Security Audit Report

Espresso Q1 2025: Epoch Change Protocol

Last revised 09.04.2025

Authors: Tatjana Kirda, Josef Widder, Martin Hutle, Hernán Vanzetto

Contents

Audit Overview

The project

In March 2025, Espresso Systems engaged Informal Systems to work on a partnership and conduct a security audit of the following items in espresso-network software:

1. Introduction of epoch change function in the HotShot protocol.

Relevant code commits

The audited code was from:

- commit hash 1ed39cc0a835d34653ccbc36993e00548bbdf72f

Scope of this report

The audit conducted an evaluation of the correctness and liveness properties associated with the modifications introduced through the epoch change mechanism in the HotShot protocol. The review focused on the examination of the code additions that support epoch changes, paying special attention to edge cases defined in the threat model.

Audit plan

The audit was conducted between March 13th, 2025 and April 4th, 2025 by the following personnel:

- Hernán Vanzetto
- Josef Widder
- Martin Hutle
- Tatjana Kirda

Conclusions

After conducting a thorough review of the project, we found it to be carefully designed and generally well-implemented.

We found that the mapping between the specification and the implementation was not obvious, so we documented in the section “Correspondence Between Specification and Implementation” how the specified functions are represented in the code, and checked whether the discrepancies cause any issues. In the end, we used the implementation to understand the protocol where documentation was lacking detail. Based on all the information, we formulated potential scenarios that could happen during the epoch change, and analyzed them in the “Threat model” section.

As a result, in addition to the mentioned analysis section, during the audit, 4 findings were documented. One is classified as critical severity, the others as informational severity. Detailed descriptions of these issues are available on the “Findings” section.

Audit Dashboard

Target summary

- **Type:** Protocol and Implementation
- **Platform:** Rust
- **Artifacts:**
  - Changes related to epoch change in folders:
    - hotshot-task-impls/src/consensus
    - hotshot-task-impls/src/quorum_proposal
    - hotshot-task-impls/src/quorum_proposal_recv
    - hotshot-task-impls/src/quorum_vote
  - Additional helper function related to the epoch change in files:
    - hotshot-task-impls/src/helpers.rs
    - hotshot-task-impls/src/vid.rs
    - types/src/v0/impls/stake_table.rs
    - hotshot-types/src/utils.rs
    - hotshot-types/src/consensus.rs
    - hotshot-types/src/data.rs

Engagement summary

- **Dates**: March 13th, 2025 - April 4th, 2025
- **Method**: Manual code review, protocol analysis

Severity summary

Finding Severity	Number
Critical	1
High	0
Medium	0
Low	0
Informational	3
Total	4

System Overview

HotShot is the Consensus Protocol that is employed in the Espresso Sequencer. It works together with a DA layer to order transactions for the L2 rollup.

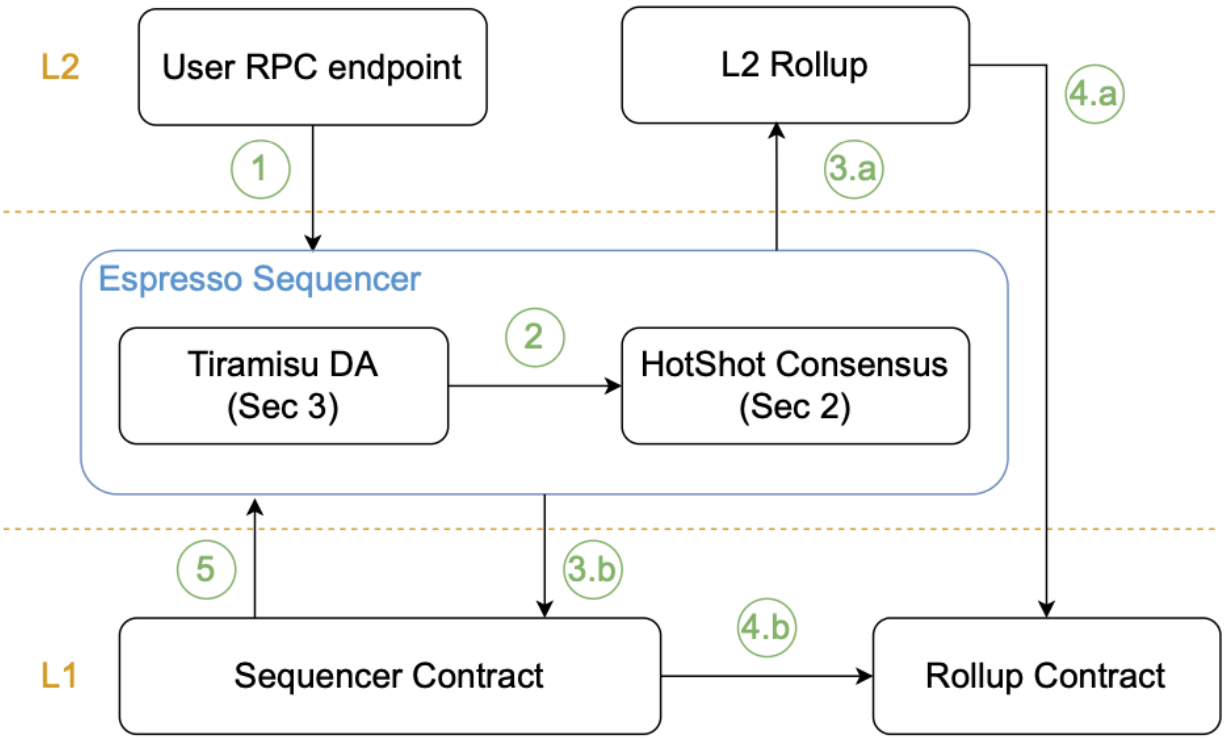

Figure 1: Espresso Sequencer

The HotShot protocol is based on a chained version of the HotStuff-2 consensus protocol. In incorporates the following concepts that are important for our analysis:

- There is no fixed set of nodes but a dynamic changing set of validators nodes in a proof-of-stake setting. Every `K` blocks, a new epoch is started, the validator set and stake table for each epoch `e+1` is determined at the beginning of epoch `e` and stored on the L1 chain.
- As all protocols of the HotStuff family, HotShot operates in views. The views and a leader for each view are provided by a pacemaker protocol that manages timeouts, view sync and leader election. To enable bribery resistance, leader election and view sync are not deterministically determined but derived from a decentralized random beacon.
- Threshold cryptography is used to build a certificate with lower communication complexity.
- Data propagation (transactions, blocks) is deferred to a data availability layer.

Focus of this audit is the epoch change protocol, i.e. the modification to the consensus algorithm to allow transitioning from an epoch `e` to another epoch `e+1` with a different validator set and stake table.

Epoch transition description

Each epoch consists of constant number `K` of blocks. The staking table for each epoch `e` is known at the beginning of epoch `e-1`. Stake tables are stored in L1 contract. Nodes in both epochs `e` and `e+1` have access to both stake tables.

Transition from epoch `e` to `e+1` in a favorable run:

1. The leader `l_v` of view `v` sends a quorum proposal for the last block `B_k` of epoch `e` to all nodes in `e` and `e+1`.
2. The nodes in `e` and `e+1` validate the proposal and send the votes to `l_v`. The nodes store the last block certificate `C(B_k')` as highQC.
3. The leader of view `v+1` in epoch `e` accumulates the votes. The QC1 for `B_k` can be formed once the thresholds for `e` and `e+1` are reached. The leader sends the quorum proposal using the new formed QC1 to the nodes in epoch `e` and `e+1`.
4. The nodes in `e` and `e+1` store the QC1 they receive from the leader as `highQC` and send their vote for the proposal of the leader of `v+1` to the leader of `v+2`.
5. The leader of view `v+2` accumulates the votes and form the new certificate when both thresholds have been reached. The leader sends a new proposal with the new formed QC2 to the nodes in `e` and `e+1`. This is the proposal for the extended quorum certificate.
6. The nodes in `e` and `e+1` store the QC2 as `highQC` and send their votes for the extended proposal to all nodes in epoch `e+1` (only).
7. Each node in `e+1` accumulates the votes from the nodes in `e` and `e+1`. Once the threshold is reached the extended quorum certificate eQC is formed and the block `B_k` is decided. The leader of `v+3` of epoch `e+1` builds a new block `B_{k+1}` and sends it as a proposal with eQC as justification to all nodes in `e+1`.

View changes by timeouts and view synchronization

- If a view change is not based on the quorum certificates (i.e., nodes do not have a QC for views `v`, `v+1` or `v+2`, but progress based on timeout certificates or view synchronization certificates, they need to restart in step 1 of the protocol.
- The synchronization across epochs is done among the nodes in `e+1` using the Naor-Keidar procotol (as for normal view synchronization) but with the eQC as justification and with the nodes from `e` as trigger.

Correspondence Between Specification and Implementation

This document outlines the relationship between the pseudo-code in the consensus protocol specification and its current Rust implementation. While the high-level flow is similar, the mapping between pseudo-code and implementation is not always exact. The Rust code addresses complexities that are not present in the specification, employs different control flows, and addresses additional edge cases. We aim to align the simplified pseudo-code with the code references and highlight significant differences.

Implementation of the specified functions

Function `certificate(type, data, view)`

```
function certificate(type, data, view):
    c.type ← type
    c.data ← data
    c.view ← view
    c.signatures ← votes_map[(vote.type, vote.data, vote.view, current_epoch)]
    if data.height mod K == 0:
        // add optional signatures from the next epoch when applicable
        c.next_epoch_signatures <- votes_map[(vote.type, vote.data, vote.view, next
    return c
```

- Partially defined in the function `create_signed_certificate`. The implementation depends on the type of the certificate, and it's defined in the `hotshot-types/src/simple_certificate.rs` file.
  - It's called from the functions `build_cert` and `accumulate`.
- `if data.height mod K == 0` section is implemented in the function `is_last_block_in_epoch`. This function is used in several places to handle epoch transitions, the relevant for this context being in `quorum_vote/handlers.rs`, when submitting votes, function `submit_vote` (code ref).
- Votes must be accumulated to create the certificate. In the `submit_vote` function, a node can vote only if it is a committee member in either epoch `e` or `e+1` (code ref). The votes are processed in `consensus/handlers.rs`, where they are collected in both the current and next epoch vote collectors (code ref).

Function `vote(type, data, view)`

```
function vote(type, data, view):
    v.type <- type
    v.data <- data
    v.view <- view
    v.signature <- signVote(type, data, view)
    return v
```

Implemented in the function `create_signed_vote`, which is called from the function `submit_vote` function (code ref), as well as in multiple points in the `hotshot-task-impls/src/view_sync.rs` file.

Function `validSignature(data, signature, public_key)`

```
function validSignature(data, signature, public_key):
    if commit(data) == decrypt(signature, public_key)
        return true
    return false
```

The `validSignature` function is called during both proposal validation and vote validation.

- When validating a proposal, this function maps to the `validate_signature` function that checks the signature against the proposal data. This function is called from the function `validate_proposal_view_and_certs` (code ref).
- When receiving the votes, the vote signature is validated as part of the `accumulate` function (code ref).

Note that the Rust implementation does not have a direct one-to-one correspondence with this function.

Function validVote(vote, node)

```
function validVote(vote, node)
// old vote
    if vote.view < current_view
        return false
    // This a vote for the last block in epoch
    if vote.data.height mod K == 0:
        epoch <- vote.data.height / K
        // Votes for the eQC go to every node in e+1
        // This is not a vote for the eQC, it should go to the leader in the old epoch
        if vote.type != "extended":
            if we are not leader of vote.view + 1 in epoch
                return false
            // Nodes from both epochs can vote for the last block in epoch
            if node not in epoch and node not in epoch+1:
                return false
    else
        epoch <- vote.data.height / K + 1
        if we are not leader of vote.view+1 in epoch
            return false
        if node not in epoch:
            return false
        if not validSignature((v.type, v.data, v.view), v.signature, node.public_key)
            return false
        return true
```

The function `handle_quorum_vote_recv` handles the validation of the vote.

- The vote view is validated in the `accumulate_vote` function (code ref).
- When validating the leader in the function `handle_quorum_vote_recv` (code ref), the utilized epoch is retrieved from `vote.data.epoch` (code ref).
- As previously mentioned, the vote signature is validated as part of the `accumulate` function (code ref).

Function accumulateSignatures(vote, node)

```
function accumulateSignatures(vote, node):
    // votes_map has a static lifetime
    // votes_map maps tuple (type, data, view, epoch)
    // to accumulated signatures
    epochs = []
    if vote.data.height mod K == 0:
        epoch <- vote.data.height / K
        if node in epoch:
            epochs.add(epoch)
        if node in epoch+1:
            epochs.add(epoch+1)
    else:
        epoch <- vote.data.height / K + 1
        if node in epoch:
```

```
            epochs.add(epoch)
    for epoch in epochs:
        if votes_map[(vote.type, vote.data, vote.view, epoch)] is not empty:
            signatures <- votes_map[(vote.type, vote.data, vote.view, epoch)]
            // adding a signature to signatures means:
            // cryptographically accumulate partial signatures
            votes_map[(vote.type, vote.data, vote.view, epoch)] <- signatures + vote.s
        else:
            votes_map[(vote.type, vote.data, vote.view, epoch)] <- vote.signature
```

`if data.height mod K == 0:`

- Implemented in the function `is_last_block_in_epoch`. This function is used in multiple locations in the code to handle epoch transitions, notably in `consensus/handlers.rs`. When handling votes, the function `handle_quorum_vote_recv` uses the `transition_indicator` and utilizes the function `is_high_qc_for_last_block` (code ref).

`epoch <- vote.data.height / K` and `epoch <- vote.data.height / K + 1`:

- The epoch calculation is implemented in the `option_epoch_from_block_number` function. When accumulating the votes, the epoch from the vote sender is used (code ref). When the vote is created, the epoch is determined using the `option_epoch_from_block_number` function (code ref).

Votes for both the current and next epochs are handled in the function `handle_quorum_vote_recv` when `handle_vote` is called (code ref). This function handles the creation of the vote collector and uses the appropriate `handle_vote_event` implementation based on the vote type being processed. Current and next epoch votes are accumulated separately in separate vote collectors. The function `accumulate` accumulates the voters' stake (code ref) and the signatures (code ref). The mapping between a voter's public key and their vote information are stored in the vote accumulator (code ref).

Function `thresholdReached(vote)`

```
function thresholdReached(vote):
    // comparison checks whether we have enough accumulated partial signatures
    two_epochs_threshold = false
    if vote.data.height mod K == 0:
        two_epochs_threshold = true
        epoch <- vote.data.height / K
    else:
        epoch <- vote.data.height / K + 1
    if two_epochs_threshold:
        votes_map[(vote.type, vote.data, vote.view, epoch)] >= CURRENT_EPOCH_T
            and votes_map[(vote.type, vote.data, vote.view, epoch+1)] >= NEXT_EPOC
    else
        votes_map[(vote.type, vote.data, vote.view, epoch)] >= CURRENT_EPOCH_T
```

The Rust implementation uses separate vote collectors for the current and next epochs and checks thresholds independently in each collector.

The `accumulate` function checks if enough votes have been collected to reach the threshold (code ref). The threshold is determined by the `threshold` function, which is implemented differently for each certificate type. In the current implementation, the success threshold is determined based on the number of nodes participating in the epoch, and it is compared to the stake of the voters, not the signers.

Function `proposal(type, data, view, cert)`

```
function proposal(type, data, view, cert):
```

```
    p.type <- type
    p.data <- data
    p.view <- view
    p.justify_cert <- cert
    p.signature <- signProposal(type, data, view, cert)
    return p
```

Implemented as part of the `publish_proposal` function (code ref).

Function `formProposal(cert)`

```
function formProposal(cert):
    // This is the last block in the current epoch
    if cert.data.height mod K == 0:
        // We've just formed eQC, this means we are leader in the next epoch
        // Obtain a new block and use eQC as justification
        if cert.type == "extended":
            type <- "quorum"
            data <- obtain a new block
        // We're still forming eQC, reuse data
        else:
            data <- cert.data
            // high_qc and locked_qc are for the same block
            // and we are in the third consecutive view.
            // We can try to form eQC
            if high_qc.data == locked_qc.data
                and current_view - 1 == high_qc.view
                and high_qc.view - 1 == locked_qc.view:
                    type <- "extended"
            else:
                type <- "quorum"
        view <- current_view
        justify_cert <- cert
    // This is NOT the last block in the current epoch
    else:
        data <- obtain a new block
        view <- current_view
        justify_cert <- certfunction formProposal(cert
    return proposal(type, data, view, justify_cert)
```

This function is partially implemented as a component of the `publish_proposal` function (code ref). The function `is_qc_forming_eqc` returns true if the QC is for the last block in the epoch and it's not an eQC. If the `is_qc_forming_eqc` returns true, the block header is reused, otherwise, a new block is used.

When forming the proposal, there is no explicit setting of the certificate type. In the code, the extended QC is formed after enough votes have been accumulated.

- The `justify_cert <- cert` part is implemented when creating the proposal (code ref). The `parent_qc` is retrieved in function `handle_dep_result` (code ref).
- The `view <- current_view` component is implemented when creating the proposal (code ref). The `view_number` is part of the state of `ProposalDependencyHandle` (code ref).

Function `broadcast_proposal(proposal)`

```
function broadcast_proposal(proposal):
    if proposal.data.height mod K == 0:
```

```
        epoch <- proposal.data.height / K
        broadcast proposal to nodes in epoch and epoch+1
    else:
        epoch <- proposal.data.height / K + 1
        broadcast proposal to nodes in epoch
```

The proposal is broadcasted from the function `publish_proposal` (code ref).

The `parse_event` function determines the type of the transmit (code ref). The broadcast section doesn't differentiate between the end of the epoch and the regular proposal. That distinction is implemented in the vote handling logic, specifically in the `submit_vote` function, based on the `extended_vote` flag (code ref). The votes would be broadcasted to everyone in the case of `HotShotEvent::ExtendedQuorumVoteSend` (code ref), while the `HotShotEvent::QuorumVoteSend` vote is sent only to the leader (code ref).

Function `validProposal(proposal, node)`

```
function validProposal(proposal, node):
    if proposal.view < current_view
        return false
    if not validSignature((proposal.type, proposal.data, proposal.view, proposa.just
        return false
    if not validLeader(proposal, node):
        return false
    if proposal.type == "extended":
        if not validExtendedProposal(proposal):
            return false
    if not validJustifyCert(proposal):
        return false
    return true
```

In the implementation, there is no specific implementation of a `validProposal` function. Instead, if all the validation checks pass without returning an error, the function completes successfully, and the proposal is considered valid. The function is mostly implemented in the `handle_quorum_proposal_recv` function (code ref):

- The function `validate_proposal_view_and_certs` handles the validation of:
  - `if proposal.view < current_view` (code ref),
  - `if not validSignature((proposal.type, proposal.data, proposal.view, proposal.justify_cert), proposal.signature)` (code ref). This function was previously discussed.
- The function `validate_proposal_safety_and_liveness` handles the validation of `if proposal.type == "extended" && if not validExtendedProposal(proposal)` (code ref). It checks if a leaf can form an extended QC by verifying that it's part of a chain of three consecutive views for the same block.
- The function `validate_qc_and_next_epoch_qc` verifies the `if not validJustifyCert(proposal)` section. It's called from the `handle_quorum_proposal_recv` function (code ref) and in case `ExtendedQcRecv` has been received(code ref).

The condition `if not validLeader(proposal, node)` is checked in `quorum_proposal/mod.rs` when creating a dependency task (code ref) and in `publish_proposal` in `quorum_proposal/handlers.rs` (code ref). Note that the logic of this function is not implemented on the proposal receiver side but when creating the proposal. Pseudo code function is studied in the following section.

Function `validLeader(proposal, node)`

```
function validLeader(proposal, node):
    // This is not the last block in epoch
    if proposal.data.height mod K != 0:
        // calculate epoch number from the block height
```

```
        epoch <- proposal.data.height / K + 1
        if node is not leader of proposal.view in epoch:
            return false
    // This is the last block in epoch
    else:
        // calculate epoch number from the block height
        epoch <- proposal.data.height / K
        if node is not leader of proposal.view in epoch:
            return false
    return true
```

This function is implemented as a section in the `publish_proposal` function (code ref).

- The epoch calculation based on block height is implemented in the `option_epoch_from_block_number` function, which handles both cases (last block in epoch or not). The actual leader determination is implemented in the `Membership` trait, which has a `leader` method.

Function `validJustifyCert(proposal)`

```
function validJustifyCert(proposal):
    cert <- proposal.justify_cert
    // This is a proposal for the last block in epoch
    if proposal.data.height mod K == 0:
        epoch <- proposal.data.height / K
        if cert.type == "extended":
            return false
        if proposal.data == cert.data:
            if cert is not correctly signed by quorum of nodes in epoch
            or cert is not correctly signed by quorum of nodes in epoch+1:
                return false
        else:
            if not safeProposal(proposal):
                return false
            if cert is not correctly signed by quorum of nodes in epoch:
                return false
    // This is a proposal for the first block in epoch
    else if proposal.data.height mod K == 1:
        epoch <- proposal.data.height / K + 1
        if cert.type != "extended":
            return false
        if not safeProposal(proposal):
            return false
        if cert is not correctly signed by quorum of nodes in epoch
        or cert is not correctly signed by quorum of nodes in epoch-1:
            return false
    // Not first, not last block in epoch
    else:
        epoch <- proposal.data.height / K + 1
        if cert.type == "extended":
            return false
        if not safeProposal(proposal):
            return false
        if cert is not correctly signed by quorum of nodes in epoch:
            return false
    return true
```

This function doesn't have a precise mapping in the code. The implementation deviates from the pseudo code, and

its sections are distributed across the `handle_quorum_proposal_recv` function.

- The implementation doesn't differentiate between eQC and regular QC.
- The function `validate_qc_and_next_epoch_qc` handles the certificate signature validation (code ref) for the specified QC's epoch. The function distinguishes between current and next epochs by checking for the presence of `next_epoch_qc` (code ref). The epoch is taken from `qc.data.epoch` and is not calculated. The validation of the `proposal.data == cert.data` condition is implemented in case the `next_epoch_qc` exists (code ref) as well as signature validation for the next epoch (code ref).
- `if not safeProposal(proposal)` is implemented in `validate_proposal_safety_and_liveness` with safety and liveness checks (code ref). Note that both the specification and implementation require either safety or liveness conditions to be met.

`cert <- proposal.justify_cert` is implemented in `handle_quorum_proposal_recv` (code ref).

Function sendVote(vote)

```
function sendVote(vote):
    if vote.data.height mod K == 0:
        epoch <- vote.data.height / K
    else:
        epoch <- vote.data.height / K + 1
    if vote.type == "extended":
        send vote to leader of current_view+1 in epoch+1
    else:
        send vote to leader of current_view+1 in epoch
```

The vote is created in the function `submit_vote` (code ref) and then added to the storage (code ref). Depending on the type of the sent vote, the event `HotShotEvent::ExtendedQuorumVoteSend` or `HotShotEvent::QuorumVoteSend` is emitted (code ref). The vote would be broadcast to everyone in the case of `HotShotEvent::ExtendedQuorumVoteSend` (code ref), while the `HotShotEvent::QuorumVoteSend` vote is sent only to the leader (code ref).

Function safeProposal(proposal)

```
function safeProposal(proposal):
    if proposal.data extends locked_cert.data // safety rule
        or proposal.justify_qc.view > locked_cert.view: // liveness rule
        return true
    return false
```

This function can be mapped to the section of implementation in the `validate_proposal_safety_and_liveness` function (code ref):

- `proposal.justify_qc.view > locked_cert.view` (code ref).
- `proposal.data extends locked_cert.data` (code ref).

Function updateCerts(cert)

```
function updateCerts(cert):
    high_qc <- cert
    justify_high_qc <- fetchProposal(high_qc).justify_cert
    justify2_high_qc <- fetchProposal(justify_high_qc).justify_cert
    if high_qc.view - 1 == justify_high_qc.view:
        if justify_high_qc.view > locked_qc.view:
            locked_qc <- justify_high_qc
    if justify_high_qc.view - 1 == justify2_high_qc.view:
        if justify2_high_qc.view > decided_qc.view:
```

```
            decided_qc <- justify2_high_qc
```

The mapping of this function in the implementation differs significantly due to the chained nature of the algorithm. When a proposal is received and validated, the function `handle_quorum_proposal_validated` invokes `decide_from_proposal` or `decide_from_proposal_2`, depending on the code version.

This pseudo code function can be mapped to the `decide_from_proposal_2` implementation.

- `high_qc <- cert` maps to retrieving the leaf from the quorum proposal (code ref).
- `justify_high_qc <- fetchProposal(high_qc).justify_cert` corresponds to the parent retrieval (code ref).
- `justify2_high_qc <- fetchProposal(justify_high_qc).justify_cert` is equivalent to the grandparent retrieval (code ref).
- The last section corresponds to the check whether the parents parent is consecutive in view to the parent, and if so we can decided the grandparents view (code ref).

The `updateCerts(cert)` function is in pseudo code utilized also in the context of vote accumulation. When votes reach the required threshold in the `accumulate` function, the system broadcasts a `HotShotEvent::Qc2Formed` event (code ref). The dependency handler collects this event in the `handle_dep_result` function (code ref), which will result in the function `publish_proposal` being triggered (code ref). This sequence ultimately leads to the decision.

Note that in the specification, function `fetchProposal(high_qc)` is not defined. Additionally, the code does not implement `locked_qc`.

Function `validExtendedProposal(proposal)`

```
function validExtendedProposal(proposal):
    // The proposed block must be the same as in the justify QC
    // and in the current high_qc
    if proposal.data != proposal.justify_qc.data
    or proposal.data != high_qc:
        return false
    // The proposal, the justify qc and the high qc
    // need to me formed in three consecutive views
    if proposal.view - 1 != proposal.justify_qc.view
    or proposal.justify_qc.view - 1 != high_qc.view:
        return false
    // If this proposal is eventually accepted by this node
    // the justify qc will become a high qc and the high qc
    // will become a locked qc. The certificate formed based
    // on this proposal will become an eQC
    return true
```

The core logic of this pseudo code function is implemented as part of the function `validate_proposal_safety_and_liveness`:

- `if proposal.data != proposal.justify_qc.data or proposal.data != high_qc` (code ref).
- `if proposal.view - 1 != proposal.justify_qc.view or proposal.justify_qc.view - 1 != high_qc.view` (code ref) - checks that the proposed block and the justify QC block belong to the same epoch or that the justify QC is the eQC for the previous block.

Core implementation

```
as a leader:
    if received vote from node:
        if validVote(vote, node):
            accumulateSignatures(vote, node)
```

```
            if thresholdReached(vote):
                cert <- certificate(vote.type, vote.data, vote.view)
                updateCerts(cert)
                current_view <- cert.view + 1
                proposal <- formProposal(cert)
                broadcast_proposal(proposal)
```

The receipt of votes happens when functions handle_quorum_vote_recv or handle_timeout_vote_recv are called from the handle function (code ref). When the function handle_vote is called, it handles the creation of the vote collector and uses the appropriate handle_vote_event implementation based on the vote type being processed.

The vote validation has been discussed previously, as well as the functions accumulateSignatures (vote, node) and thresholdReached(vote).

The accumulate function handles the creation of the certificate when the threshold is reached (code ref).

The function updateCerts(cert) has been analyzed earlier in the document, as well as formProposal(cert) and broadcast_proposal(proposal).

The current view update happens when the event HotShotEvent::ViewChange is emitted. This event is broadcasted in the function handle_quorum_proposal_recv when the proposal's parent leaf cannot be found in storage (code ref) and at the end of the function, after validating proposal safety and liveness (code ref).

```
        if entered new_view because of timeout or view sync:
            current_view <- new_view
            cert <- high_qc
        proposal <- formProposal(cert)
        broadcast_proposal(proposal)
```

The condition if entered new_view because of timeout or view sync maps to multiple points throughout the code.

- In the file hotshot-task-impls/src/view_sync.rs, the HotShotEvent::ViewChange event is emitted in:
  - The handle function in case of HotShotEvent::Timeout - When the first timeout occurs (code ref).
  - The handle function in case of HotShotEvent::ViewSyncCommitCertificateRecv (code ref) and HotShotEvent::ViewSyncFinalizeCertificateRecv (code ref).

The proposal creation can be triggered by the HotShotEvent::ViewSyncFinalizeCertificateRecv and HotShotEvent::Qc2Formed(Right(certificate)) events (code ref). The timeout certificate is created when enough timeout votes are collected (code ref). The event is then handled in the handle function (code ref), after which the dependency task is created for view view_number = timeout_cert.view_number + 1;. The proposal will then be created, after the events SendPayloadCommitmentAndMetadata and VidDisperseSend have been received, with a timeout certificate as view change evidence (code ref). Similarly, for the ViewSyncFinalize certificate (code ref)(code ref).

The functions formProposal(cert) and broadcast_proposal(proposal) have been previously discussed.

```
as a node:
    if received proposal from node:
        if validProposal(proposal, node):
            updateCerts(proposal.justify_cert)
            current_view <- proposal.view
            vote <- vote(proposal.type, proposal.data, proposal.view)
            sendVote(vote)
```

The proposal receipt in the node starts from the handle function, case HotShotEvent::QuorumProposalRecv (code ref).

The functions validProposal(proposal, node), updateCerts(proposal.justify_cert), vote(proposal.type, proposal.data, proposal.view), and sendVote(vote) have been discussed previously in the document.

The section `current_view <- proposal.view` maps to proposal reception triggering the `ViewChange` event (code ref).

```
    if received vote from node:
        if validVote(vote, node):
            accumulateSignatures(vote, node)
            if thresholdReached(vote):
                cert <- certificate(vote.type, vote.data, vote.view)
                updateCerts(cert)
                current_view <- cert.view + 1
```

In the Rust implementation, vote accumulation is handled the same way for both nodes in epoch `e+1` and the leader in epoch `e`. A node will accumulate votes if either it's in epoch transition or it's the leader for the next view in the current epoch (code ref). This section has been analyzed previously.

Threat Model

Approach

The epoch change protocol is part of the overall HotShot consensus protocol as it was given in the Espresso paper. The paper gives a security sketch for safety and liveness, and therefore a high confidence that the protocol is correct from a theoretical point of view.

The focus is therefore to see if the protocol is correctly translated into code, and in particular if all corner cases - although solvable from a theoretical perspective - are correctly handled. The protocol specification describes the optimal procedure that can be interrupted by new view events. We therefore identified potential scenarios that can occur in this and used this to guide our inspection of code. While analyzing potential scenarios we observed that some are actually not possible in the implementation, while some scenarios have been merged after analyzing them.

In the end, we did not find any liveness violation (except the finding computing and comparing thresholds). We provide the detailed analysis results below.

For the threat model, we identify specific threats that could cause a violation of the invariants of the protocol (basically, the consensus properties). In addition, we add checks to ensure that the protocol is executed as specified.

In the following, let `v` be the view when the epoch change protocol from `e` to `e+1` is started.

1. Optimal run

   All view changes occur based on QC received

2. Timeout/Sync View change in `v`

   Normal recovery as in non-epoch change case

3. Timeout/Sync View change in `v+1`

   Normal recovery as in non-epoch change case

4. Timeout/Sync View change in `v+2`

   1. Scenario 1: Validators of `e` time out
   2. Scenario 2: Validators of `e+1` time out

5. Timeout/Sync View change in `v+3`

   1. Scenario 3

6. Epoch sync

Optimal run

Check #1

For all views of the last block `B_k` of `e`, i.e. `k mod K==0`, the leader sends the QC not only to the nodes in `e` but also to `e+1`.

Conclusion:

- The leader handles the votes in the function `handle_quorum_vote_recv`, where, based on the value of `transition_indicator`(code ref), votes are accumulated in order to create the certificate (code ref). If we are in the last block of an epoch, we collect votes for epochs `e` and `e+1` (code ref) and form appropriate certificates.
- The formed certificates are then sent to the nodes in epochs `e` and `e+1` (code ref).

Check #2

A node that is in `e+1` but not in `e` needs to participate in all views (`v`, `v+1`, `v+2`) of block `B_k`, `k mod K==0`, and validate the proposal of the leader of `v` in epoch `e` (not `e+1`!) and send it to the leader of `v+1` in `e`.

Conclusion:

- Once the event `HotShotEvent::QuorumProposalValidated` has been received while processing the last block of epoch `e`, nodes can vote for the proposal if they belong to the committee of either epoch `e` or `e+1`, where `e` is the epoch from which the block originates (code ref). Votes are then sent (code ref) to the leader of the next view from the epoch `e` (code ref).
- As noted in the conclusion of the previous point, votes from epochs `e` and `e+1` (code ref) are collected to create certificates when voting in the last block of an epoch. Consequently, the leader of epoch `e` accumulates votes from epoch `e+1` to form the next epoch's QC.
- When accumulating the votes, it is verified that the receiving node is either the leader or it's transitioning to the next epoch (code ref).

Check #3

The leader of `v+1` needs to form a quorum certificate of `e` and `e+1`.

- The majority of the stake is reached individually, both for `e` and `e+1`.
- The quorum certificate is calculated based on the staked values in `e` and `e+1`, respectively, and compared to the 2/3 threshold of the overall stake.
- For blocks `k mod K != 0`, only the threshold for `e` is needed.

Conclusion:

- When processing the final block of epoch `e`, nodes are able to vote on the proposal if they are either:
  - Part of the epoch `e` committee,
  - Part of the epoch `e+1` committee,

  where `e` is the epoch from which the block originates (code ref). The vote is then sent (code ref) to the leader of the view the proposal originated from (code ref).
- The votes are collected separately for epochs `e` and `e+1`, using `task_state.next_epoch_vote_collectors` (code ref) and `task_state.vote_collectors` (code ref).
- The vote senders' stake is accumulated (code ref) and compared to the calculated threshold (code ref). For both QCs, the threshold is calculated as `(2n/3) + 1`, where `n` represents the total number of nodes in the vote sender's epoch (code ref).
- The finding highlights the comparison between stake and number of committee members.

Check #4

Similar for `v+2`.

Conclusion:

- This conclusion follows the same reasoning as the previous point inspection.

Check #5

The quorum proposal for `v+1` includes block `B_k` (same as in the previous view). No new block is proposed.

Conclusion:

- The property holds. When publishing the proposal, the same block header from the parent leaf is reused if the parent QC is one of the three-chain certificates, excluding the eQC itself (code ref) and the condition `version >= V::Epochs::VERSION` is met (code ref). The block header is constructed based on `parent_leaf.block_header()` (code ref). The parent QC is set when the event `HotShotEvent::Qc2Formed`

is registered, which contains the QC in case the `SimpleCertificate` has been received (code ref). Otherwise, it can be obtained through the `wait_for_highest_qc()` function call (code ref).

Check #6

The extended quorum proposal for `v+2` includes block `B_k` (same as in `v` and `v+1`). No new block is proposed.

Conclusion:

- The quorum proposal for view `v+2` is handled the same way as in `v+1` and `v`. The same block header from the parent leaf is proposed, as described in the previous conclusion.

Check #7

The leader for `v`, `v+1` and `v+2` in epoch `e` is taken from the membership of epoch `e`.

Conclusion:

- The function `leader(view_number, epoch)` results in the leader of the committee for view `view_number` in `epoch` (code ref).
- Upon receiving a DRB result (code ref), it's stored and used to create a randomized committee (code ref). The DRB also generates a stake CDF (code ref). The `randomized_committee` then selects the leader in a manner that is both random and deterministic within all views in an epoch (code ref).
- When proposing, leadership check uses epoch derived from block header `block_header.block_number()` (code ref). Additionally, the utilized epoch value can be extracted from the votes epoch (e.g., code ref), task state current epoch (e.g., code ref), block number (e.g., code ref), and proposal data (code ref).

Check #8

The leader for `v+3` is from epoch `e+1`.

Conclusion:

- The leadership checks are performed as formerly described in the conclusion of the previous point.
- Epoch transitions happen through view changes, and the `HotShotEvent::ViewChange` event with a new epoch and increased view (code ref) will update the internal state of the task states, e.g. `NetworkEventTaskState` (code ref), `ViewSyncTaskState` (code ref), `ConsensusTaskState` (code ref), `QuorumProposalTaskState` (code ref), and `QuorumProposalRecvTaskState` (code ref) and set the value of the `self.cur_epoch` to the new `epoch`. The `QuorumVoteTaskState` does not keep track of the current epoch since it only needs the epoch information in the context of the specific data it's voting on. This ensures that when the internal state is used for leader checks, the task state variables hold the correct value.

Check #9

The last block of an epoch (`k mod K == 0`) is decided only if there is an extended quorum certificate for that block.

Conclusion:

- The decision process for the last block of an epoch follows the standard HotStuff decision rule implemented in `decide_from_proposal_2` (code ref). A block is decided when there exist three consecutive views with quorum certificates, regardless of whether it's the last block of an epoch or not or that they are proposing the same block.
- For the decision to be made and the `decide_from_proposal_2` function to be triggered (code ref), the `HotShotEvent::QuorumProposalValidated` event needs to be emitted in the `validate_proposal_safety_and_liveness` function (code ref). This function ensures that the epoch transition proposal includes the next epoch QC (code ref). During proposal creation, the `next_epoch_justify_qc` is set to the `next_epoch_high_qc` (code ref).
- On epoch change, when sending the high QC, a check verifies if the QC is forming an eQC. If it is, we retrieve the next epoch QC and emit the event `HotShotEvent::ExtendedQcSend`. This event is processed

upon registering `HotShotEvent::ExtendedQcRecv` (code ref), where we confirm that the QC representing the eQC matches the `next_epoch_high_qc` (code ref). We then update the `next_epoch_high_qc` and `high_qc` (code ref). In the function `handle_eqc_formed`, which is called when the extended QC is formed, a check ensures that the eqc corresponds to the `next_epoch_high_qc` (code ref).
- While we believe this implementation is safe, it differs notably from the specification (bullet point 7).

Check #10

The first block of an epoch (`k mod K == 1`) is decided only if there is an extended quorum certificate for the previous block.

Conclusion:

- As noted previously, the decision process requires three consecutive views to form QCs.
- When validating the first proposal of epoch `e+1` (code ref), since `proposal_epoch != justify_qc_epoch`, the epoch safety check in `validate_proposal_safety_and_liveness` (code ref) enforces that:
  - The parent block is from the immediately preceding epoch (code ref),
  - The parent block could form an extended QC (code ref) - checks if the QCs were formed in the three previous consecutive views for the same block number (code ref) and that the leaf is for the last block in the epoch (code ref).

Threat #1

The same view numbers are used for a new proposal in epoch `e+1` and for an epoch change proposed in epoch `e`.

→The proposal originates from the planned leader and is calculated from the correct epoch. The epoch is determined based on the block number, as deriving the proposer from local data may lead to discrepancies with the actual epoch of the proposal.

Conclusion:

- The proposal is published only in case the node is the leader of the view and epoch (code ref). The function `publish_proposal` ensures that the node is the leader for the epoch calculated based on block height (code ref) and the current view `self.view_number` from the `ProposalDependencyHandle`.
- When validating the received proposal (code ref), the `validate_proposal_view_and_certs` function ensures that the proposal was sent from the leader in the epoch calculated based on the block number (code ref). Additionally, the proposal is considered safe if the proposed block and the justify QC block belong to the same epoch, or for the nodes in epoch `e+1`, the justify QC is the eQC for the previous block (code ref).
- If the proposal passes the validation checks and fulfills the required dependencies, the event `HotShotEvent::QuorumProposalValidated` will initiate the `submit_vote` function (code ref). If it's the third proposal (core ref), the event can trigger the `handle_eqc_voting` function (code ref), which sends extended votes.
- During vote accumulation, we verify if we are the leader only in case the node is not in epoch transition, meaning the high QC is not for the last block in the epoch (code ref). The `leader` function check uses the votes view number, which is set when the vote is created in `create_signed_vote()` while submitting the vote (code ref), and `VoteCollectionTaskState` epoch value. During the epoch change, nodes in epoch `e` broadcast their votes for the extended certificate to nodes in `e+1` so that all nodes in epoch `e+1` can form the certificate.

→ Double voting on the same view number (one for the old epoch, one for the new).

- Is double voting for the same view possible?
- How is it treated at the sender side?
- How is it treated at the receiver side?

Conclusion:

- The vote submission process validates that `proposal.data.view_number() > self.latest_voted_view` to determine if a vote has already been cast for this view (code ref). The `self.latest_voted_view` is updated when the function `update_latest_voted_view` is called when the event `ViewChange` is processed.

- Double voting is possible during epoch transitions and when a node has stake in both the current epoch and the next epoch (code ref). The `leaf` in these checks originates from the proposal (code ref).
- On the receiver side, double voting for the same view is handled through `total_vote_map` (code ref). Furthermore, if a node is already in the signers list, the vote is rejected (code ref).

Timeout / Sync View change in `v` and `v+1`

Check #1

On a (first-time) timeout during view `v`, a node should advance to `v+1` and restart the 3-chain process.

Conclusion:

- A node that timeouts advances to the next view `v+1`. On the view sync task, the first time the task handles a timeout for view `v`, it will trigger a `HotShotEvent::ViewChange` event for `v+1` (code ref).
- In a node that timeouts, all tasks reset their internal state accordingly by updating their view to `v+1` and cancelling tasks for views `v` or older. This happens when handling events `Timeout` (for `v`) and `ViewChange` (for `v+1`), in particular in the following tasks:
  - network (code ref)
  - quorum proposal (code ref and ref)
  - quorum proposal recv (code ref)
  - quorum vote (code ref and ref)
  - consensus: it does not cancel any task for previous views.
  - view sync: it manages the timeout counter and will either trigger a `HotShotEvent::ViewChange` (on the first timeout) or `HotShotEvent::ViewSyncTrigger` (on the second).

Check #2

After a timeout, the first proposal of the 3-chain process must have highQC as certificate.

Conclusion:

- The highest QC seen is saved on each view (code ref).
- When creating a proposal, pick as proposal certificate (`parent_qc`) either the highest QC seen or the certificate of the latest committed block (code ref).
- On each view change, the highest QC is sent to the leader of the next view or to all nodes in epochs `e` and `e+1` if it's an eQC (code ref).

Check #3

On a (first) timeout during view `v+1`, the node should advance to the next view `v+2` and restart the 3-chain process.

Conclusion:

- This is analogous to 1, but with view `v+1` instead of `v`. In this case, the certificate for the `v+2` proposal comes from the votes in `v+1`, and not from highQC.

Check #4

On a second timeout during view `v`, start the view sync protocol to transition to `v+1`.

Conclusion:

- The view sync protocol is triggered when there are two consecutive timeouts for a view change. On a view change, the node will send highQC to the next leader. If the next leader does not reply, the view change handler will timeout for the first time (code ref). The view sync handler will try again to do a view change (code ref). If this times out again, it will emit a `HotShotEvent::ViewSyncTrigger` event that will start the view sync protocol (code ref).

Timeout / Sync View change in v+2

Scenario #1

We were investigating this scenario candidate, as from the specification and paper, it appeared to describe legitimate behavior of the protocol. However, after our analysis and discussions with the Espresso team we concluded that this actually cannot happen. As a result we found no liveness violation in this case. We kept it here to document the analysis and checks we did.

- Nodes 1, 2, 3, 4 are in `e`; nodes 1, 5, 6, 7 are in `e+1`. Note that node 1 is in both epochs.
- View `v`, leader 1
  - send proposal `P(B_k, C(B_k'))` to 1..7
- View `v+1`, leader 2
  - 1..7 sign and send vote `V_v(B_k)` to 2 and store `C(B_k')` as highQC
  - 2 forms QC (first certificate) and sends `P_{v+1}(B_k, C_v(B_k))` to 1..7
- View `v+2`, leader 3
  - 1..7 sign and send vote `V_{v+1}(B_k)` to 3 and store `C_v(B_k)` as highQC
  - 3 forms QC (second certificate) and sends `Pext_{v+2}(B_k, C_{v+1}(B_k))` to 1..7
  - now, either network is slow or 3 faulty: only 5, 6, 7 receive `Pext`
    - They store `C_{v+1}(B_k)` as highQC and send `V_{v+2}(B_k)` to 1, 5, 6, 7
  - 1, 2, 4 time out and progress to `v+3`, 3 follows
- View `v+3`, leader 4 (for epoch `e`), leader 5 (for epoch `e+1`)
  - all nodes continue in `v+1` to build `B_k` and try to transition to `e+1` as no double certificate can be formed without the votes from 1,2,4.

This case thus falls back to the analysis as it was done for a timeout / view change in v or v+1.

Scenario #2

- Similar to Scenario 1, but in view `v+2` only the old validator set 1..4 receives the extended proposal message `Pext` (from leader 3) before a view change. Nodes 5, 6, and 7 do not receive the extended proposal and timeout.
- Nodes 1..4 broadcast `Vext` to 1..7 and proceed to `v+3`.
- Nodes 5..7 timeout and initiate view change to `v+3` for epoch `e` (with leader 4).
- No process is able to build the double certificate for `e` and `e+1`.
- All processes time out in `v+3` and proceed to `v+4` in `e` (with leader 1).
  - 1..4 will time out waiting for `Vext` votes from 5..7 to build the double certificate.
  - 5..7 will time out waiting for 4, the leader of `v+3`.
- The leader of `v+4` in epoch `e` (node 1) restarts the 3-chain process.

Check #1:

- An eQC is considered valid only if there is a certificate with the nodes of `e` and a certificate with the nodes from `e+1` (double certificate).

Conclusion:

- Every time a new view starts, the node calls `send_high_qc`, which immediately sends the latest certificate to the next leader (this is part of HotStuff 2) (code ref).
- When it's the time for an extended certificate, as is the case for nodes 1..4 in `v+3`, it will call first `wait_for_next_epoch_qc`, which will block until receiving a quorum certificate from nodes in `e+1`. Once received, it emits `HotShotEvent::ExtendedQcSend`, which will broadcast eQC instead to all nodes (code ref). An eQC is composed of two certificates, one for `e` and one for `e+1` (code ref).

Check #2:

- If a timeout / view change happens in `v+3`, the protocol continues in epoch `e` with `v+4` and the leader of `v+4` in `e`.

Conclusion:

- In case of a first-time timeout in a view, the node will attempt to advance to the next view in the same epoch (code ref).

Check #3:

- Nodes that are not validators in epoch `e` do not trigger a view change in validators that are in `e`.

Conclusion:

- In this case, nodes initiate a view change when receiving a valid proposal (code ref). Since nodes in `e+1` but not in `e` do not receive the proposal, they stay in the same view until timeout. On timeout, they will try to transition to the next view `v+3` (code ref).

Timeout / Sync View change in `v+3`

Scenario #3

- In `v+2`, all nodes in `e` and `e+1` have signed the proposal for eQC and sent their votes to all nodes.
- All nodes receive all votes, form eQC, and decide `B_k`. At this point all nodes have committed the block.
  - Note that this differs from the spec (point 7), which only mentions the nodes in `e+1` collecting votes and forming eQC to decide the block.
- Nodes that form eQC broadcast it to all nodes in `e` and `e+1`.
- Upon forming or receiving eQC, nodes immediately start view `v+3` in epoch `e+1`.
- Node 5 (`v+3`'s leader) proposes block `B_{k+1}` and sends `P_{v+3}^{quorum}(B_{k+1}, eQC)` only to nodes in `e+1`.
- Nodes in `e+1` continue with the regular execution of the HotShot protocol in `v+3`. At this point, if nodes time out or start the view sync protocol, this does not interfere with the epoch sync protocol.

Checks:

- A node that forms eQC broadcasts it to all nodes in `e` and `e+1`.
  - Upon forming eQC, the node advances to the next view (code ref).
  - Upon starting a view, the function `send_high_qc` sends the latest QC (code ref).
  - Since the latest QC is an eQC, it is broadcasted to all nodes (code ref).
- Block `B_k` is decided at every node in `e` that receives an eQC for that block.
  - On a `HotShotEvent::QuorumProposalValidated` the function `handle_quorum_proposal_validated` decides the block and view (code ref).
  - The decision process for the last block of an epoch follows the standard HotStuff decision rule implemented in `decide_from_proposal_2` (code ref).
- A node that has committed the last block `B_k` in `v+2`, transitions to the next view and epoch.
  - A node advances to the next epoch when it forms eQC for (code ref) or receives eQC from a peer (code ref).
  - Note that nodes in `e` but not `e+1` may continue participating passively in `e+1`.
- The leader of `v+3` in epoch `e+1` sends a new proposal only to nodes in `e+1` with eQC as justification.
  - A valid eQC is saved as `high_qc` and `next_epoch_high_qc` (code ref and ref).
  - The proposal `B_{k+1}` has eQC as justification, which is composed of `high_qc` (code ref) and `next_epoch_qc` (code ref).
  - This is the first block for `e+1`. Proposing and voting proceeds according to the regular HotStuff protocol without epochs involved.

Epoch sync

Check: If a node proceeds to `e+1` every other node will receive the eQC for the last block of `e`.

Threat:

- Nodes in `e+1` don't broadcast the epoch change data.
- Nodes in `e` receive it and don't act on it to jump to finalise epoch `e`.

Conclusion:

The epoch change data is broadcasted on two instances:

- `HotShotEvent::ExtendedQc2Formed` - When the event is received, the next epoch number is calculated based on block height, and the `HotShotEvent::ViewChange` is broadcasted with the new epoch number and increased view number (code ref).
  - This event is emitted when a node in epoch `e+1` collects enough votes to form a certificate (code ref). These votes are emitted from the `handle_eqc_voting` function (code ref), and they are broadcasted to all the nodes (code ref), which can then emit the `ExtendedQc2Formed` after collecting enough votes.
- `HotShotEvent::ExtendedQcRecv` - When the event is received, the extended leaf and both QCs are validated, and the consensus state with both QCs is updated. If either QC is updated, the event `HotShotEvent::ViewChange` is broadcasted with the new epoch number and increased view number (code ref).
  - During view changes, `send_high_qc` checks if there's an extended QC and broadcasts it as a `HotShotEvent::ExtendedQcSend` event (code ref). When nodes receive this broadcast, they handle it when the `HotShotEvent::ExtendedQcRecv` event is received.
  - In `network.rs`, when handling `ExtendedQcSend`, it's converted to a broadcast message(code ref).

Nodes in `e` that are not in `e+1`, when they receive the epoch change data, they advance to `v+3` in `e+1`.

- Upon receiving a proposal (code ref), the node checks that it belongs to the epoch of the proposal with the function `validate_epoch` (code ref). Then, these nodes participate in `e+1` passively, without proposing or voting.

Local concurrency considerations

In the implementation, task handlers are not atomic, with many fine-grained locks accessing the task's state and the storage. For example, handling the `ExtendedQcRecv` event in consensus (code ref) reads two times the consensus state and then writes to it, each action guarded by separate locks. Or the `NextEpochQc2Formed` event in quorum_proposal (code ref), which performs a state read call, a state write call, and a storage write call. If not handled properly, all tasks running in parallel could lead to unexpected interleaved executions that are hard to analyse and debug.

We have not found any scenario that could result in race conditions, in particular when looking at the epoch-related code. It is however recommended to start with handlers that have less locks or more coarse-grained locks, which is easier to analyse, and then introduce more fine-grained access control to shared data for improving the performance.

Findings

Finding	Type	Severity	Status
Threshold calculation inconsistency between stake and node count	Implementation	Critical	Resolved
Unreachable epoch handling code	Implementation	Informational	Acknowledged
Miscellaneous code findings	Implementation	Informational	Acknowledged
Specification is lacking details and is inconsistent with implementation	Documentation	Informational	Acknowledged

Threshold calculation inconsistency between stake and node count

ID	IF-FINDING-001
Severity	Critical
Impact	3 - High
Exploitability	3 - High
Type	Implementation
Status	Resolved

Involved artifacts

- types/src/v0/impls/stake_table.rs
- hotshot-types/src/vote.rs

Description

When collecting votes from the nodes, the leader accumulates the voters' stakes (code ref). The total stake is then compared to the threshold value (code ref), and if the condition `total_stake_casted >= threshold` is satisfied, the signed certificate is formed (code ref).

The threshold value is obtained by calling the `threshold(&*membership_reader, epoch)` function (code ref), which calculates the success threshold based on the number of nodes in the committee (code ref). The function `success_threshold` determines the value of `quorum_len` based on the number of elements in the stake table for the given epoch and calculates the threshold as `((quorum_len * 2) / 3) + 1` (code ref).

This creates a discrepancy where votes are weighted by stake during collection, but the threshold they're compared against is based on node count.

Problem scenarios

Assume there are `n` nodes in the committee for epoch `e`, each with a stake of `*Sn*`. For the certificate to be valid when calculating `success_threshold`, the following condition must be satisfied:

`Sum(Sn) >= ((n * 2) / 3) + 1`.

This indicates that a certificate may not represent a true quorum if the node stakes differ from 1. It could be formed by only a small number of high-stake nodes or a large number of minimal-stake nodes relatively to the `n`.

This discrepancy between vote collection and threshold calculation could significantly weaken the system's security guarantees.

Recommendation

The current threshold calculation assumes each node has equal weight but vote accumulation weights by stake. We suggest implementing the threshold functions to reflect the stake of participating nodes, making both vote collection and threshold calculation stake-based.

Status

Resolved

Unreachable epoch handling code

ID	IF-FINDING-002
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- hotshot-task-impls/src/quorum_vote/handlers.rs
- hotshot-task-impls/src/helpers.rs

Description

In the `handle_quorum_proposal_validated` function, the `LeafChainTraversalOutcome` is determined by calling either `decide_from_proposal_2` or `decide_from_proposal`, based on the code version (code ref). If `version >= V::Epochs::VERSION`, the function `decide_from_proposal_2` is invoked with `with_epochs` set to true. If `version < V::Epochs::VERSION`, `decide_from_proposal` is called with `with_epochs` set to false.

Within the `decide_from_proposal` function (code ref), the following if statement exists:

```
if with_epochs && res.new_decided_view_number.is_some() {
  if let Some(decided_leaf_info) = res.leaf_views.last() {
    decide_epoch_root(&decided_leaf_info.leaf, epoch_height, membership).await;
  } else {
    tracing::info!("No decided leaf while a view has been decided.");
  }
}
```

Here, the value of the `with_epochs` flag is always set to false since the function `decide_from_proposal` is only called from the function `handle_quorum_proposal_validated`.

Problem scenarios

The epoch handling code in the `decide_from_proposal` function will never execute because `with_epochs` will always be false when that function is called.

Recommendation

We recommend removing the dead code from the `decide_from_proposal` function since it can never be reached.

Miscellaneous code findings

ID	IF-FINDING-003
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

In this finding, we describe a number of improvements to the code. Those typically do not affect the functionality, but improve the code readability, make code more robust with respect to future changes, or represent a good engineering practice.

1. In the current implementation, the function `wait_for_highest_qc()` is only called when the version is greater than or equal to `V::Epochs::VERSION` (code ref). Given this, the check for `V::Epochs::VERSION` inside `wait_for_highest_qc()` appears redundant (code ref).

Specification is lacking details and is inconsistent with implementation

ID	IF-FINDING-004
Severity	Informational
Impact	1 - Low
Exploitability	0 - None
Type	Documentation
Status	Acknowledged

Involved artifacts

- Espresso paper
- Protocol specification document (pdf from Notion site)
- Naor-Keidar paper
- Implementation

Description

We identified that there are some differences between the Espresso paper, the specification and the implementation. While none of these differences seems to lead to a concrete problems in the code (unless mentioned in another finding), these discrepancies are technical debt: currently, understanding the core of the implemented protocol requires reverse-engineering of the code, which is hard and error prone for distributed systems, in particular for partially synchronous BFT systems. As a result, this reverse-engineering step will slow down or jeopardize future onboarding, future changes in the code, and future audits.

Discrepancies between the spec and the implementation

- While the Espresso paper and the code follow an event driven architecture, the specification was given in procedural form. We did a mapping between the functions in the specification and the code, but this was much more complex than necessary.
- In the specification (point 6 and 7) only the nodes in `e+1` build the eQC and decide block `B_k`. In the implementation, this is done by all processes, in `e` and `e+1`. Strictly following the problem definition, only the processes in `e` should decide `B_k`.
- According to the specification (point 9), an epoch `e` is finished for a node when it sees blocks committed in `e+1`. In the implementation, nodes in `e` commit the last block of `e` and advance to `e+1` where they participate in consensus passively, without proposing or voting but committing new blocks.

Inconsistencies in the pseudocode

- There's a `decided_qc` variable that is assigned a value but then never used.
- In a `proposal`, fields `justify_cert` and `justify_qc` seem to be referring to the same value.
- Variable `highQC` stores the highest certificate a node forms or receives. This variable is used to check the validity of extended certificates (a check on 3 consecutive views with `proposal.view - 1 = proposal.cert.view` and `proposal.cert.view - 1 = state.highQC`). The problem here is that upon reaching the threshold for votes for the quorum proposal that precedes the extended proposal, the leader will form a quorum certificate and store it as `highQC` (function `updateCerts` is called in the main loop). This will cause the leader to reject it's own proposal because the 3 consecutive views rule does not hold anymore (given that `highQC` was updated when forming the QC). As a consequence, a leader will not validate it's own extended proposal and this creates a deadlock. This problem was not observed in the code.
- Step 6 of the text explicitly requires nodes to send `V_{v+2}^extended` votes to all nodes in epoch `e+1`. However, the pseudocode implementation only reflects votes being transmitted to the leader, as it can be seen in the `sendVote(vote)` function.

Inconsistencies between the Espresso paper and the implementation

- Section 2.4.3 of the paper says that "the nodes in epoch e [...] drive view synchronization in epoch e + 1. We achieve this by running a variant of the Naor-Keidar protocol ...". And Figure 4, "ViewSync" section, says

> "For the special case where view v-1 is the last view in epoch e-1, the honest nodes in epoch e-1 will trigger the variant of the Naor-Keidar protocol and send the eQC to the nodes in epoch e". This could imply that the view sync protocol is triggered on each epoch transition. In the code, it's only triggered on two consecutive timeouts of a given view and it's part of the "normal" protocol.

Lack of details in documentation

The combination of the actual consensus protocol, the timeout mechanism, the view sync protocol and the epoch view sync protocol is not explicitly specified. While each of the protocols is described separately in academic papers, the implementation seems to integrate them much more into one mechanism that is hard to analyse only by looking at the implementation. The given specification only comprises the consensus part, but not the view change protocol and how this is integrated with the "normal" protocol messages.

Recommendation

A precise and complete documentation of all parts of the protocol, their expected behaviors and interactions helps onboarding on the code, facilitates future changes in the code and will expedite future audits.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for $< 3\%$; *medium* for 3-10%; *large* for 10-33%, *all* for $>33\%$. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

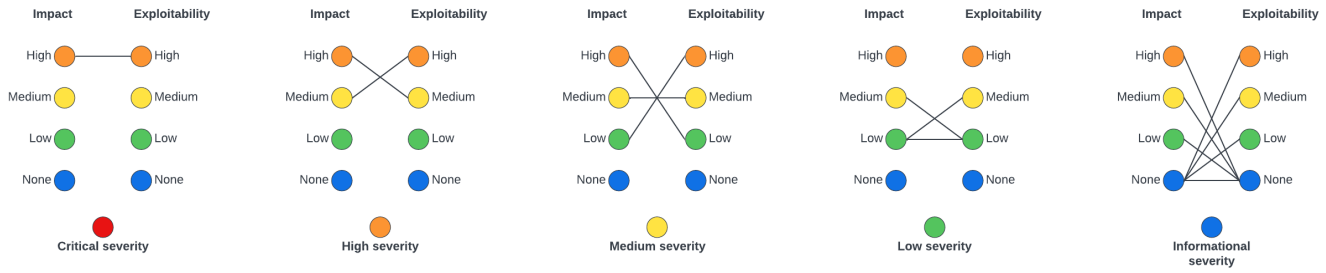

Figure 2: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.