HotStuff: BFT Consensus with Linearity and Responsiveness

Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., & Abraham, I.

PODC ‘19

Contents

1. Contributions
2. Prior works
3. System model
4. Review: 2-phase paradigm (PBFT, Tendermint)
5. Basic HotStuff
6. Chained HotStuff

Contributions

- **Linear view change**
  - Use a threshold signature scheme of which threshold is $2f + 1$.
- **Optimistic Responsiveness**
  - *Responsiveness* requires that a non-faulty leader can drive the protocol to consensus in time depending only on the actual message delays, independent of any known upper bound on message transmission delays
  - *Optimistic* means that *responsiveness* is required only in beneficial circumstances—here, after GST is reached.

Prior works

Protocol	Authenticator complexity *Correct leader*	 *Leader failure (view-change)*	 *f leader failures*	Responsiveness
DLS [25]	$O(n^4)$	$O(n^4)$	$O(n^4)$	
PBFT [20]	$O(n^2)$	$O(n^3)$	$O(fn^3)$	✓
SBFT [30]	$O(n)$	$O(n^2)$	$O(fn^2)$	✓
Tendermint [15] / Casper [17]	$O(n^2)$	$O(n^2)$	$O(fn^2)$	
Tendermint* / Casper*	$O(n)$	$O(n)$	$O(fn)$	
HotStuff	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(fn)}$	✓

*Signatures can be combined using threshold signatures, though this optimization is not mentioned in their original works.

Table 1: Performance of selected protocols after GST.

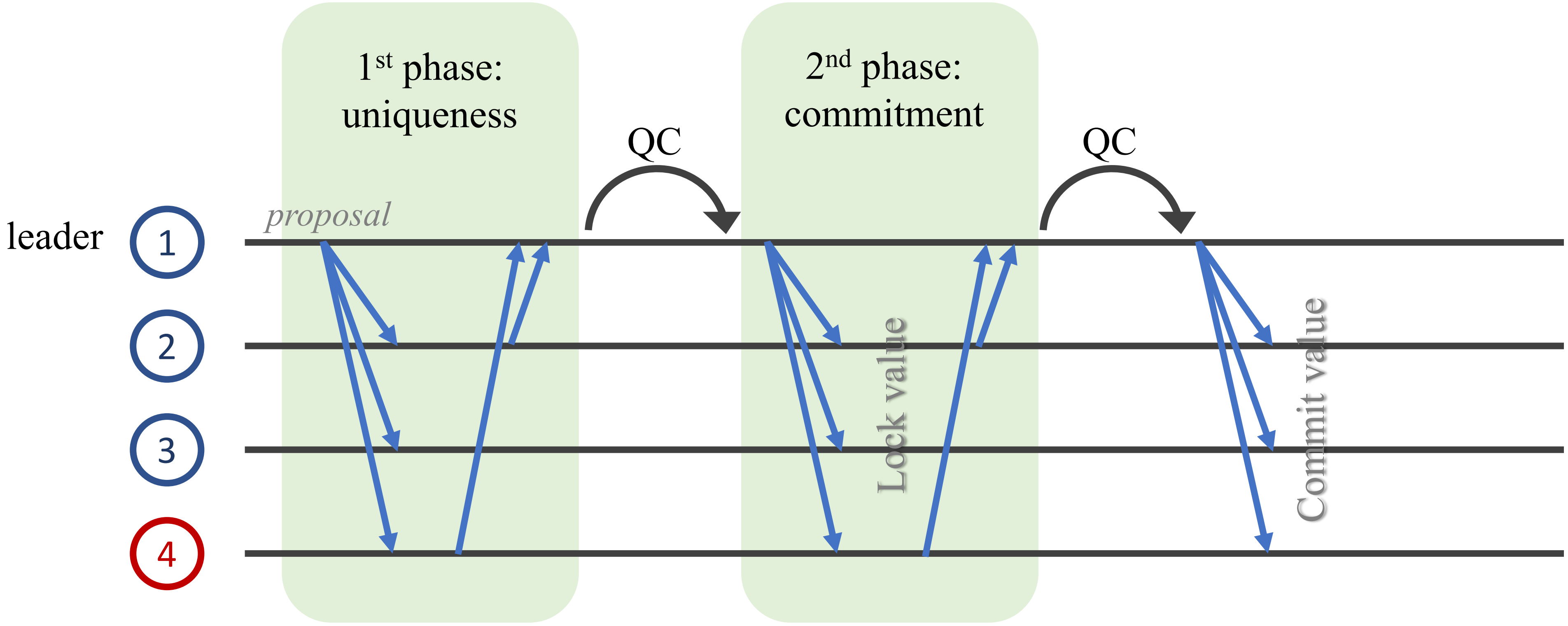
System Model (1)

- **Security model**
  - An *adversary* coordinates Byzantine replicas and learns all internal states held by these replicas
- **Network assumption**
  - Communication is p2p, authenticated and reliable.
  - “broadcast” involves the sender.
  - Partial synchrony model.

System Model (2)

- **Threshold Signature**
  - A single public key held by all replicas, and each of the n replicas holds a distinct private key.
  - The i-th replica can use its private key to contribute a partial signature $\rho_i \leftarrow \boldsymbol{tsign_i}(m)$ on message m.
  - $\sigma \leftarrow \boldsymbol{tcombine}(m, \{\rho i\}_{i \in I})$ on message m, where $|I| = threshold = 2f + 1$.
  - Any other replica can verify the signature using the public key and the function $\boldsymbol{tverify}(\sigma, m)$
- **Authenticator complexity**
  - an *authenticator* is either a partial signature or a signature.
  - Sum of the number of *authenticator*s received by replica i in the protocol to reach a consensus decision.

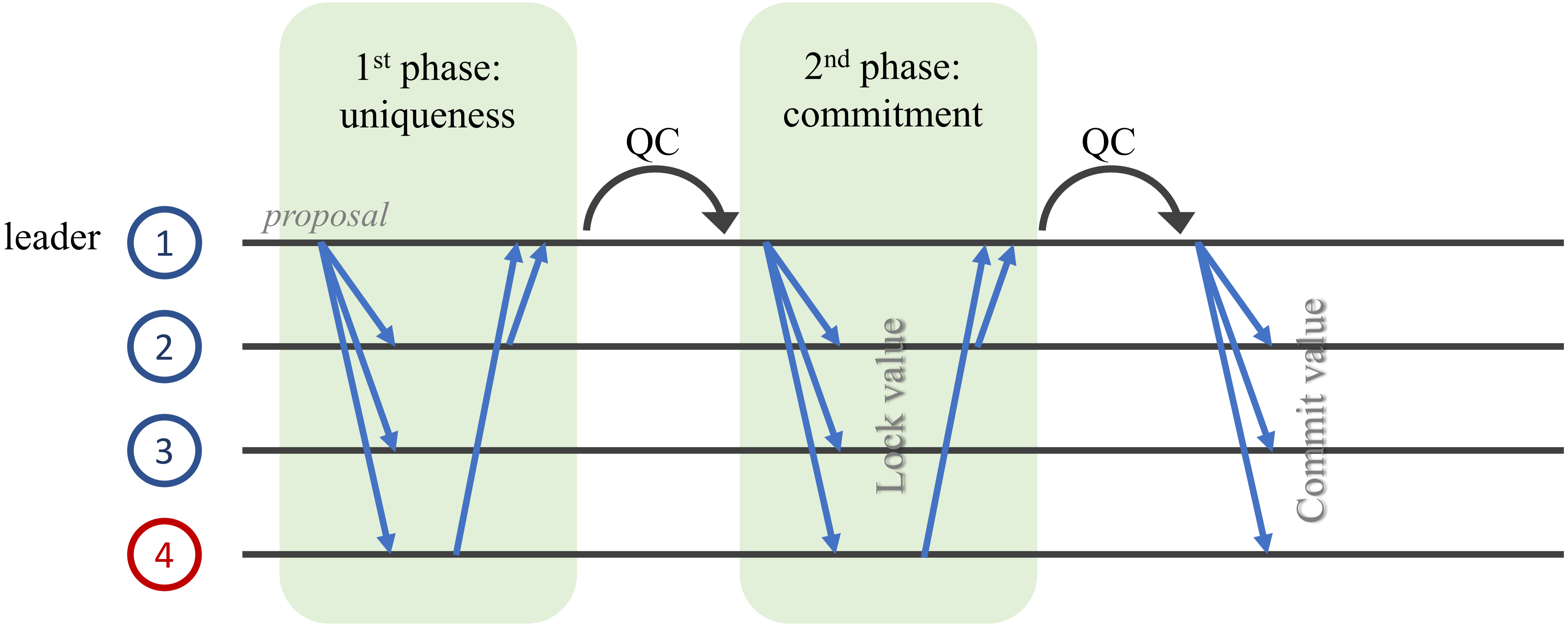
Review: 2-phase paradigm



- ***1st phase :*** guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n − f) votes. validity condition is important.
- ***2nd phase*** : guarantees that the next leader can convince replicas to vote for a safe proposal, using *commit-adopt**

- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

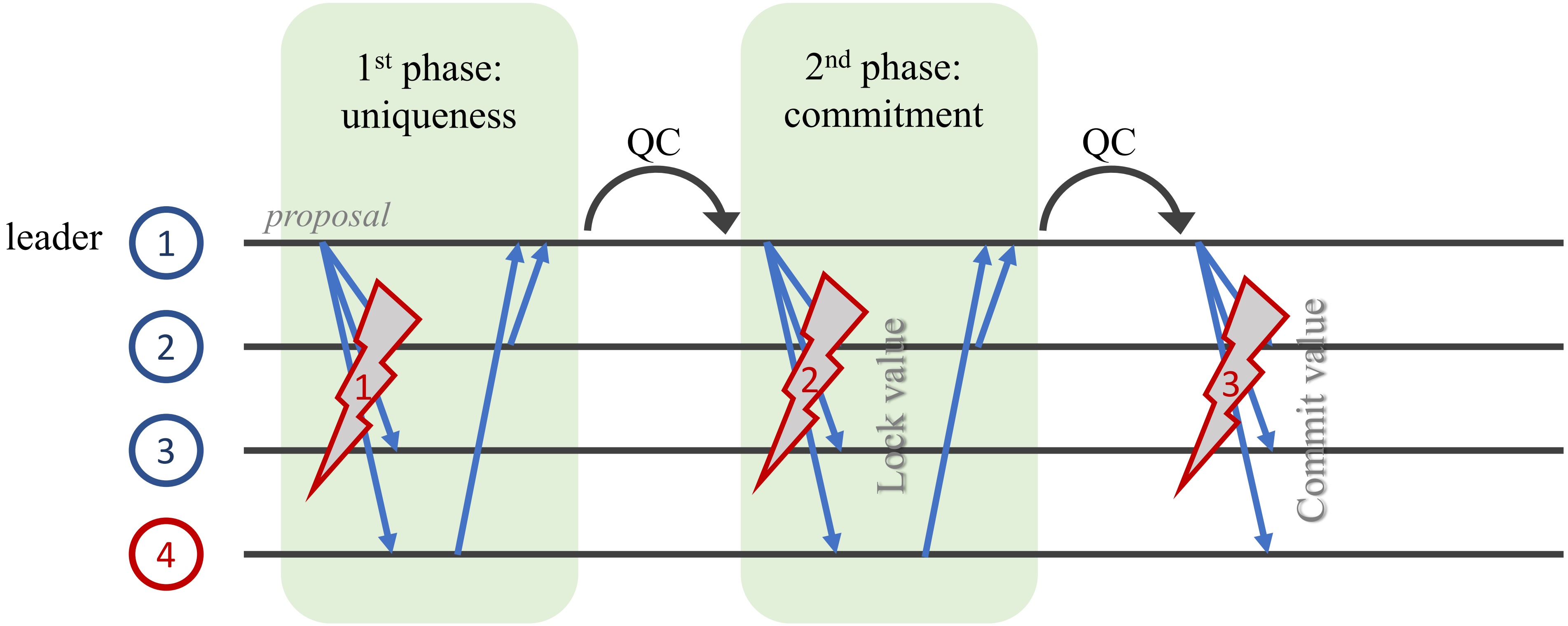
Review: 2-phase paradigm (cont.)



- ***Commit-Adopt***: if a correct replica commits a value v, at least $2f+1$ replicas adopt and protect v (lock the value v). At least $f+1$ replicas are correct among those replicas, and the corresponding $f+1$ replicas guard safety of the committed value v. Thus, they would not vote on a conflicting value unless they are shown a proof that it is safe to do so.

- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

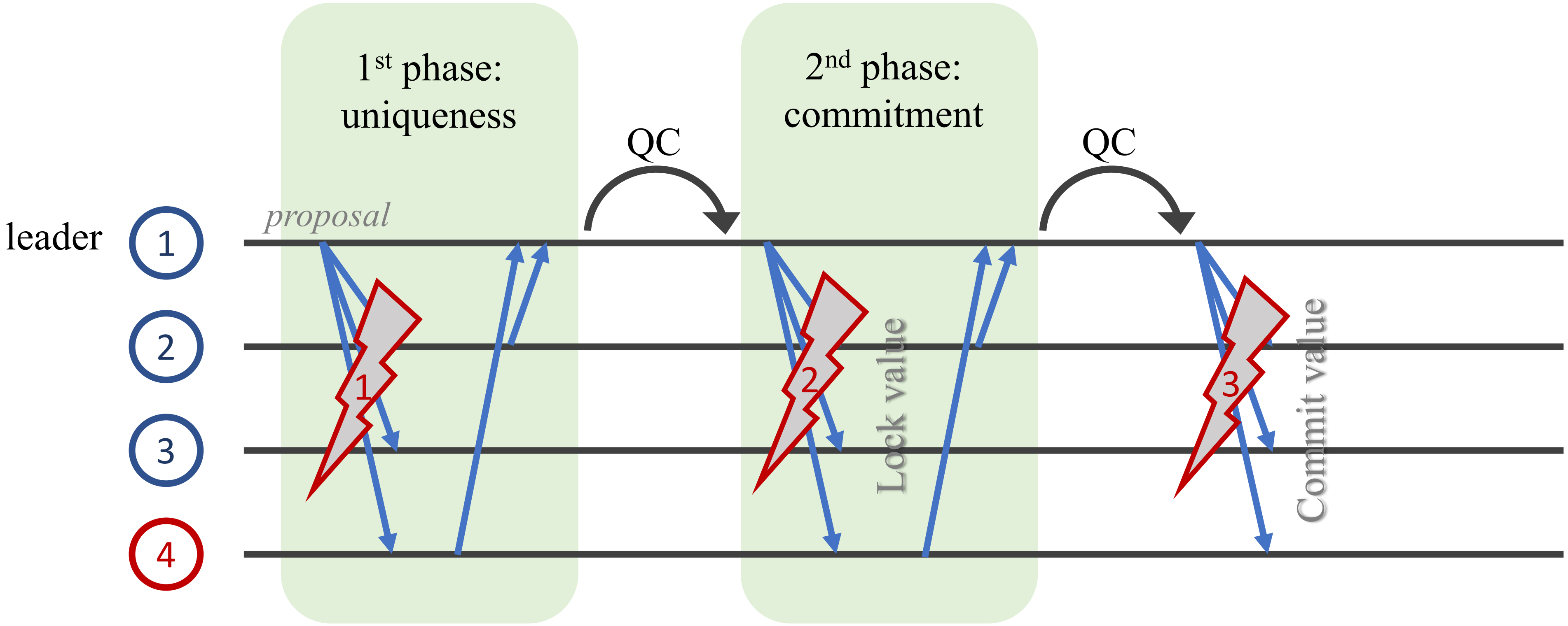
Review: failure cases



1. No replica has a lock, no value is committed.
2. One or a few replicas are locked on a value, but no correct replica has committed.
3. $2f + 1$ parties are locked, some correct replicas have committed but not all

- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

Review: failure cases



1. ~~No replica has a lock, no value is committed.~~
2. One or a few replicas are locked on a value, but no correct replica has committed.
3. ~~$2f + 1$ parties are locked, some correct replicas have committed but not all~~

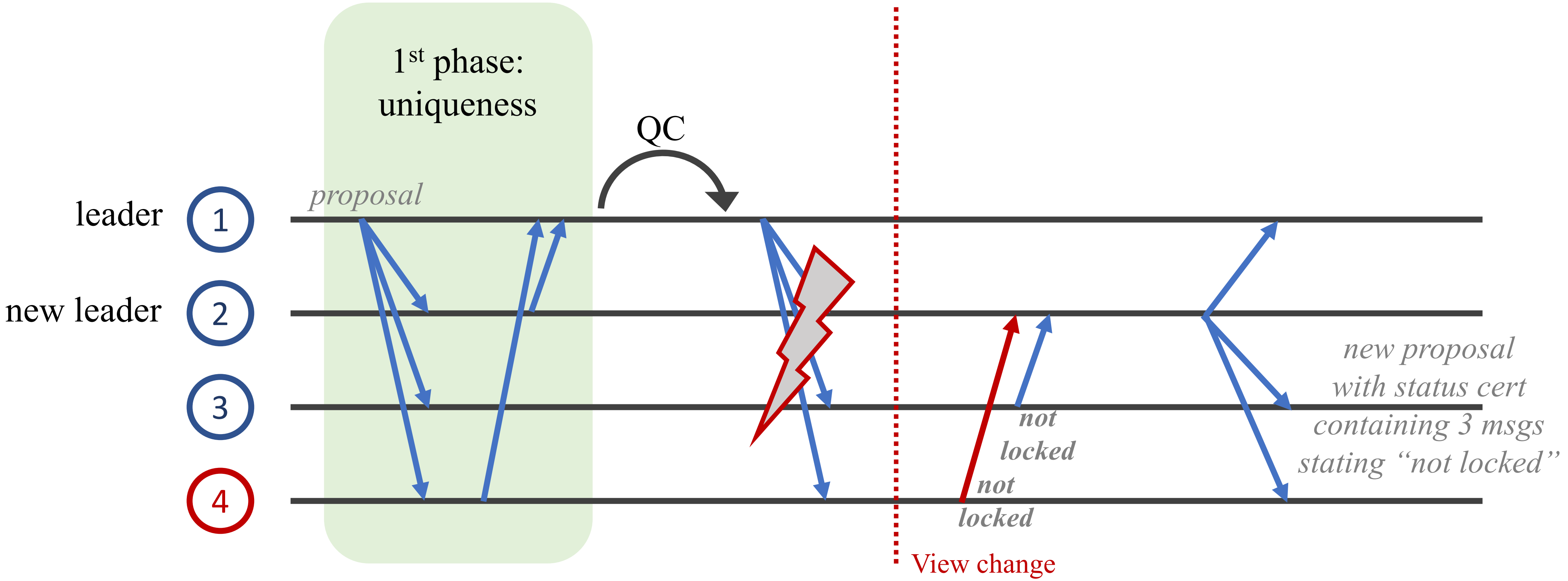
- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

Review: failure cases

- In case 2, some mechanism is required to ensure that <u>these locked correct replicas vote for a safe proposal from a leader (perhaps conflicting with a locked value) after a view-change.</u>
    1. What does the leader learn about the status of the system (the three scenarios) at the start of a view?
    2. How does the leader convince other replicas about the status of the system (and thus to vote for its proposal)?

- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

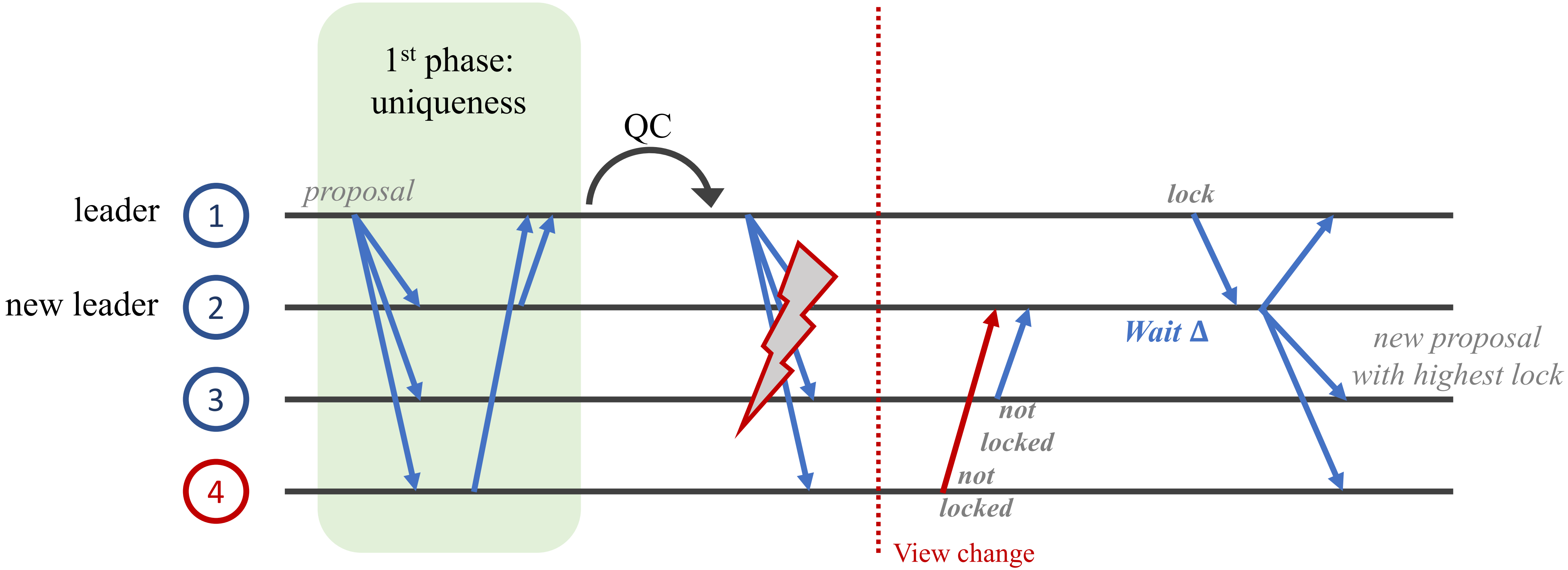
Review: PBFT



- Leader sends the *status certificate* containing the $2f+1$ locks in its proposal.
- Quadratic complexity / Responsiveness

- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

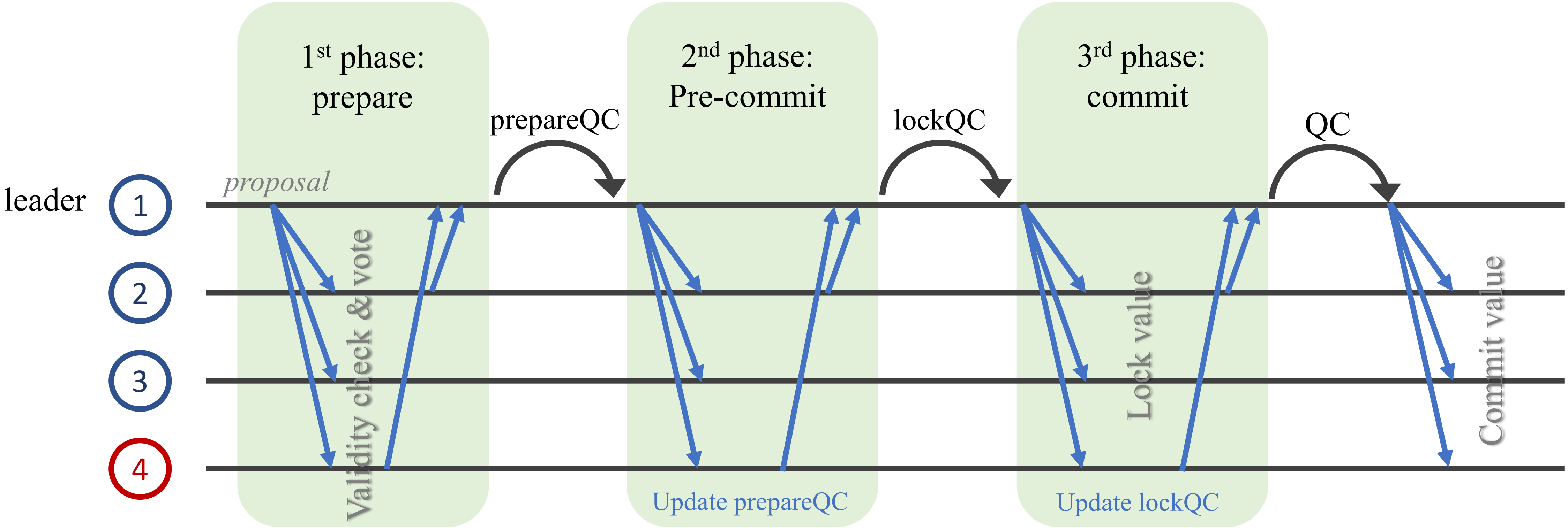
Review: Tendermint



- Avoid sending a $(2f+1)$-sized status certificate →linear view-change
- During **Δ**, the leader collects all locks and make new proposal with highest lock.
- Linear view-change / Non-responsiveness

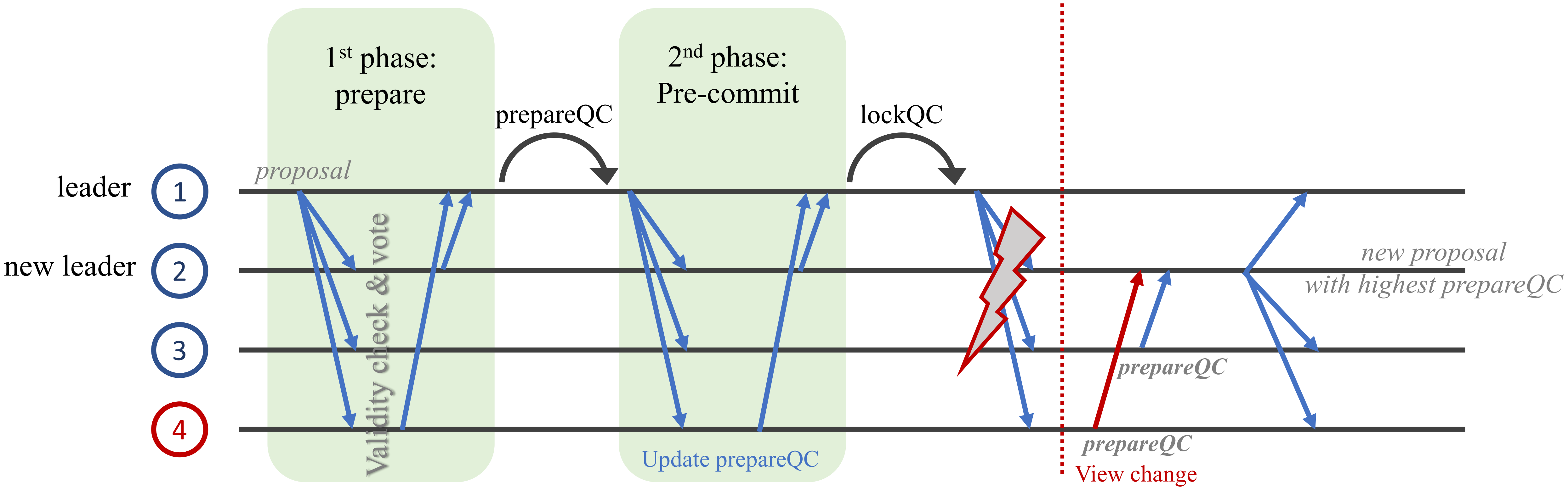
- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

Basic HotStuff



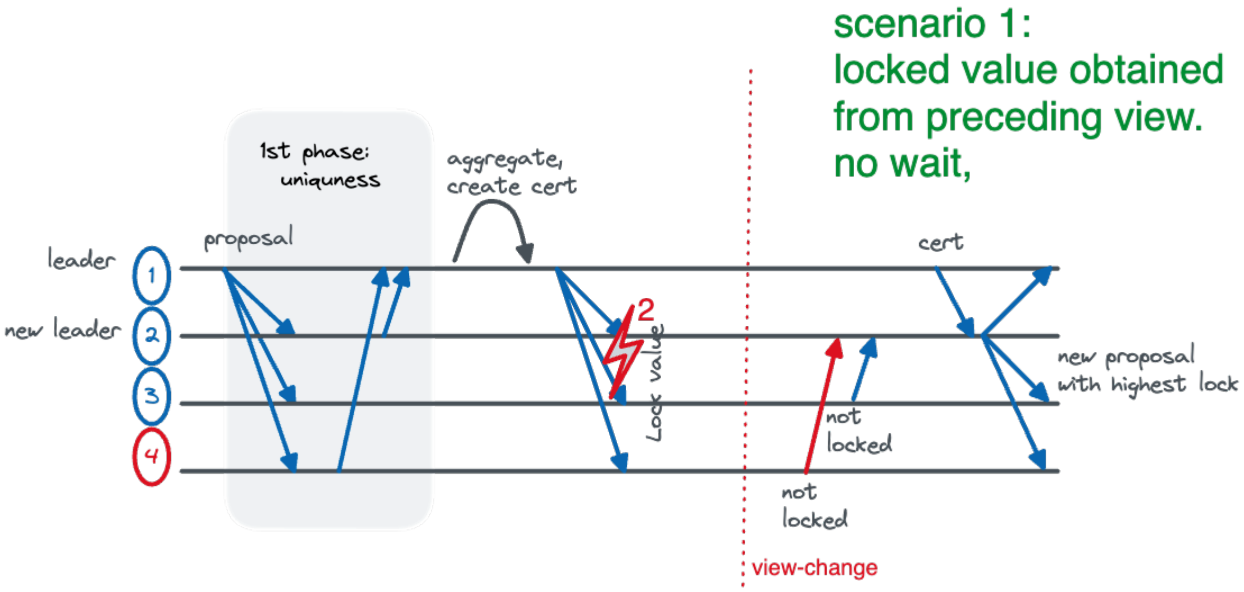
- prepareQC : qualified certificate for $2f + 1$ validity votes.
- lockQC: qualified certificate for $2f + 1$ locks on a proposal.

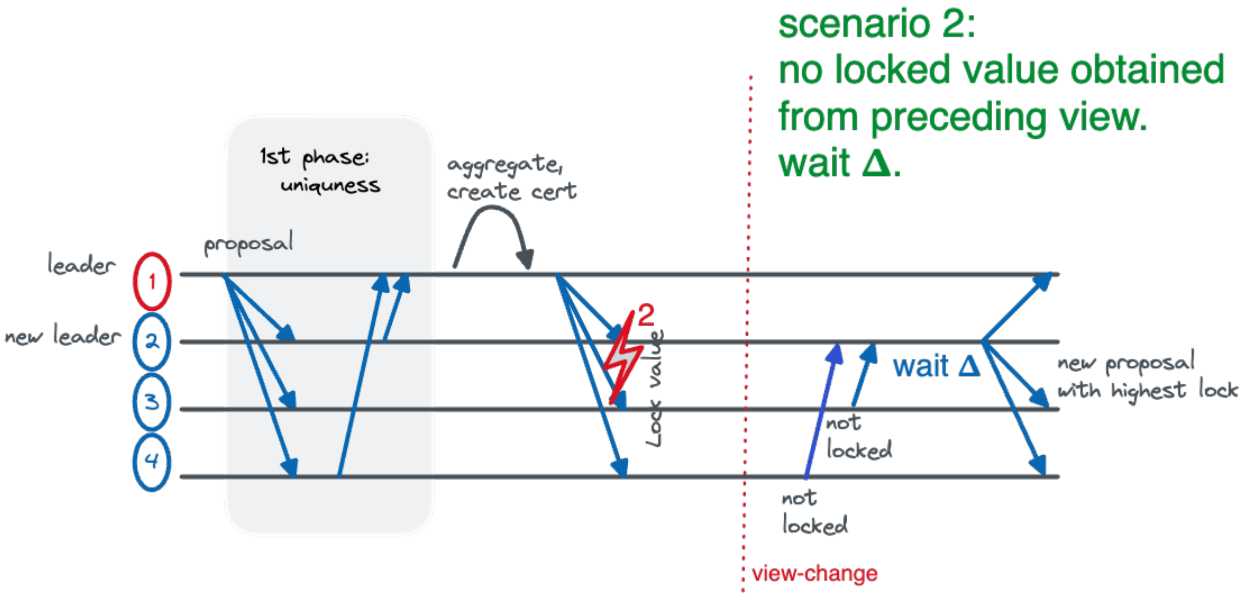
Basic HotStuff _ mitigation of case-2



- Each replica sends its own prepareQC (replica3 in not locked but has a prepareQC).
- Upon receiving 2f+1 prepareQCs, leader makes new proposal with $highQC$ among those prepareQCs.
- Locked correct replicas vote to new proposal when highQC > lockedQC.

Cf) HotStuff-2





- Highest *qc* must be found in 2-phase BFT.
- If the leader obtains a *qc* from the preceding view, the *qc* is maximal value that possibly exists in the system. In this case, it proceeds with a proposal in a responsive manner.
- Otherwise, must waits **Δ**. This can happen when the leader from the previous view is malicious or the previous view was before GST.

- https://decentralizedthoughts.github.io/2023-04-01-hotstuff-2/
- Dahlia Malkhi and Kartik Nayak, Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT, cryptoeprint:2023/397, 2023

Basic HotStuff _ pseudocode(1)

Algorithm 1 Utilities (for replica r).

```
function Msg(type, node, qc)
    m.type ← type
    m.viewNumber ← curView
    m.node ← node
    m.justify ← qc
    return m
function voteMsg(type, node, qc)
    m ← Msg(type, node, qc)
    m.partialSig ← tsign_r(⟨m.type, m.viewNumber, m.node⟩)
    return m
procedure createLeaf(parent, cmd)
    b.parent ← parent
    b.cmd ← cmd
    return b
function QC(V)
    qc.type ← m.type : m ∈ V
    qc.viewNumber ← m.viewNumber : m ∈ V
    qc.node ← m.node : m ∈ V
    qc.sig ← tcombine(⟨qc.type, qc.viewNumber, qc.node⟩,
                      {m.partialSig | m ∈ V})
    return qc
function matchingMsg(m, t, v)
    return (m.type = t) ∧ (m.viewNumber = v)
function matchingQC(qc, t, v)
    return (qc.type = t) ∧ (qc.viewNumber = v)
function safeNode(node, qc)
    return (node extends from lockedQC.node) ∨      // safety rule
    (qc.viewNumber > lockedQC.viewNumber)           // liveness rule
```

Algorithm 2 Basic HotStuff protocol (for replica r).

```
for curView ← 1, 2, 3, ... do
    ▷ prepare phase
    as a leader    // r = leader(curView)
        // we assume special new-view messages from view 0
        wait for (n − f) new-view messages: M ← {m | matchingMsg(m, new-view, curView − 1)}
        highQC ← (arg max_{m∈M} {m.justify.viewNumber}).justify
        curProposal ← createLeaf(highQC.node, client's command)
        broadcast Msg(prepare, curProposal, highQC)
    as a replica
        wait for message m : matchingMsg(m, prepare, curView) from leader(curView)
        if m.node extends from m.justify.node ∧
                    safeNode(m.node, m.justify) then
            send voteMsg(prepare, m.node, ⊥) to leader(curView)
    ▷ pre-commit phase
    as a leader
        wait for (n − f) votes: V ← {v | matchingMsg(v, prepare, curView)}
        prepareQC ← QC(V)
        broadcast Msg(pre-commit, ⊥, prepareQC)
    as a replica
        wait for message m : matchingQC(m.justify, prepare, curView) from leader(curView)
        prepareQC ← m.justify
        send voteMsg(pre-commit, m.justify.node, ⊥) to leader(curView)
```

Basic HotStuff _ pseudocode(2)

Algorithm 1 Utilities (for replica *r*).

```
function Msg(type, node, qc)
    m.type ← type
    m.viewNumber ← curView
    m.node ← node
    m.justify ← qc
    return m
function voteMsg(type, node, qc)
    m ← Msg(type, node, qc)
    m.partialSig ← tsign_r(⟨m.type, m.viewNumber, m.node⟩)
    return m
procedure createLeaf(parent, cmd)
    b.parent ← parent
    b.cmd ← cmd
    return b
function QC(V)
    qc.type ← m.type : m ∈ V
    qc.viewNumber ← m.viewNumber : m ∈ V
    qc.node ← m.node : m ∈ V
    qc.sig ← tcombine(⟨qc.type, qc.viewNumber, qc.node⟩,
                      {m.partialSig | m ∈ V})
    return qc
function matchingMsg(m, t, v)
    return (m.type = t) ∧ (m.viewNumber = v)
function matchingQC(qc, t, v)
    return (qc.type = t) ∧ (qc.viewNumber = v)
function safeNode(node, qc)
    return (node extends from lockedQC.node) ∨     // safety rule
    (qc.viewNumber > lockedQC.viewNumber)          // liveness rule
```

```
▷ commit phase
    as a leader
        wait for (n − f) votes: V ← {v | matchingMsg(v, pre-commit, curView)}
        precommitQC ← QC(V)
        broadcast Msg(commit, ⊥, precommitQC)
    as a replica
        wait for message m : matchingQC(m.justify, pre-commit, curView) from leader(curView)
        lockedQC ← m.justify
        send voteMsg(commit, m.justify.node, ⊥) to leader(curView)
▷ decide phase
    as a leader
        wait for (n − f) votes: V ← {v | matchingMsg(v, commit, curView)}
        commitQC ← QC(V)
        broadcast Msg(decide, ⊥, commitQC)
    as a replica
        wait for message m from leader(curView)
        wait for message m : matchingQC(m.justify, commit, curView) from leader(curView)
        execute new commands through m.justify.node, respond to clients
▷ Finally
    nextView interrupt: goto this line if nextView(curView) is called during "wait for" in any phase
    send Msg(new-view, ⊥, prepareQC) to leader(curView + 1)
```

Chained HotStuff

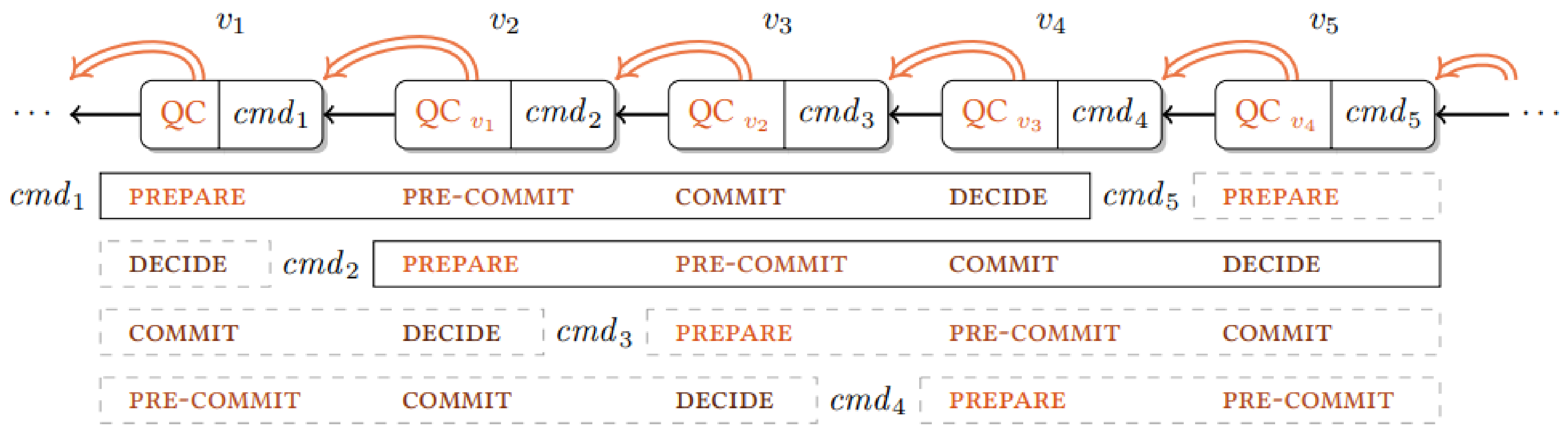


Figure 1: Chained HotStuff is a pipelined Basic HotStuff where a QC can serve in different phases simultaneously.

Chained HotStuff _ pseudocode

```
for curView ← 1, 2, 3, . . . do
  ▷ GENERIC phase
    as a leader   // r = LEADER(curView)
    // M is the set of messages collected at the end of previous view by the leader of this view
        highQC ← ( arg max_{m∈M} {m.justify.viewNumber} ).justify
        if highQC.viewNumber > genericQC.viewNumber then genericQC ← highQC
        curProposal ← CREATELEAF(genericQC.node, client's command, genericQC)
        // PREPARE phase
        broadcast MSG(GENERIC, curProposal, ⊥)
    as a replica
        wait for message m : MATCHINGMSG(m, GENERIC, curView) from LEADER(curView)
        b* ← m.node; b'' ← b*.justify.node; b' ← b''.justify.node; b ← b'.justify.node
        if SAFENODE(b*, b*.justify) then
            send VOTEMSG(GENERIC, b*, ⊥) to LEADER(curView + 1)
        // start PRE-COMMIT phase on b*'s parent
        if b*.parent = b'' then
            genericQC ← b*.justify
        // start COMMIT phase on b*'s grandparent
        if (b*.parent = b'') ∧ (b''.parent = b') then
            lockedQC ← b''.justify
        // start DECIDE phase on b*'s great-grandparent
        if (b*.parent = b'') ∧ (b''.parent = b') ∧ (b'.parent = b)  then
            execute new commands through b, respond to clients
    as the next leader
        wait for all messages: M ← {m | MATCHINGMSG(m, GENERIC, curView)}
            until there are (n − f) votes: V ← {v | v.partialSig ≠ ⊥ ∧ v ∈ M}
        genericQC ← QC(V)
  ▷ Finally
    NEXTVIEW interrupt: goto this line if NEXTVIEW(curView) is called during "wait for" in any phase
    send MSG(GENERIC, ⊥, genericQC) to LEADER(curView + 1)
```

Evaluation _ setup

- EC2, c5.4xlarge (16 vCPUs supported by Intel Xeon Platinum 8000 processors @ 3.4GHz)
- The maximum TCP bandwidth measured by iperf was around 1.2 Gigabytes per second.
- The network latency between two machines was less than 1 ms. (LAN?)

Evaluation _ base performance

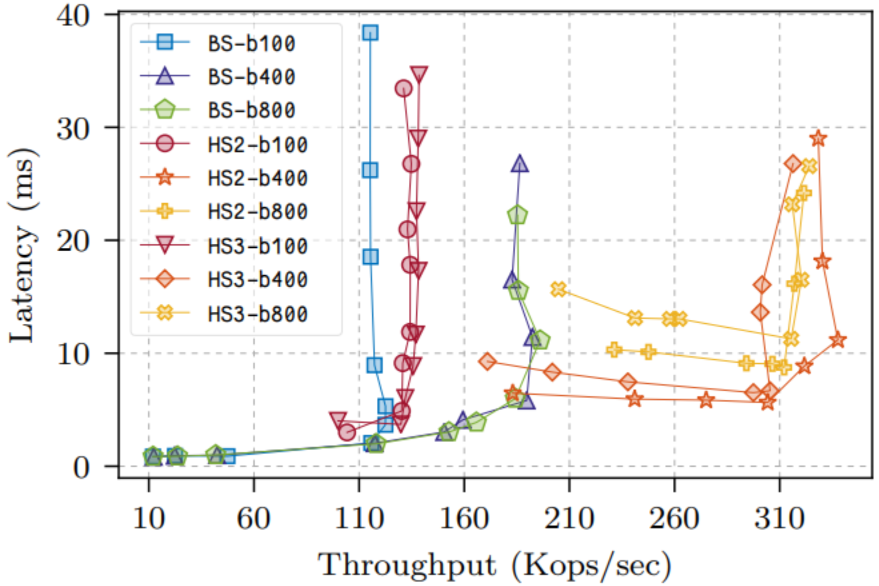


Figure 4: Throughput vs. latency with different choices of batch size, 4 replicas, 0/0 payload.

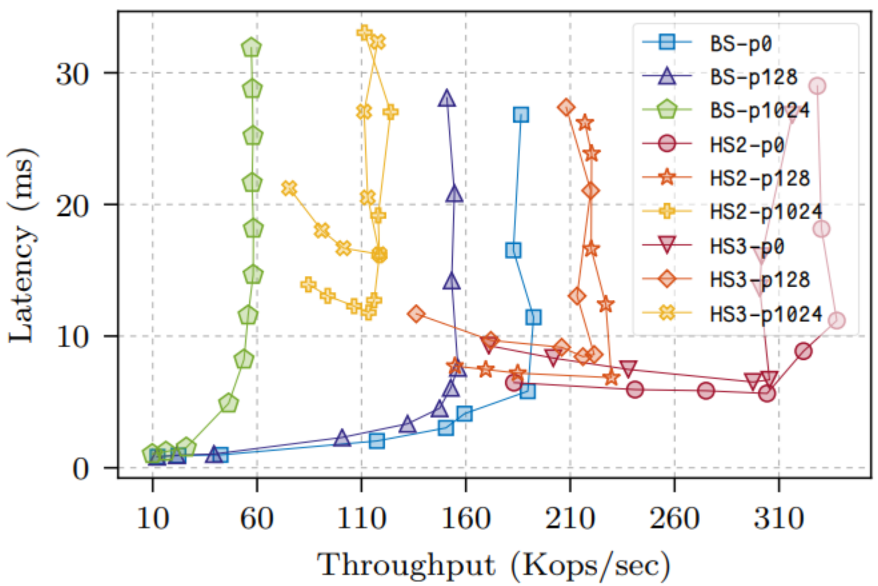


Figure 5: Throughput vs. latency with different choices of payload size, 4 replicas, batch size of 400.

Evaluation _ scalability

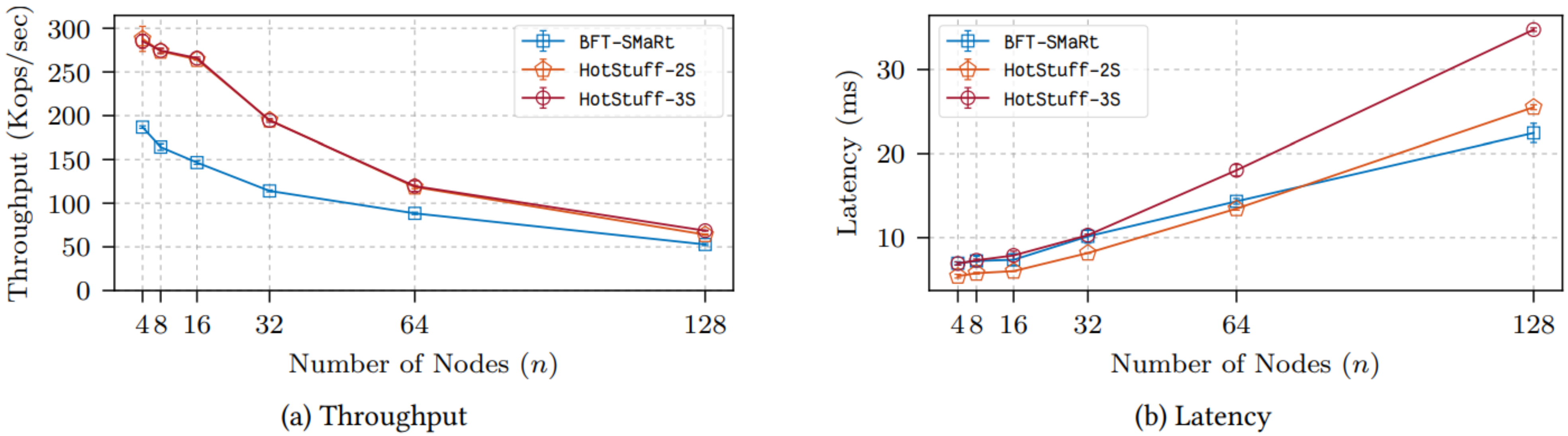
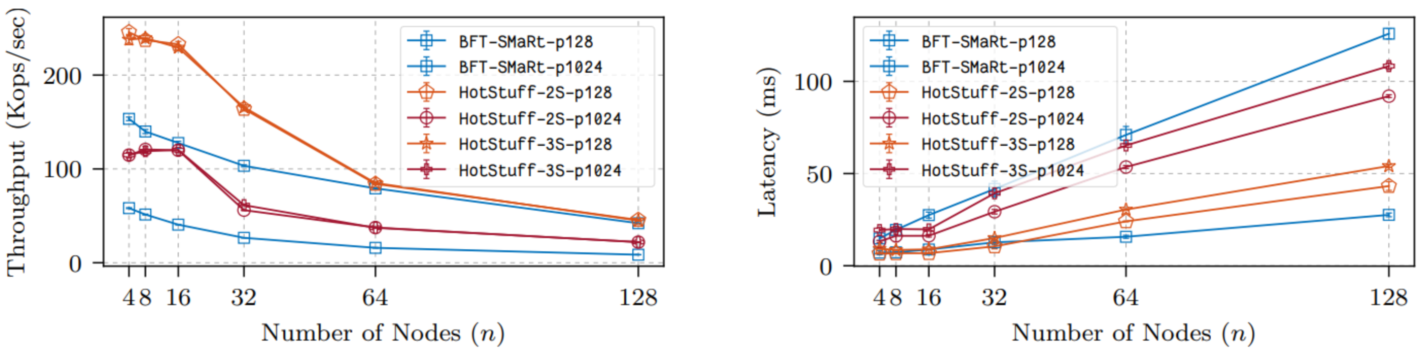


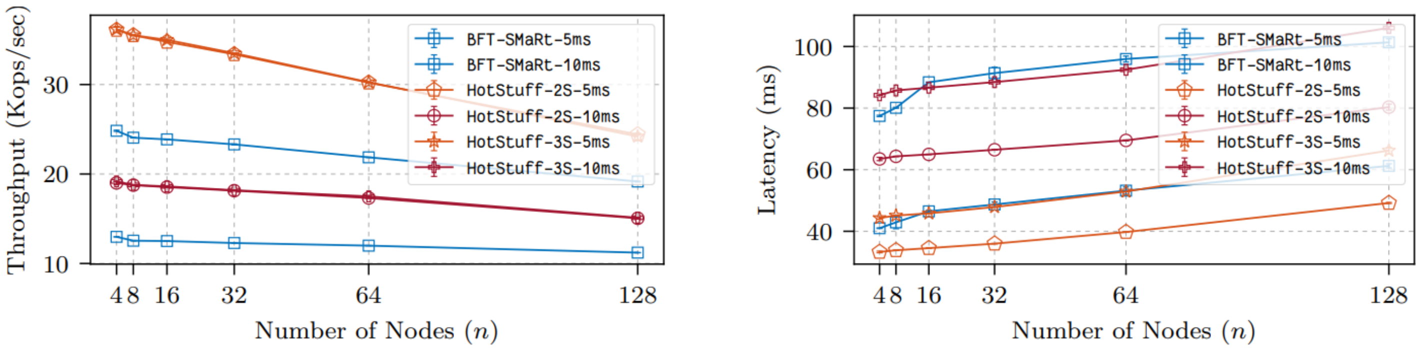
Figure 6: Scalability with 0/0 payload, batch size of 400.

Evaluation _ scalability



(a) Throughput (b) Latency

Figure 7: Scalability for 128/128 payload or 1024/1024 payload, with batch size of 400.



(a) Throughput (b) Latency

Figure 8: Scalability for inter-replica latency 5ms ± 0.5ms or 10ms ± 1.0ms, with 0/0 payload, batch size of 400.

Evaluation _ view change

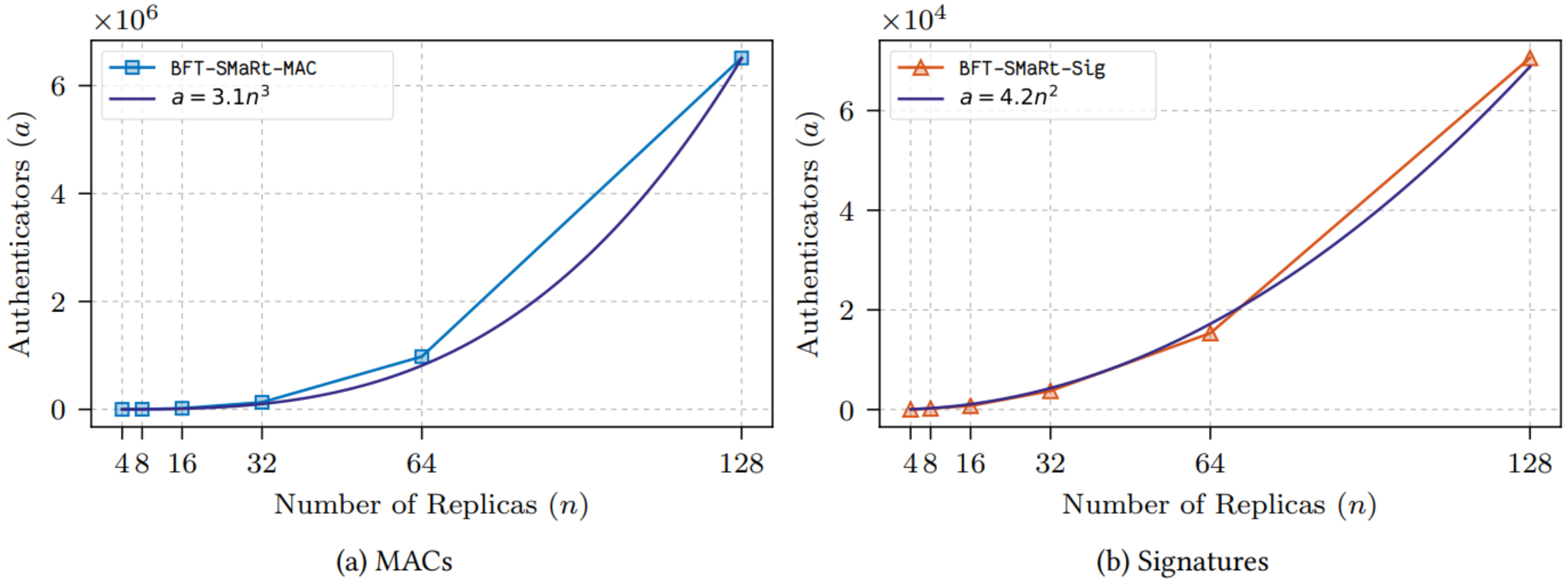


Figure 10: Number of extra authenticators used for each BFT-SMaRt view change.