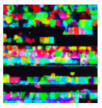


ZKSECURITY

Audit of Matter Labs' Era Consensus

April 22, 2024

Introduction

On April 22, 2024, Matter Labs tasked zkSecurity with auditing its Era Consensus implementation. The specific code to review was shared via GitHub as a public repository (https://github.com/matter-labs/era-consensus at commit `e008dc2717b5d7a6a2d8fb95fa709ec9dd52887f`). Matter Labs emphasized that this was a preliminary review, with another one planned in the future, when the code is more stable. The audit lasted 3 weeks with 2 consultants.

Internal design documentation was also provided by Matter Labs, including a network design document, a specification on their consensus protocol (called Two-step Fast-HotStuff), a document on security considerations, a security review on their cryptographic primitives, a roadmap to gradually replace the current protocol with their decentralized sequencer, a document on handling hard forks, and a rationale document on the reasons behind decentralizing the sequencer.

The documentation provided, and the code that zkSecurity looked at, was found to be of great quality. In addition, the Matter Labs team was responsive and helpful in answering questions and providing additional context.

Scope

The work centered around the consensus implementation, which comprised of:

- An Actor model with three actors (called bft, network, and sync block, although the last one was later merged in the network actor) and a supervisor (called executor).
- A variety of libraries to support the topology of the network, including an instantiation of the Noise protocol framework to secure connections between nodes.
- Cryptographic wrappers, as well as an implementation of the BLS signature scheme using the BN254 curve.

Overview of Era Consensus

The zkSync Era consensus protocol is to be seen in the context of the zkSync Era rollup. Currently, the zk rollup has a single sequencer, which processes transactions and orders them before their execution can be proven (and before the proof can be submitted on-chain). The goal of the consensus protocol is to decentralize the liveness of the zkSync protocol, by having the on-chain smart contract be a node processing the output of the consensus.

The consensus protocol we reviewed is a mix of **FaB Paxos** and **Fast-HotStuff**, which assumes $\mathrm{n} = 5\mathrm{f} + 1$ participants including f byzantine nodes. This is a larger threshold, and as such tolerates less faulty nodes, than some of the other consensus protocols which usually rely on $3\mathrm{f} + 1$ participants (and so can tolerate a third of the nodes being faulty concurrently). The reason is that this threshold allows for a smaller number of rounds to commit. This insight also appeared in **Good-case Latency of Byzantine Broadcast: A Complete Categorization** which proved that a closely-related bound ($5\mathrm{f} - 1$) is the minimum threshold (of byzantine node to number of nodes) to achieve such a short number of round trips before being able to commit.

The protocol works in successive *views*, in which an elected leader (via a round-robin election) proposes a single block, and other participants (called *replicas*) attempt to commit to it.

As different replicas see different things, the subtlety of the protocol is to ensure that the protocol never leads to *forks*, meaning that different replicas commit to conflicting blocks. In the current implementation, a fork would translate to different honest replicas commiting to different blocks sharing the same *block number*.

To understand the protocol, one must understand that there are three important cases that need to be taken into account:

1. A proposal is committed and the protocol moves on.
2. The protocol *times out* but the proposal is committed by some.
3. The protocol *times out* and the proposal is not committed.

The protocol aims at being able to go "full-throttle" when the first case happens (which is expected to be true for long periods of "normal" operations), while making sure that if case 2 happens, then the next leader is **forced** to repropose to ensure that every honest node eventually commits the proposal.

Sometimes, as you will see, case 3 leads to a missed proposal still being reproposed, because we are not sure if we are in case 2 or 3. But before we can explain how the safety of the protocol is guarded, let's see how things work when we're in the first case.

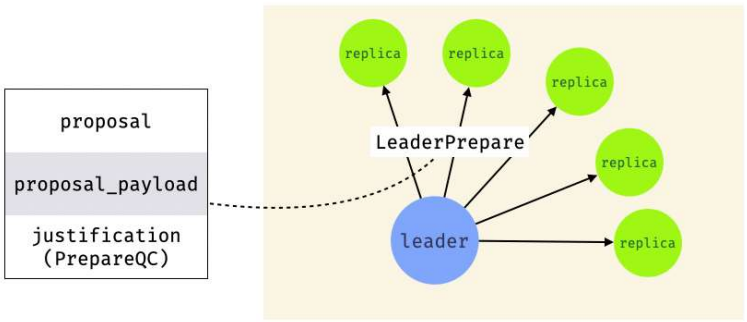
The "Happy Path" of The Protocol

There are four steps to a view, which are split equally into two "phases": a *prepare* phase, and a *commit* phase. Messages from the leader of a view always start with `Leader_` and end with the name of the

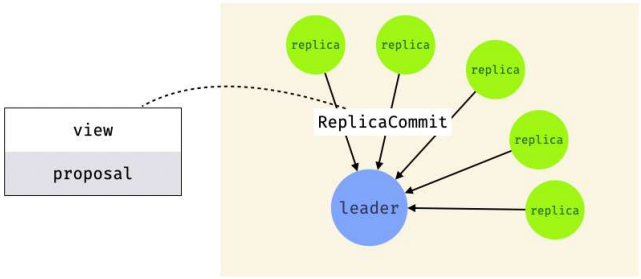
phase, same for the replicas.

A new view can start (in the prepare phase) for two reasons: either because things went well, or because replicas timed out. We will discuss these in detail later.

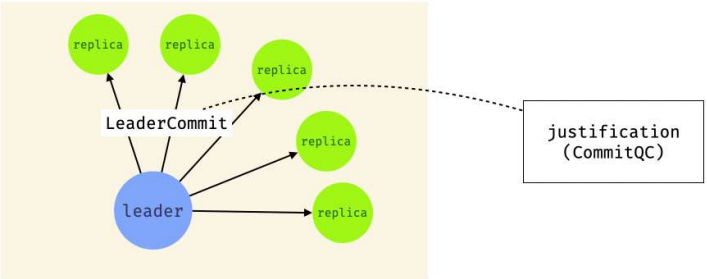
During the prepare phase, a leader proposes a block using a `LeaderPrepare` message, which it broadcasts to all replicas (including itself). In the current implementation, a block can be uniquely identified using its block number and hash. In addition, the `LeaderPrepare` message must carry a *justification* for why the leader is allowed to enter a new view and propose (more on that later).



After receiving a `LeaderPrepare` message, each replica will verify the proposal as well as the justification (more on that later), and vote on it if they deem it valid. Voting on it happens by sending a `ReplicaCommit` message back to the leader, which contains a signature of the view and the proposal.



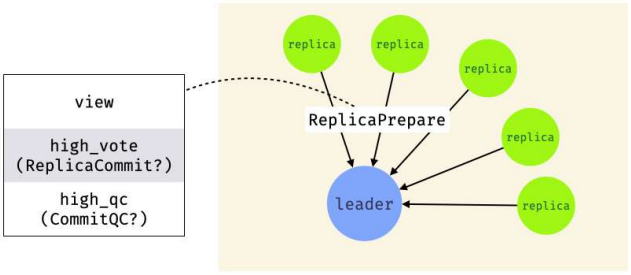
If the leader can collect a "threshold" of (specifically, $4f + 1$) such messages, it creates a certificate called a `CommitQC` that proves that a block can be committed, and broadcasts it to all replicas (within a `LeaderCommit` message).



Importantly, if at this point a leader can create a `CommitQC`, then we know that the block can be committed *even if the leader fails to broadcast it in the next step*. We will explain later why.

When replicas receive this message, they are ready to enter the next view (which they can do by incrementing it like a counter). When entering a new view, they nudge the newly-elected leader to do so as well by sending them a `ReplicaPrepare` message.

A `ReplicaPrepare` message is sort of an informational announcement about a replica's state, which carries their highest vote (for the highest view) as well as the highest committed block they've seen (through a `CommitQC`).



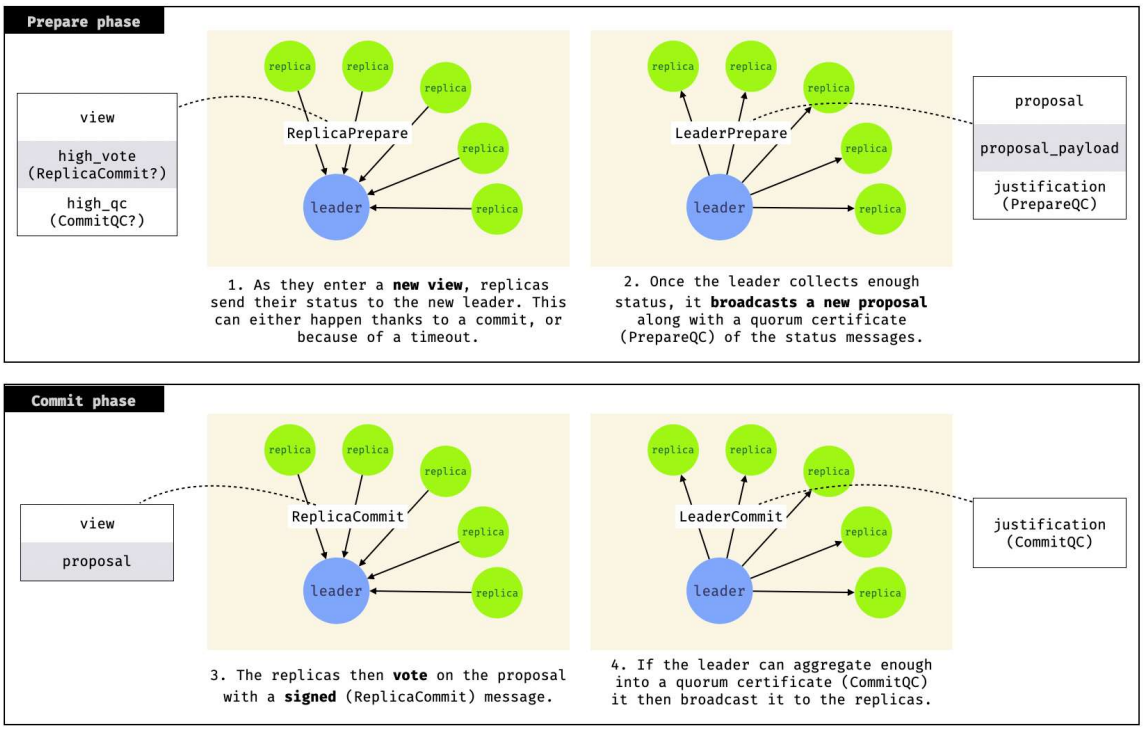
The new view's leader, who sees such a `CommitQC` in one of the replicas' message can use that as *justification* to enter the new view and propose a new block.

Note that while in the "happy path", it is enough to justify a proposal with the previous view's `CommitQC`, the implementation currently has leaders include a threshold ($4f + 1$ specifically) of such `ReplicaPrepare` messages.

As such, a replica can commit to a block in any of the two types of leader messages they can receive:

- when they observe a `CommitQC` in a `LeaderCommit` message, at the end of a view, or
- if they fail to observe such a view ending (due to network issue, for example), they will see it in the next `LeaderPrepare` message which will carry them in the `ReplicaPrepare` messages it carries

We summarize the full protocol in the following diagram:



Next, let's talk about what happens when things don't go so well...

The "Unhappy Path": Timeouts and Reproposals

The previous explanation emphasized the "happy path", which is the easiest way to understand the protocol.

What if a network issue, or slow participants (including a slow leader), or worse malicious participants (including a malicious leader), prevent the protocol from moving forward at one of the previous steps?

In this case, replicas are expected to wait a certain amount of time until they *time out*. The time they will wait depends on how long it's been since they last saw a commit. When a time out happens, replicas preemptively enter the next view, sending the same `ReplicaPrepare` message as we've seen previously.

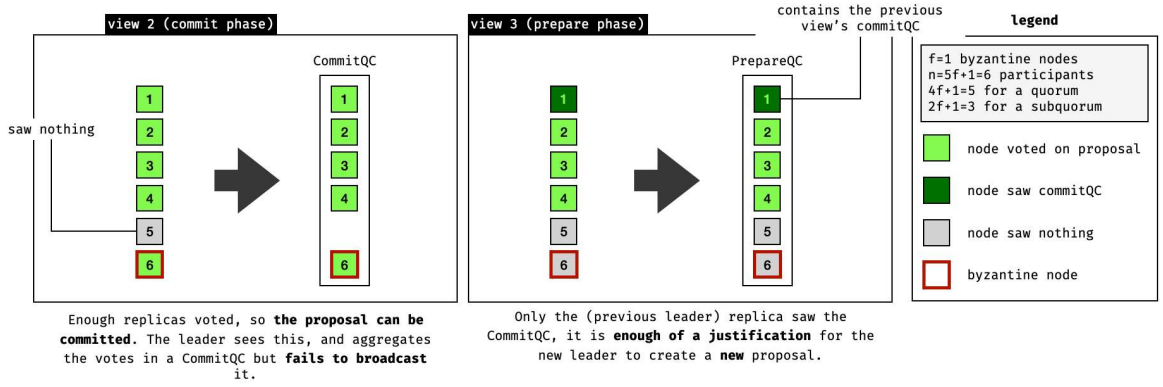
The next leader will use a threshold of them (which some protocols call that a "timeout certificate") as *justification* to enter the new view. This justification will show that enough replicas were ready to enter the new view.

Importantly at this point, we must be able to detect if it is possible that something was committed by an honest node. One example of such a scenario is the leader being honest and committing but being unable to broadcast the `CommitQC` due to network issues. Another example could be a malicious leader that would only send the `CommitQC` to a single node (who would then not be able to share it with the rest of the participants).

As the justification carried by the `LeaderPrepare` message also include the highest vote and highest commit of $4f + 1$ replicas, we can distinguish two scenarios:

1. a commit was seen by some people
2. no commit was seen by most people

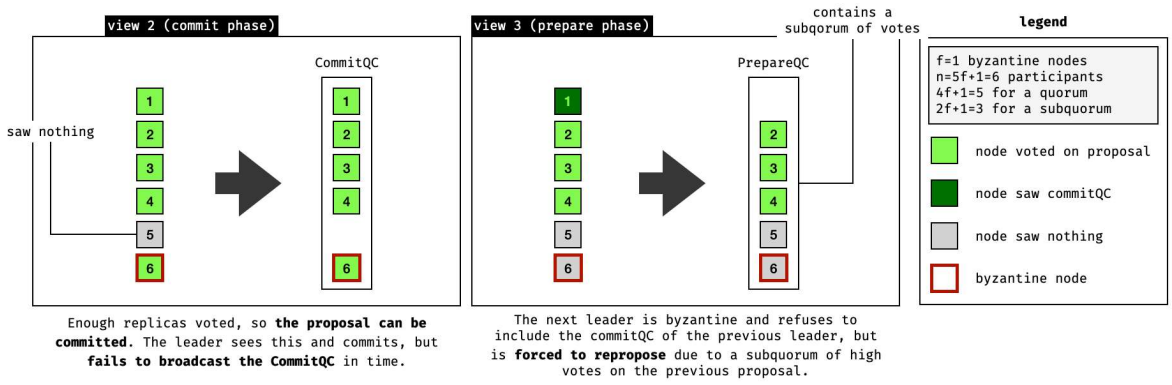
The first case is easy to handle, as anyone can see the previous `CommitQC` in some of the replicas' status, and as we said previously a `CommitQC` is enough of a justification for the leader to propose something new. We illustrate such a scenario in the following diagram:



The latter case will also be obvious, as the `high_vote` of replicas won't match their `high_commit`. But still... it doesn't give us the full picture. Since we only see a threshold of the status, it is not immediately obvious if *some* of the silent honest replicas have committed to the previous proposal.

Either the proposal failed to commit, or some nodes committed to it but failed to broadcast the `CommitQC`. If some honest nodes committed to it, we need to honor their decision and ensure that we do not throw away the previous proposal. If we were to do that, we would allow other nodes to commit to something at the same block number (which would be a fork, and thus a breach of the *safety property* of the consensus protocol).

The solution is in this simple rule: if the leader proposal's justification carries a *subqorum* of $2f + 1$ high votes that are for the previous proposal, then we force the leader to repropose the previous proposal. We illustrate such a scenario in the following diagram:



Why does this work? Let's look at the following safety proof. We need to prove that if someone has committed a proposal A, then either a new view's leader will extend the proposal A (by including a

`CommitQC` for proposal A), or they will repropose the proposal A.

1. Let's say that someone commits a proposal A.
2. This can only happen because they saw a `CommitQC` on proposal A.
3. Which can only happen because there were $4\mathrm{f}+1$ votes for the proposal A.
4. Out of these $4\mathrm{f}+1$ votes, at least $3\mathrm{f}+1$ votes are from honest nodes.
5. In the next view, the proposer will carry a `PrepareQC` showcasing $4\mathrm{f}+1$ new view messages from replicas, which will showcase the replicas highest votes.
6. These $4\mathrm{f}+1$ highest votes will be taken as a combination of $\mathrm{n}=5\mathrm{f}+1$ highest votes.
7. From the $3\mathrm{f}+1$ honest nodes, **AT MOST** f might not be included in the $4\mathrm{f}+1$ highest votes.
8. This means that **AT LEAST** $2\mathrm{f}+1$ of these votes on proposal A will be included in the new leader's proposal.
9. Since there were $2\mathrm{f}+1$, the new leader **MUST** repropose it, and all honest nodes will verify that this was done.

As such, we can say with strong confidence that a block gets committed as soon as $3\mathrm{f}+1$ *honest* nodes vote for it.

Catching up

In case a node is lagging behind, there needs to be a way for them to catch up.

There's a few situations where a node realizes that they are lagging behind, which arises when receiving any of the four different types of messages. As such, a node should always be ready to receive messages that are in future views.

A node should also make sure that they are not being fooled (otherwise a byzantine node could force other replicas to randomly catch up for no good reason). This can be done by always validating signatures and messages before acting on them, and only catching up when receiving a valid `CommitQC` (which can be received in any of the messages except a `ReplicaCommit` message).

In addition, one should avoid to slow down the protocol when possible. For example, a leader might receive enough information to understand that they have to repropose the previous proposal without knowing what it is (as enough replicas are ready to revote on it). Another example is if a replica sees a `commitQC` for a view in the future, it should store it in its state as the highest commit seen without having to fetch all the now-committed blocks.

Catching up means that a replica will fetch all the blocks which have a block number between the highest one we have in storage and the one we just saw a `CommitQC` for. Each block in between will come accompanied with their own `CommitQC`, proving that they were committed at some point in time.

Weighted BFT

Such BFT protocols are often seen implemented (in cryptocurrencies) with a proof of stake protocol in order to define who the replicas are. In such a setting, nodes become replicas by staking some amount

of tokens, and their votes are weighted based on the amount of tokens they have staked compared to the rest of the replicas.

A weighted vote essentially means that a replica can play the role of multiple replicas in the protocol. More subtly, what it also means is that the thresholds that are computed are not necessarily as exact as with the discrete version of a protocol. For example, while the discrete protocol might consider $2\mathrm{f} + 1$ messages to represent a certificate, a weighted protocol might not be able to have exactly that number of messages (as the sum of the votes might either carry less or more, but not exactly, the threshold we have set).

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	crypto/bn254	Lack of Proof of Possession for BLS Signatures	Medium
01	*	No Reconfiguration Protocol	Medium
03	validator/leader_prepare	Subquorum Choice Can Affect Liveness	Medium
04	libs/concurrency	Cloneable RNG Could Lead To Predictable Randomness	Low
05	network/consensus	Consensus Handshake Could Facilitate DoS Attacks	Low
06	crypto/bn254	Non-Standard BLS Implementation	Informational
07	*	Enforce Validity of Data Statically	Informational
08	bft/leader/replica_commit	Denial of Service Due to Late Validation	Known Issue

00 - Lack of Proof of Possession for BLS Signatures

crypto/bn254

Medium

Description. The signature scheme used, BLS signatures, is well-known to be vulnerable to "rogue key attacks" if not used correctly. A rogue key attack allows a malicious actor to forge an aggregated signature that looks like other (victim) signers were involved when they were not.

The first line of defense against such attacks is to ensure that each participant truly knows their keypairs. As discussed in "The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks", not any "public key validation" scheme is secure, and either a knowledge of the secret key (KOSK) or a Proof of Possession (PoP) scheme (with a separate hash function) must be used.

Currently, it is not clear if the public keys of the validators are vetted in some way that would prevent such rogue key attacks. As such, it is possible that two malicious replicas could collude to forge arbitrary quorum certificates in the consensus protocol. The reason for the collusion is that one malicious replica would need to create a malicious keypair that would not allow them to sign on their own (and thus one of the malicious replicas would not be able to participate directly in the consensus protocol).

To understand the attack, let's spend some time understanding how BLS works.

BLS works by checking that a signature S is equal to $\mathrm{r} \cdot \mathrm{x}$, where r is an "unknown and hidden value" derived from the message and x is the private key of the signer.

The check works because of the combination of:

- r being unknown and hidden: given a message and a base point P anybody can compute $\mathrm{Q} = \mathrm{r} \cdot \mathrm{P}$, but not r itself
- only the signer can produce $(\mathrm{r} \cdot \mathrm{x}) \cdot \mathrm{P}$: they can do so by scaling Q with their secret key

Anyone can then verify the equality without knowing the values themselves by using a pairing over public values (the hidden value $\mathrm{Q} = [\mathrm{r}]$, the public key $\mathrm{X} = [\mathrm{x}]$, and the signature S):

$$\mathrm{e}([\mathrm{r}], [\mathrm{x}]) = \mathrm{e}(\mathrm{S}, [1])$$

An aggregated signature over the same message is simply the addition of multiple signatures $\sum_\mathrm{i} \mathrm{S_i}$, which can be checked with a similarly aggregated public key $\sum_\mathrm{i} \mathrm{X_i} = \sum_\mathrm{i} [\mathrm{x_i}]$. This follows from the generalization of the previous pairing equation:

$$\mathrm{e}([\mathrm{r}], \sum_\mathrm{i} [\mathrm{x_i}]) = \mathrm{e}(\sum_\mathrm{i} \mathrm{S_i}, [1])$$

The attack works by creating a keypair $(\tilde{x}, \tilde{X})$ such that its public key cancels a set S of "victim" public keys involved in the aggregated signature:

$$\tilde{X} = (\tilde{x} \cdot P) - (\sum_{i \in S} X_i)$$

and to simply sign as if we were alone:

$$\tilde{S} = \tilde{x} \cdot Q$$

so that the pairing check passes:

$$\begin{aligned} & e([r], \tilde{X} + \sum_{i \in S} X_i) = e(\tilde{S}, [1]) \\ \Leftrightarrow\ & e([r], [\tilde{x}]) = e([\tilde{x} \cdot r], [1]) \end{aligned}$$

In addition, the library does not enforce nor warn its potential users of the necessity of such proofs of possessions.

Recommendation. Document the requirements on the user of the library to use an anti-rogue attack measure, and implement the IETF's **BLS standard** (section 3.3) proposed solution.

Consider using a type-state pattern (as in **Enforce Validity of Data Statically**) to statically enforce the validity of the public keys.

01 - No Reconfiguration Protocol

*

Medium

Description. The protocol, as implemented currently, does not allow to easily reconfigure the set of validators. The set of validators is hardcoded in the genesis and cannot be changed without a manual intervention:

```
/// Genesis of the blockchain, unique for each blockchain instance.
pub struct Genesis {
     /// Genesis encoding version
    pub version: GenesisVersion,
    /// Set of validators of the chain.
    pub validators: Committee,
    /// Fork of the chain to follow.
    pub fork: Fork,
}
```

This can become problematic if validators' keys get compromised, or lost, or worse if validators turn malicious and need to be removed from the validator set.

Recommendation. We recommend addressing this issue either in documenting manual reconfiguration processes to streamline such interventions, or in researching augmentation to the current protocol to support reconfigurations (while preserving the safety of the protocol).

03 - Subquorum Choice Can Affect Liveness

validator/leader_prepare

Medium

Description. The consensus protocol must force a leader to repropose in case a commit might have happened in the previous view, even if the commit is not clearly visible. As we explained in the overview, replicas do that by ignoring proposals that do not repropose when a subquorum of $2f + 1$ "high votes" is present in the leader's certificate AND is different from the highest observed commit.

There are two problems with the way the current check is done. First, a certificate can hold as many votes as the leader wants them to carry, as long as it has more than $4f + 1$. So the leader could include $5f + 1$ votes, for example, which would lead to two subquorum potentially being present in the certificate.

In addition, the current implementation supports a "weighted" BFT variant which counts the number of votes based on the weight of each signer (who can potentially weigh for more than 1 vote). In this case, an exact quorum of $4f + 1$ votes might be impossible to achieve.

Thus, the problem remains that it is possible to have two subquorums of $2f + 1$ high votes. When faced with such a situation, replicas currently pick non-deterministically (due to the non-determinism of Rust's `HashMap`):

```
pub fn high_vote(&self, genesis: &Genesis) -> Option<BlockHeader> {
    let mut count: HashMap<_, u64> = HashMap::new();
    for (msg, signers) in &self.map {
        if let Some(v) = &msg.high_vote {
            *count.entry(v.proposal).or_default() +=
genesis.validators.weight(signers);
        }
    }
    // We only take one value from the iterator because there can only be at most one
block with a quorum of 2f+1 votes.
    let min = 2 * genesis.validators.max_faulty_weight() + 1;
    count.into_iter().find(|x| x.1 >= min).map(|x| x.0)
}
```

But the logic of the replica would later dismiss the proposal altogether if its block number doesn't match the expected one (which should be one more than the latest commit):

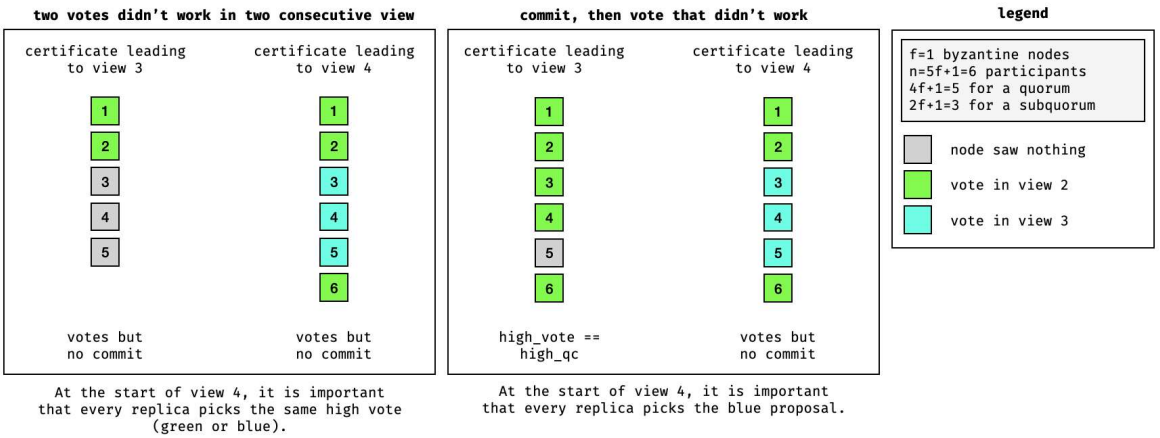
```
/// Verifies LeaderPrepare.
pub fn verify(&self, genesis: &Genesis) -> Result<(), LeaderPrepareVerifyError> {
    // TRUNCATED...

    // Check that the proposal is valid.
    match &self.proposal_payload {
```

```
        // The leader proposed a new block.
        Some(payload) => {
            // TRUNCATED...
        }
        None => {
            let Some(high_vote) = &high_vote else {
                return Err(Error::ReproposalWithoutQuorum);
            };
            if let Some(high_qc) = &high_qc {
                if high_vote.number == high_qc.header().number {
                    return Err(Error::ReproposalWhenFinalized);
                }
            }
            if high_vote != &self.proposal {
                return Err(Error::ReproposalBadBlock);
            }
        }
    }
    Ok(())
}
```

Thus this issue might lead to replicas refusing to proceed on a valid leader proposal, affecting the liveness of the protocol.

Note that if the previous proposal was committed, the problem doesn't occur as this means that $3f + 1$ honest replicas have voted for it and there aren't enough honest and dishonest replicas left to create a subquorum of $2f + 1$ high votes for a previous proposal.

Recommendation. Change the logic to return as "high vote" the proposal that has amassed at least $2f + 1$ votes AND that has the highest block number. If there are still more than one such proposal, then deterministically choose which one to pick (for example, based on its hash) or return no high vote (which will allow a reproposal). This should address the two example scenarios depicted in the following diagram:



04 - Cloneable RNG Could Lead To Predictable Randomness

libs/concurrency

Low

Description. Cryptographically-Secure Pseudo-Random Number Generators (CSPRNGs) are used to obtain random numbers throughout the protocols. Such CSPRNGs are expected to provide unpredictable numbers. While this assumption is currently correct, the internal libraries implement patterns that could lead a code change to violate the assumption.

To see why, let's look at how a context's rng is constructed when we call `ctx.rng()`:

```
// -- libs/concurrency/src/ctx/mod.rs --
impl Ctx {
    pub fn rng(&self) -> rand::rngs::StdRng {
        self.0.rng_provider.rng()
    }
}

// -- libs/concurrency/src/ctx/rng.rs --
impl Provider {
    pub(super) fn rng(&self) -> StdRng {
        StdRng::from_seed(match self {
            // StdRng seeded with OS entropy is what rand::thread_rng() does,
            // except that it caches the rng in thread-local memory and reseeds it
            // periodically. Here we push the responsibility of reseeding
            // (i.e. dropping the rng and creating another one) on the user.
            Self::OsRng => OsRng.gen(),
            Self::Split(split) => split.next_builder().finalize().into(),
        })
    }
}
```

As you can see, the output of `OsRng` is used not directly (which would be secure) but to seed a software CSPRNG. This pattern is dangerous as the seeded RNG is implemented as `Clonable`, which could lead to multiple instantiation producing the same RNG. For example:

```
#[test]
fn test_rng() {
    abort_on_panic();
    let ctx = &ctx::root();
    let rng = &mut ctx.rng();

    fn sample(mut rng: impl rand::RngCore) {
        println!("Random number: {}", rng.next_u32());
    }
```

```
    sample(rng.clone());
    sample(rng);
}
```

Recommendation. We recommend to only use `OsRng` and to avoid seeding software RNGs, unless when testing.

05 - Consensus Handshake Could Facilitate DoS Attacks

network/consensus

Low

Description. The **Noise protocol framework** is used to encrypt and authenticate communications between consensus participants of the consensus protocol. When someone attempts to connect to a node, they first perform an **Noise NN** handshake with them. A Noise NN handshake is an unauthenticated connection, sometimes called "opportunistic", that can be man-in-the-middle'd during the first few messages. It is opportunistic as the connection will be secure if an attacker is not present during this narrow window of time. That being said, it is often not deemed enough as man-in-the-middle attackers can easily force peers to restart their connection.

To strongly authenticate both sides of the encrypted channel, an additional "handshake" protocol is implemented on top of a connection secured by the Noise protocol. The handshake consists of the two peers sending each other a `Handshake` object,, which contains the genesis configuration used (to ensure that the peers are not part of a different chain) and a signature over the session ID (which is designed to be unique for every new connection).

The `inbound` function is written to follow this handshake protocol when a validator attempts to connect to our own validator:

```
pub(super) async fn inbound(
    ctx: &ctx::Ctx,
    me: &validator::SecretKey,
    genesis: validator::GenesisHash,
    stream: &mut noise::Stream,
) -> Result<validator::PublicKey, Error> {
    let ctx = &ctx.with_timeout(TIMEOUT);
    let session_id = node::SessionId(stream.id().encode());
    let h: Handshake = frame::recv_proto(ctx, stream, MAX_FRAME)
        .await
        .map_err(Error::Stream)?;
    if h.genesis != genesis {
        return Err(Error::GenesisMismatch);
    }
    if h.session_id.msg != session_id.clone() {
        return Err(Error::SessionIdMismatch);
    }
    h.session_id.verify()?;
    frame::send_proto(
        ctx,
        stream,
        &Handshake {
            session_id: me.sign_msg(session_id.clone()),
            genesis,
        },
```

```
    )
    .await
    .map_err(Error::Stream)?;
    Ok(h.session_id.key)
}
```

Interestingly, the public key of the node is only returned at the end and checked on the caller side (who will dismiss the connection if it is not from a validator).

As such, a non-consensus participant can connect to a node, establish an encrypted channel via the Noise protocol, and then force them into further potentially-expensive operations (verify a signature, sign a handshake message, send the handshake message) with their own signature even if they are not part of the validators.

Recommendation. Check that the public key (`h.session_id.key`) is part of the validator set as soon as it is received. It is easy to do that in the current function as the genesis configuration is available in the scope.

Furthermore, consider using different **Noise handshake patterns** that force the peers authentication during the handshake itself, and would allow a receiving peer to discard the connection attempt as early as it can.

Ideally, one should try to use a Noise handshake pattern that provides mutual authentication, such as the **Noise IK** pattern. This would ensure both parties are authenticated before any further operations are performed. That being said such handshake patterns come with their own downsides, like replayability of the messages or lack of **key confirmation**, and should be carefully considered before being implemented (**see how similar protocols handled these downsides**).

06 - Non-Standard BLS Implementation

crypto/bn254

Informational

Description. The consensus protocol uses the BLS signature scheme to benefit from its signature aggregation feature.

The current implementation follows the **BLS Signatures RFC** which is a well-accepted Internet Draft in the community.

The implementation does not follow the standard in a few places. For example, the key generation is simply done using rejection sampling, which seems much more simple, and the hash-to-curve algorithm used is the try-and-increment algorithm which is not part of **RFC 9380: Hashing to Elliptic Curves**.

While both examples are secure divergences of the standard, the latter might lead to non-interoperability with other libraries, which could force people to implement their own BLS implementation, which could in turn lead to security issues.

Recommendation. Choose one of the "ciphersuite" from the **BLS Signatures RFC** in order to implement one of the standardized variants of BLS.

07 - Enforce Validity of Data Statically

*

Informational

Description. In a number of places, validation of data can be enforced statistically to avoid programming mistakes that could lead to bugs.

For example, at the moment signed messages can be used without verifying their attached signature (through the `verify` function). This is due to the object giving free access to its fields:

```
pub struct Signed<V: Variant<Msg>> {
    /// The message that was signed.
    pub msg: V,
    /// The public key of the signer.
    pub key: validator::PublicKey,
    /// The signature.
    pub sig: validator::Signature,
}

impl<V: Variant<Msg> + Clone> Signed<V> {
    /// Verify the signature on the message.
    pub fn verify(&self) -> anyhow::Result<()> {
        self.sig.verify_msg(&self.msg.clone().insert(), &self.key)
    }
}
```

The validation is thus done in different places, manually. As part of the audit we confirmed that every signed message was validated when used. That being said, it can be difficult to track where a signature is being checked, and future refactors and code changes have the potential to introduce bugs.

The above example could be refactored to prevent access to a message before verification:

```
pub struct Signed<V: Variant<Msg>> {
    /// The message that was signed.
    msg: V,
    /// The public key of the signer.
    pub key: validator::PublicKey,
    /// The signature.
    pub sig: validator::Signature,
}

impl<V: Variant<Msg> + Clone> Signed<V> {
    /// Verify the signature on the message.
    pub fn verify(self) -> anyhow::Result<V> {
        self.sig.verify_msg(&self.msg.clone().insert(), &self.key)?;
        Ok(self.msg)
```

```
    }
}
```

Another example is for public keys in BLS, which need to be validated to prevent rogue key attacks (as mentioned in finding **Lack of Proof of Possession for BLS Signatures**).

To statically enforce that public keys can't be used without having been validated, we recommend a type state pattern, as described in **The Typestate Pattern in Rust** , to keep track of objects in their unvalidated and validated forms, and enforce validation before usage of the object:

```
pub struct PublicKey<const VERIFIED: bool>(u8);

impl PublicKey<false> {
    fn from_byte(byte: u8) -> Self {
        Self(byte)
    }

    fn verify(self) -> PublicKey<true> {
        // TRUNCATED verification logic...

        PublicKey(self.0)
    }
}

impl PublicKey<true> {
    fn do_stuff(&self) {
        println!("doing stuff!");
    }
}

fn main() {
    let pubkey = PublicKey::from_byte(5);
    let pubkey = pubkey.verify();
    pubkey.do_stuff();
}
```

08 - Denial of Service Due to Late Validation

bft/leader/replica_commit

Known Issue

Description. Anyone on the network can block the connection between validators, effectively performing a Denial of Service (DoS) attack on the network.

Alternatively, they can spam the storage of validators to achieve similar results.

To do this to validator V, first find all views i such that V is a leader. There's almost an infinite number of them, as you just need to solve $\mathrm{V}_{\mathrm{idx}} = \mathrm{i} \mod \mathrm{N}$ where N is the number of validators.

Then, create a message `**ReplicaCommit { view, proposal }**` with a garbage proposal, potentially sign it if you are a validator (otherwise choose a random validator V' and a garbage signature), and send it to V.

V will call `**process_replica_commit**` which will do a number of checks that will pass, and then store the garbage proposal:

```
impl StateMachine {
    #[instrument(level = "trace", skip(self), ret)]
    pub(crate) fn process_replica_commit(
        &mut self,
        ctx: &ctx::Ctx,
        signed_message: validator::Signed<validator::ReplicaCommit>,
    ) -> Result<(), Error> {
        // TRUNCATED number of checks that will pass...

        // Get current incrementally-constructed QC to work on it
        let commit_qc = self
            .commit_qcs
            .entry(message.view.number)
            .or_default()
            .entry(message.clone())
            .or_insert_with(|| CommitQC::new(message.clone(),
self.config.genesis()));

        // If we already have a message from the same validator and for the same
view, we discard it.
        let validator_view = self.validator_views.get(author);
        if validator_view.is_some_and(|view_number| *view_number >=
message.view.number) {
            return Err(Error::DuplicateSignature {
                message: commit_qc.message.clone(),
            });
        }
        self.validator_views
```

```
            .insert(author.clone(), message.view.number);

        // TRUNCATED...

        // Check the signature on the message.
        signed_message.verify().map_err(Error::InvalidSignature)?;

        // TRUNCATED...
    }
```

Due to the signature being checked after the storage update, the attack can be performed by non-validators as well.

To spam the validator's storage, either we choose a new view number as discussed above to create a new entry, or we create a new message with a new garbage proposal which will create a new subentry.

In addition, such messages in the future will also block the author V' from sending valid `ReplicaCommit` messages to this validator, as one of the checks above will prevent any messages from previous views to be processed.

Note that after this finding was reported, Matter Labs informed us that they were already aware of it and even had a PR (#100) fixing it. As such, this finding is marked as a "known issue".

Recommendation. Move the signature check before all storage updates.