract directs calls to the implementation contract, which contains the logic of the system.

When an upgrade is needed, a new implementation contract is deployed and the proxy contract's storage is updated so that it will now route requests to the new implementation. This allows for modifications to be made without affecting the state stored in the contract. Espresso users can continue interacting with the same contract address (the address of the proxy) to access the updated functionalities of the implementation contract. Careful consideration will be made to ensure backward compatibility and data consistency during the upgrade process.