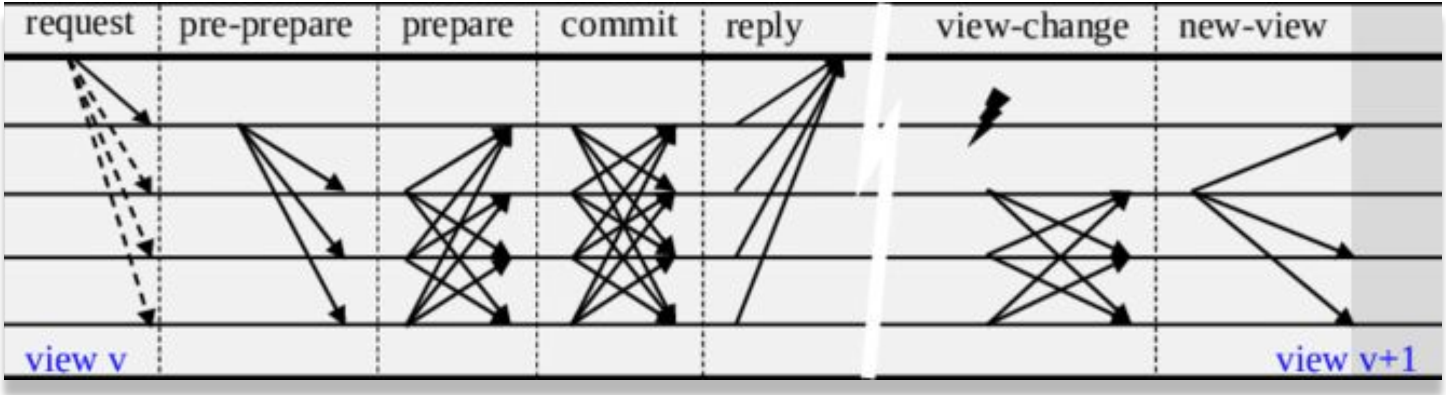
HotStuff

BFT Consensus in the Lens of Blockchain

Xianda Hou, Oliver Shen, Ashwin Sekhari, Sheshavishnuprasad D, Mythreya K

The Problem

- View-changes are buggy and time-consuming in PBFT



Step 1: When faulty primary is detected, replica sends VIEW-CHANGE message

Step 2: The next primary in line sends a NEW-VIEW message to everyone

What HotStuff Offers

- Quicker view-changes
    - Achieved linearly, O(n) messages
    - Cost is small enough to where it can change views after every protocol
- Optimistic Responsiveness
    - New leader only needs n-f responses to know progress can be made

Model

Network assumptions

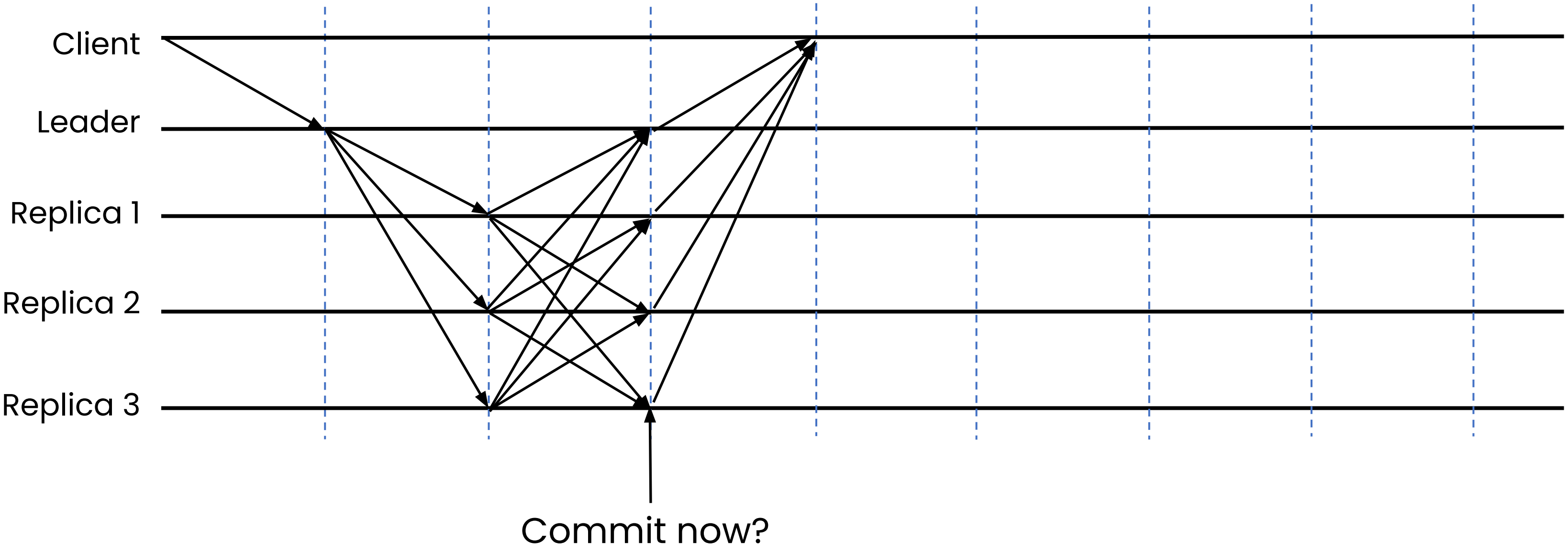
- Synchrony: Known upper bound on the message delays.
- Asynchrony (asynchrony): No known upper bound.
- **Partial synchrony:** The system has an uncertain GST (global stable time) and a Δ , so that the system is in a synchronized state within Δ after the end of GST .

HotStuff works in a Partially Synchronous model!

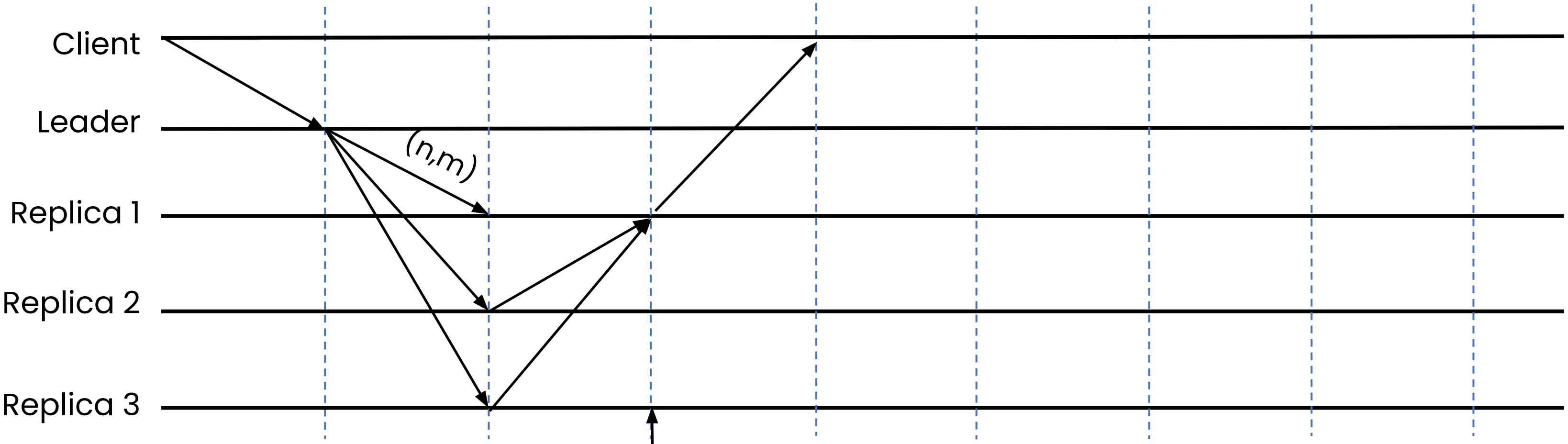
Transition

- 1-Phase PBFT?
- 2-Phase PBFT
- 2-Phase PBFT Without View Change?
- 2-Phase HotStuff
- 2-Phase HotStuff with Optimistic Responsiveness?
- 3-Phase HotStuff (Basic HotStuff)
- Chained HotStuff

1-Phase PBFT?

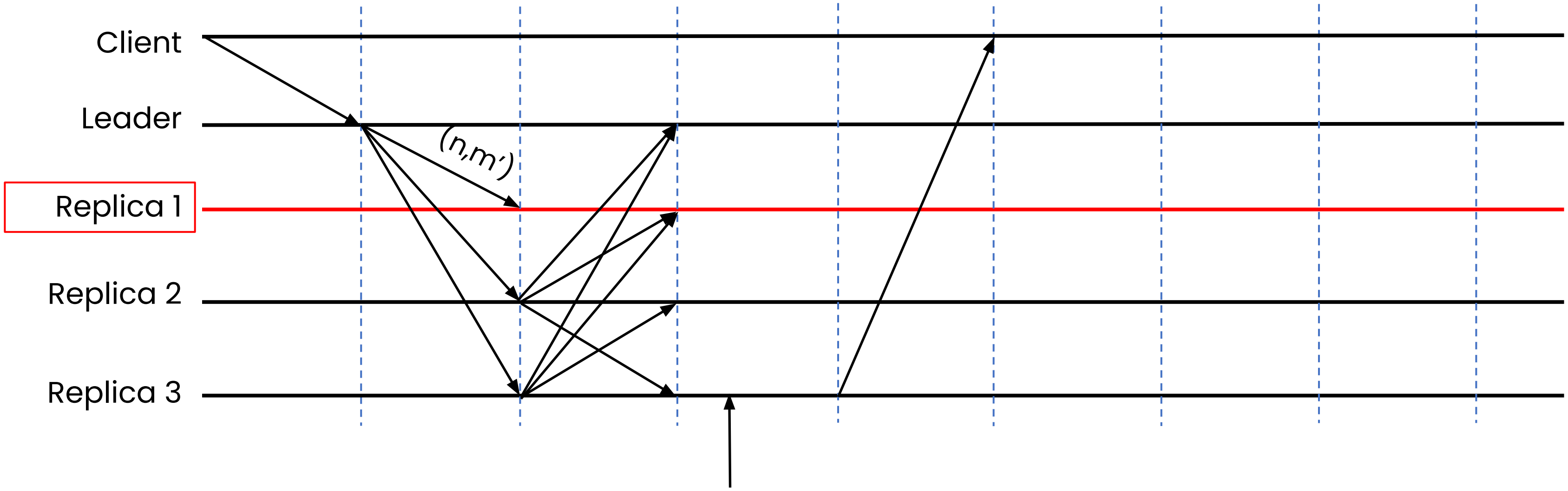


1-Phase PBFT - Problem



Replica 1 committed on (n, m)

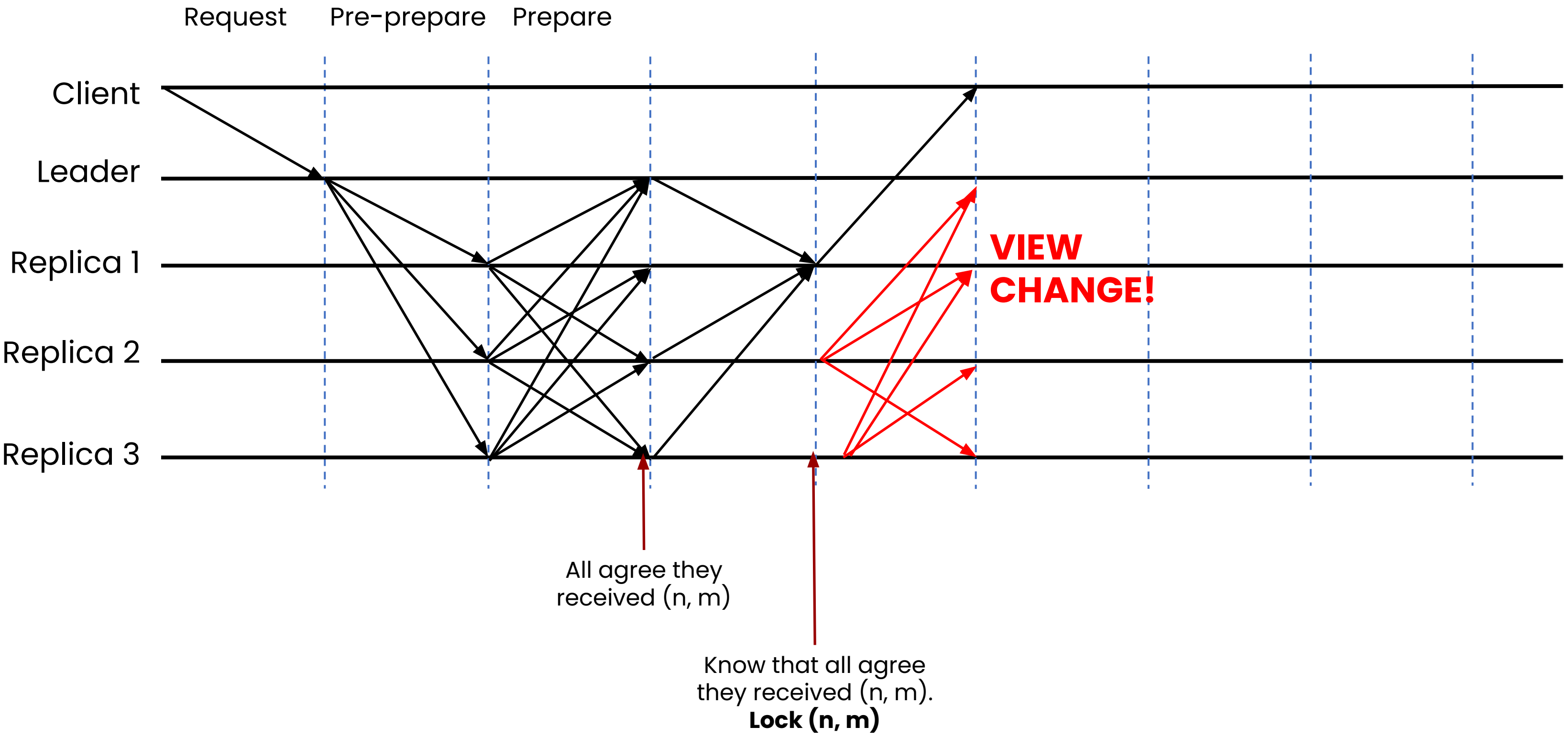
1-Phase PBFT - Problem



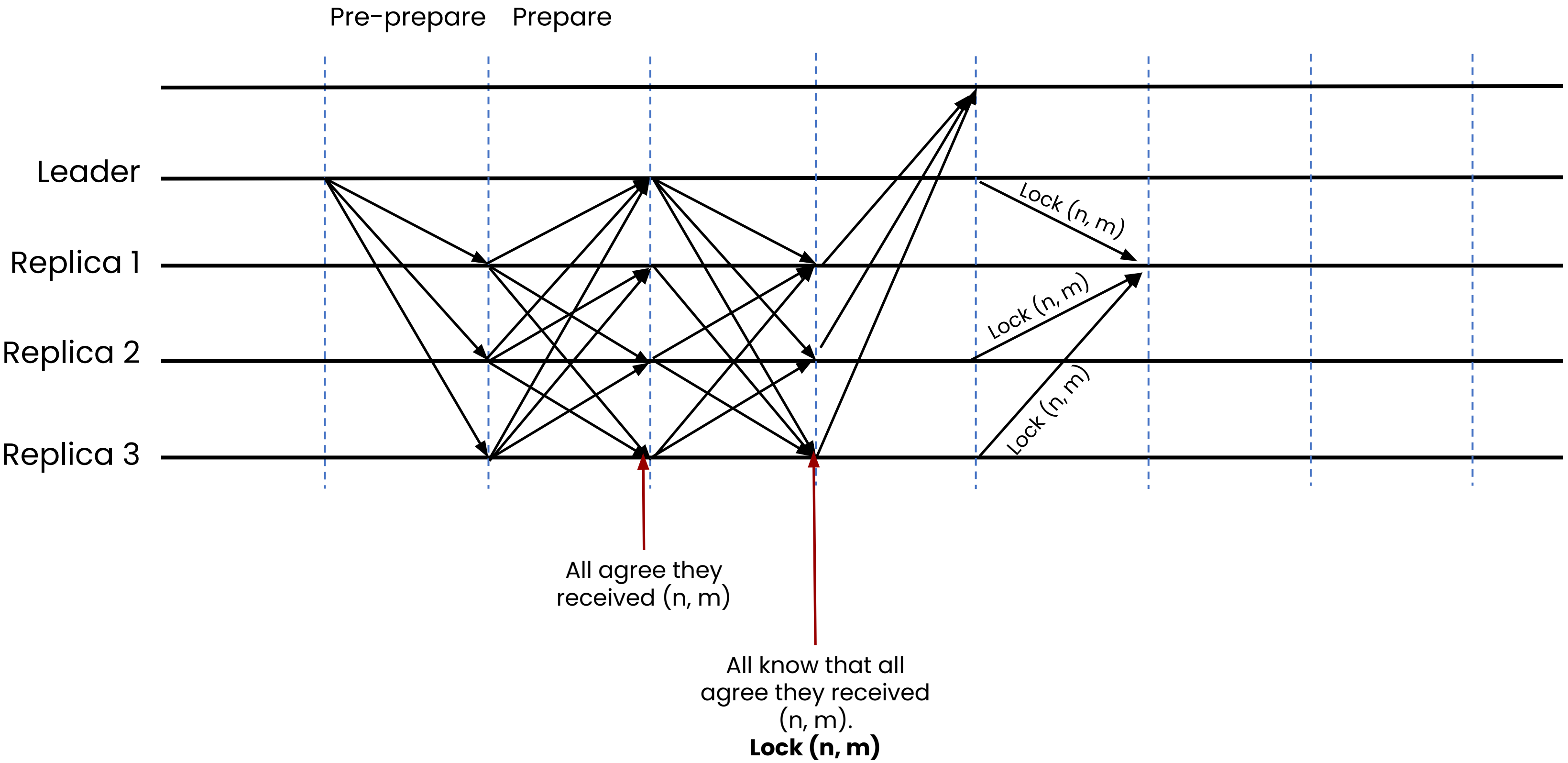
Committed (n, m')
Violation of Safety!

When someone gets stuck, add a phase!

2-Phase PBFT


Request
Pre-prepare
Prepare
Client
Leader
Replica 1
Replica 2
Replica 3
VIEW CHANGE!
All agree they received (n, m)
Know that all agree they received (n, m).
Lock (n, m)

2-Phase PBFT without View Change?



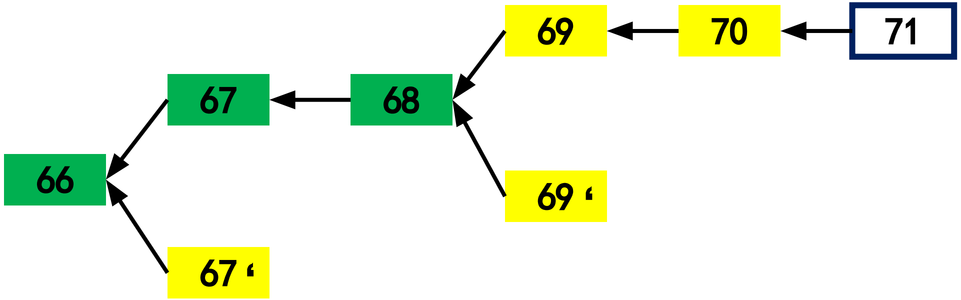
Definitions

Quorum Certificate

Combines a collection of signatures for the same tuple <*type, viewNumber, node*> signed by (n - f) replicas

Tree and branches

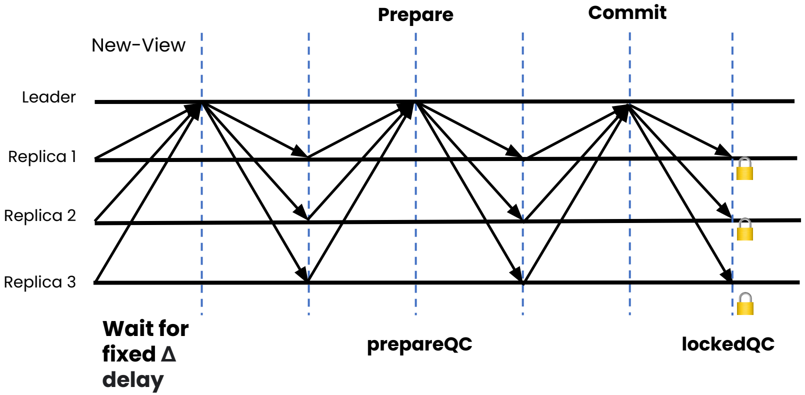
Each command is wrapped in a node that additionally contains a parent link which could be a hash digest of the parent node. Also, in practice, a replica who falls behind can catch up by fetching missing nodes from other replicas



Leader Designation

HotStuff works in a succession of views numbered with monotonically increasing view numbers. Each *viewNumber* has a unique dedicated leader know to all

2-Phase HotStuff



Messages are stored in the form of nodes.

Leader

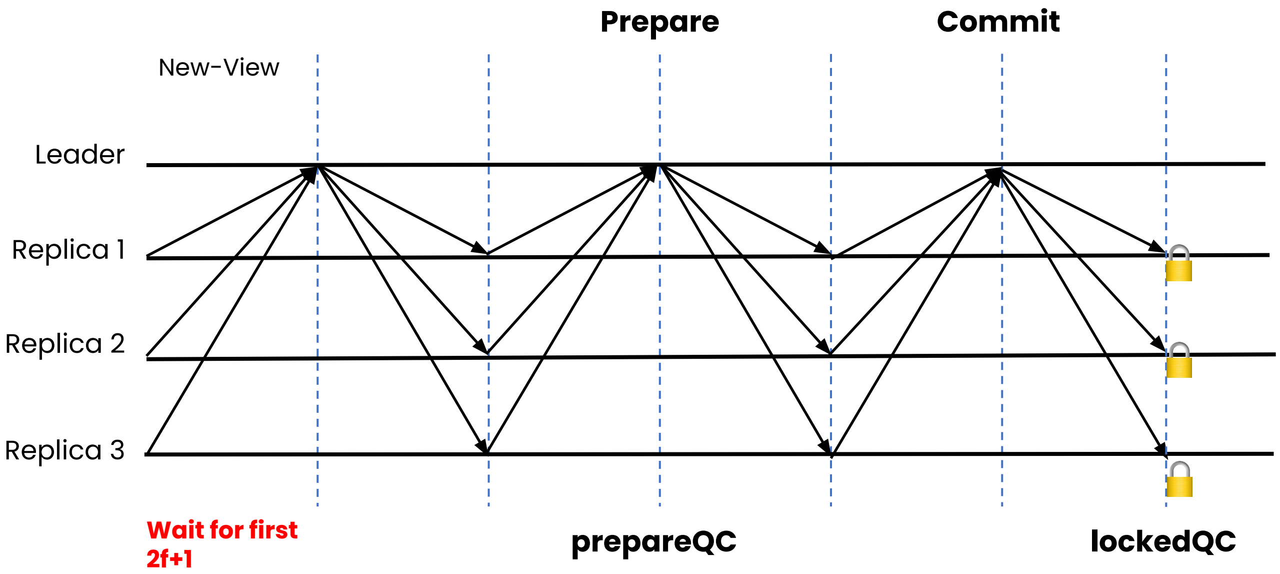
1. Waits for maximum network delay
2. Proposes new block only if 2f+1 same lockedQC received.
3. Proposes same prepareQC otherwise.

Replica

Accepts new proposal if either:

1. 2f+1 same lockedQC threshold sign is true, and, proposed.node extends from local.lockedQC.node
2. Have the same prepareQC as local prepareQC.

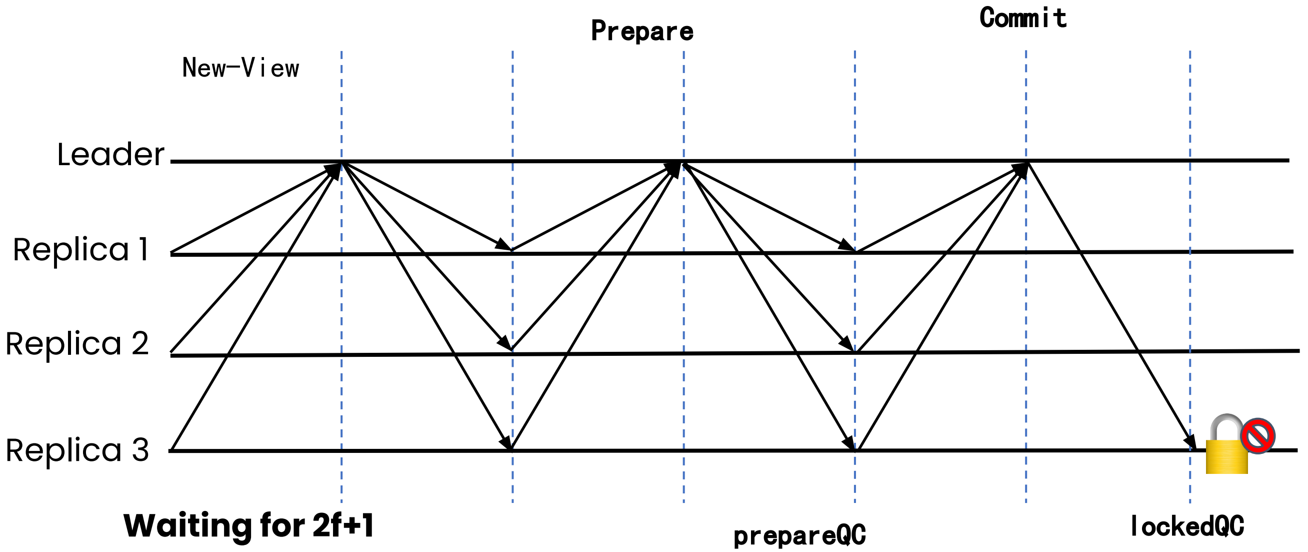
2-Phase HotStuff with optimistic responsiveness

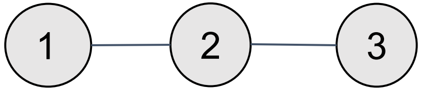


To optimise,

1. The primary only waits for first 2f+1 messages.
2. Replica accepts new proposal if either:
   a. *proposed.node extends from local.lockedQC.node*
   b. *proposed.LockedQC.viewNumber > local.LockedQC.viewNumber*

Liveness Issue

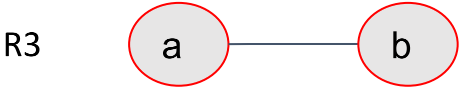




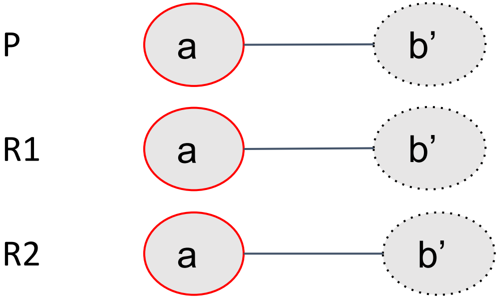
New nodes can only extended from previously locked nodes.

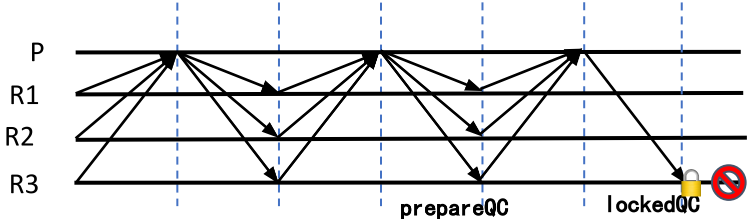
#Safety

Liveness Issue



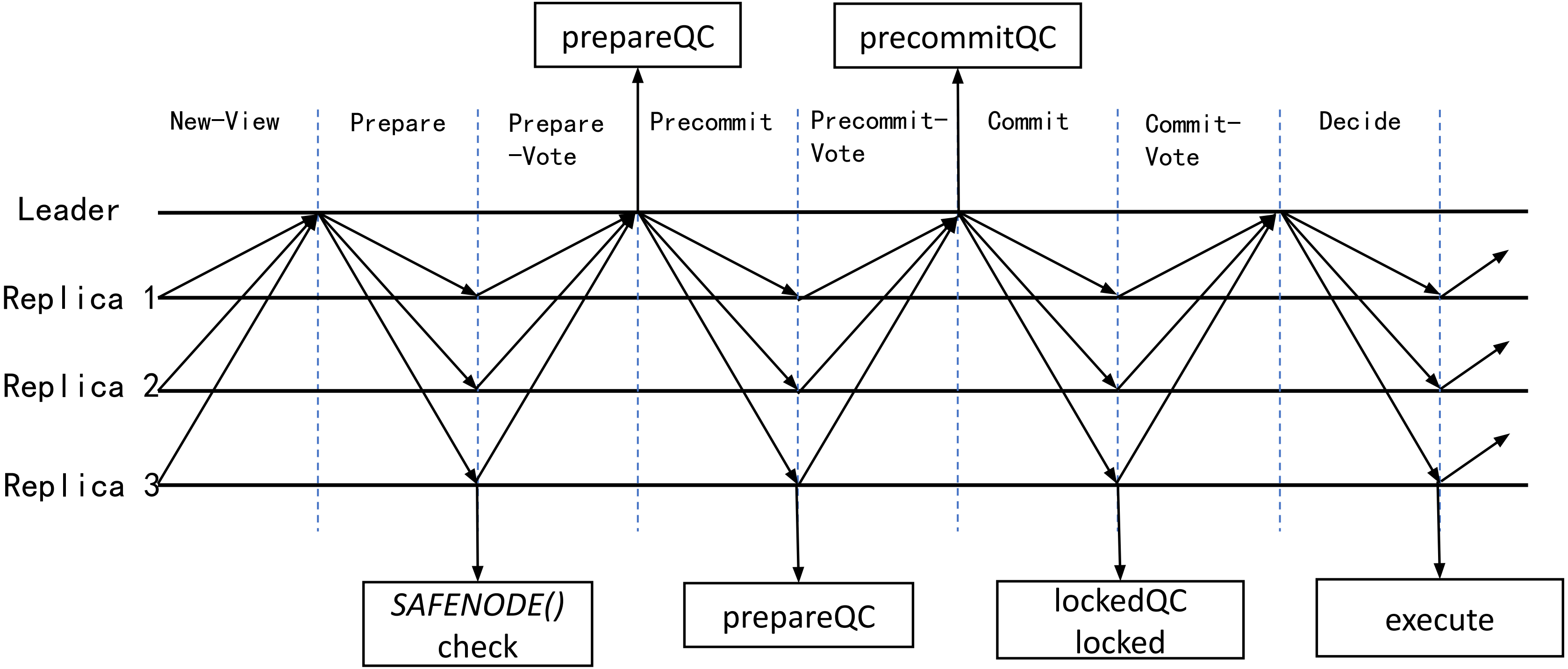
R3 Stays offline for a while, 3f nodes continue.



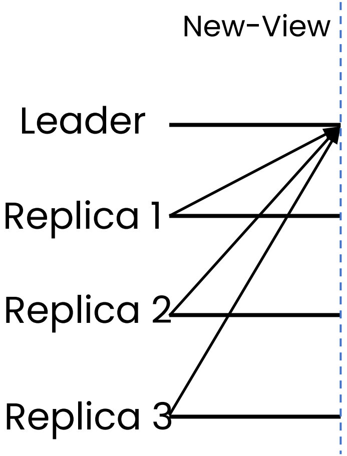


1. 3f nodes commit on b’.
2. f nodes drop off.
3. 2f locked on b’, 1 locked on b.
4. It’s a deadlock!

Basic HotStuff 3-Phase Process



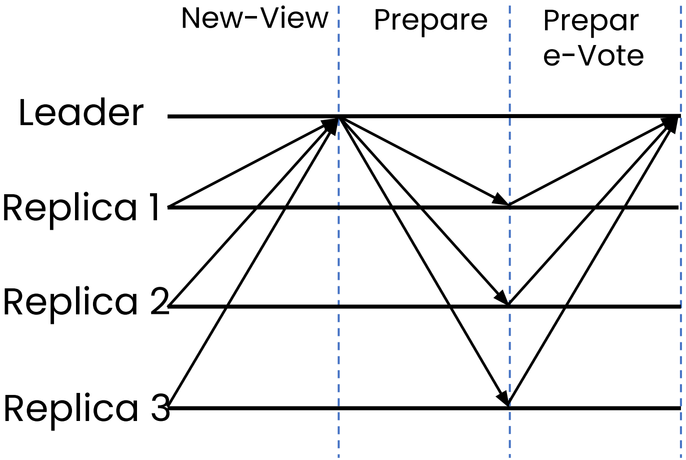
New View

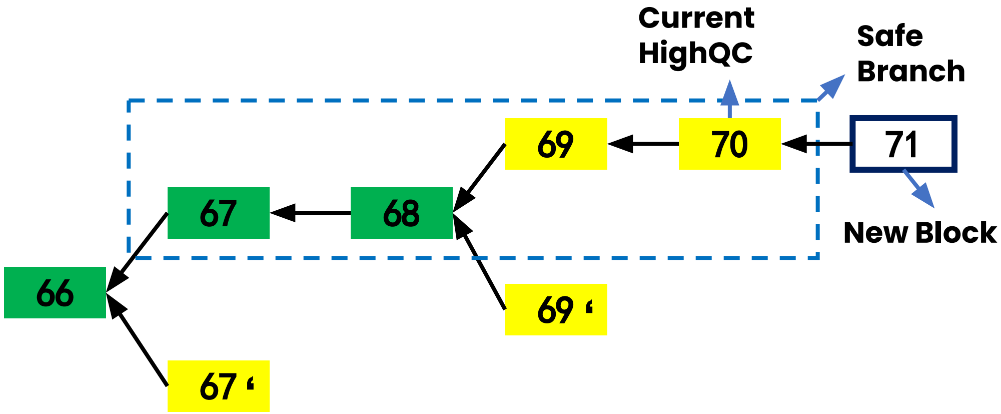


Replica:

- NEXTVIEW interrupt (after global timeout in any phase)
- Send *MSG(NEW-VIEW, ⊥, prepareQC)* to Leader(*curView* + 1)

Prepare Phase





Leader:

- Wait for (n - f) *NewView* messages
- Choose *prepareQC* with the highest *viewNumber* as *highQC*
- Create *leaf* on the node with *highQC*
- Broadcast *MSG(PREPARE, curProposal, highQC)*

Replica:

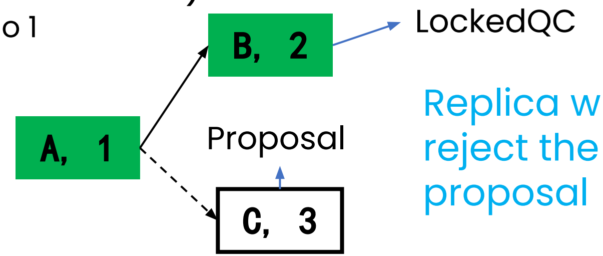
- Wait for *MSG(PREPARE, curProposal, highQC)* from Leader
- Do *SAFENODE* check
- Send *VOTE_MSG(PREPARE, m.node, ⊥)* to Leader

Prepare Phase - Replica

SAFENODE check rules (true if **either** of two rules holds):

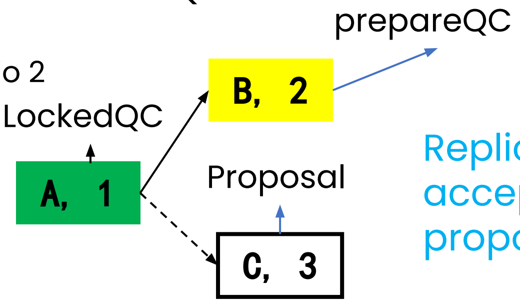
- the branch of m.node extends from the currently locked node (Safety rule)
- m.justify has a higher *viewNumber* than the current lockedQC (Liveness rule)

Scenario 1



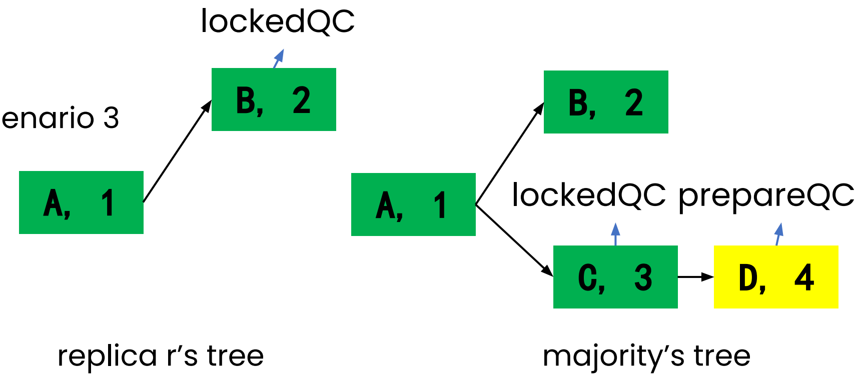
Replica will reject the proposal

Scenario 2



Replica will accept the proposal

Scenario 3

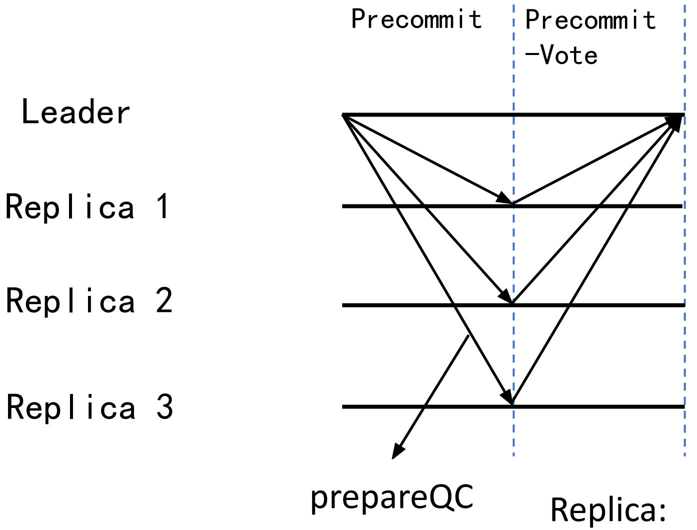


Replica will accept the prepareQC and be unlocked

Why?

Majority accepted the proposal, which means they did not receive the lockedQC and thus there was no commit on view 2

Precommit Phase



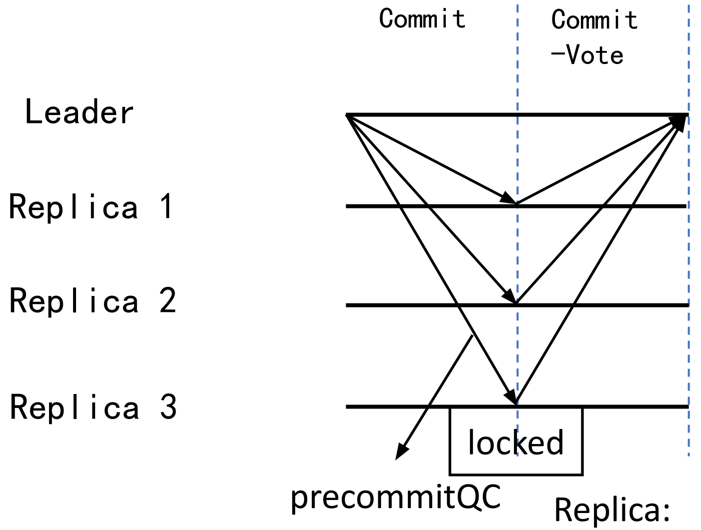
Leader:

- Wait for (n - f) *PREPARE* votes
- Combine votes to *prepareQC*
- Broadcast *MSG(PRE-COMMIT, ⊥, prepareQC)*

Replica:

- Wait for *MATCHING_QC(M.JUSTIFY, PREPARE, curView)* from Leader
- Assign *m.justify* to local *prepareQC*
- Send *VOTE_MSG(PRE-COMMIT, m.justify.node, ⊥)* to Leader

Commit Phase



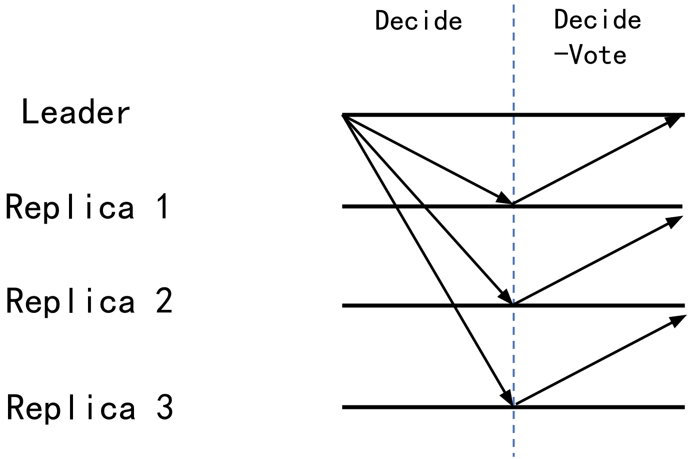
Leader:

- Wait for (n - f) *PRE-COMMIT* votes
- Combine votes to *precommitQC*
- Broadcast *MSG(COMMIT, ⊥, precommitQC)*

Replica:

- Wait for *MATCHING_QC(M.JUSTIFY, PRE-COMMIT, curView)* from Leader
- Assign *m.justify* to local *lockedQC*
- Send *VOTE_MSG(COMMIT, m.justify.node, ⊥)* to Leader

Decide Phase



Leader:

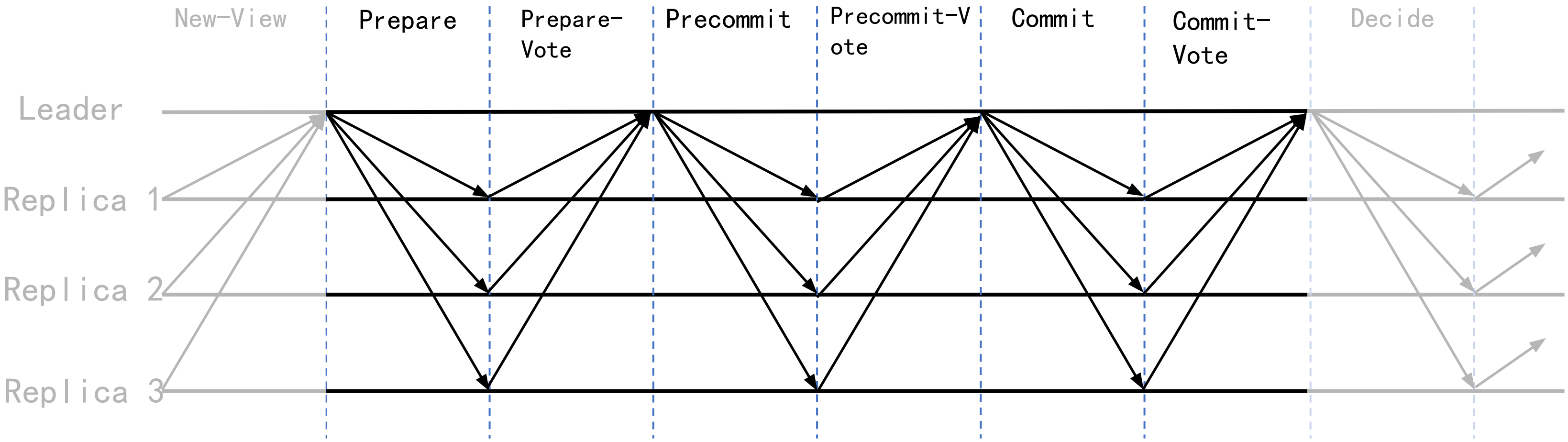
- Wait for (n - f) *COMMIT* votes
- Combine votes to *commitQC*
- Broadcast *MSG(DECIDE, ⊥, commitQC)*

Replica:

- Wait for *MATCHING_QC(M.JUSTIFY, COMMIT, curView)* from Leader
- Execute the command
- Respond to the client

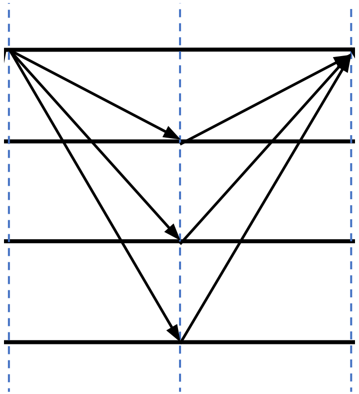
Chained HotStuff

- How we made this flow diagram



Chained HotStuff

- How we made this flow diagram



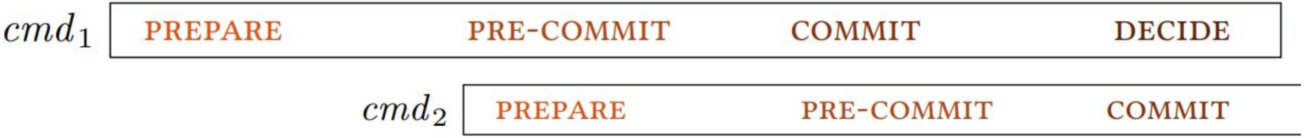
- All the phases do the the same computation - broadcast messages, replicas partial sign, leader aggregates them. Only the underlying data is different.
- When the state machine is in a phase ‘x’, all other phases are idle
- Can we take advantage of this, and make all phases do useful work?

Chained HotStuff

- Generalizing the phases prepare, pre-commit, commit into a “general” message
- How does this effect QC for the phases?

Say we’re here

Prepare phase can start working on cmd2

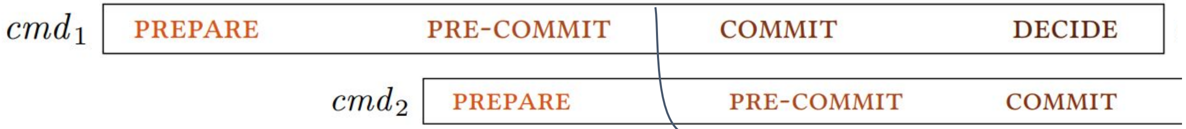


- Read this as “when cmd1 is in pre-commit, the system starts processing prepare for cmd2”
- Assume cmd1 starts with view ‘v’, cmd 2 starts with view ‘v+1’

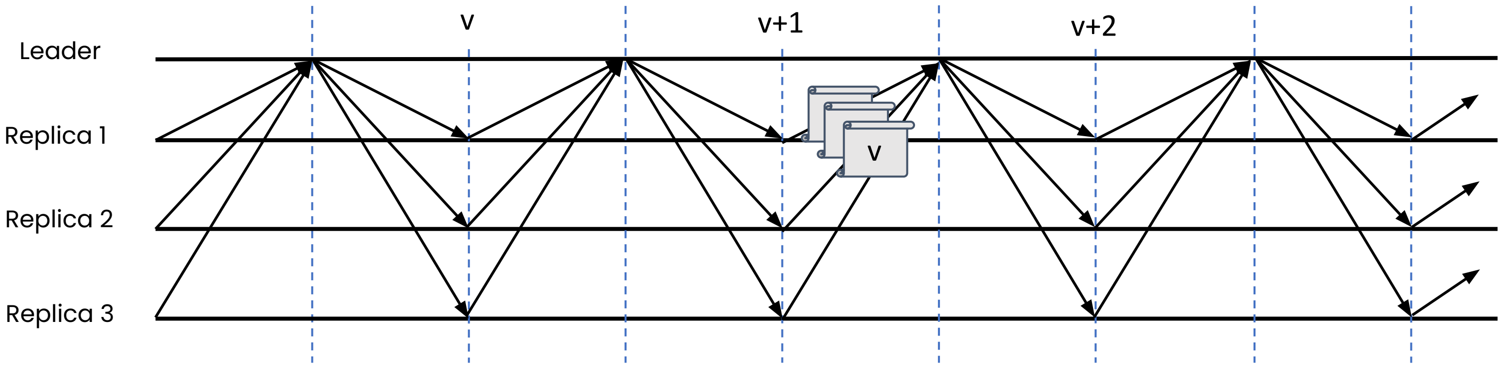
Chained HotStuff

Say we're here

Prepare phase can start working on cmd2



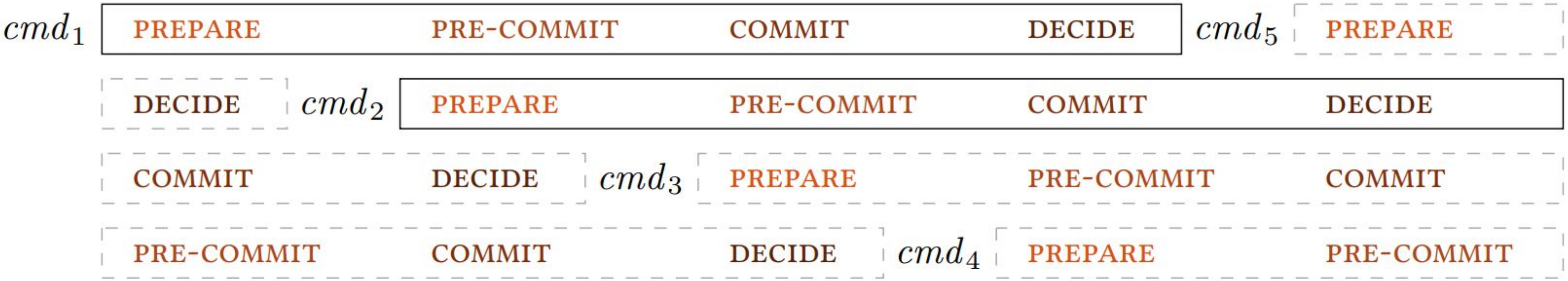
Consider the system at this state
This message has view number 'v',
originally generated from prepare of cmd1



General message with view v in phase v+1 implies precommit phase for phase v, prepare phase for v+1

Chained HotStuff

- Other commands are processed simultaneously
- System is ‘pipelined’
- Increased throughput, as previously, latency of one command was 3 phases, now it’s just one



cmd_1 PREPARE PRE-COMMIT COMMIT DECIDE cmd_5 PREPARE

DECIDE cmd_2 PREPARE PRE-COMMIT COMMIT DECIDE

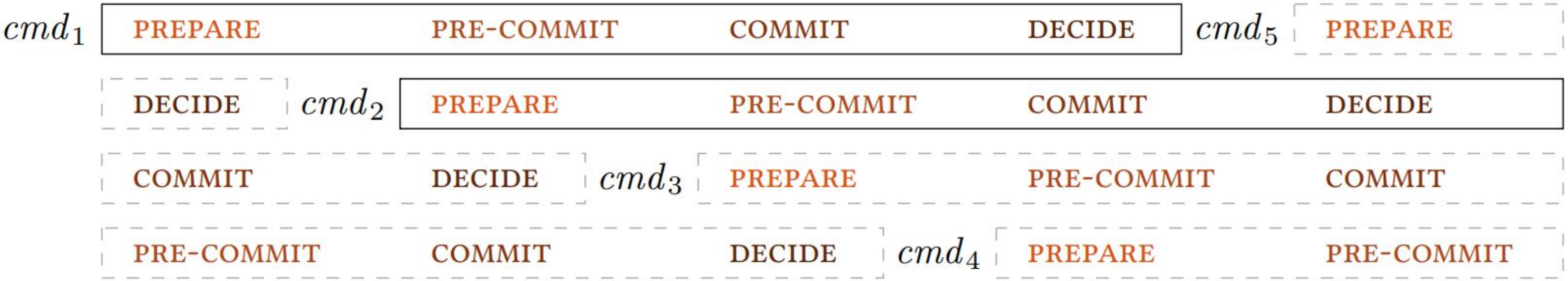
COMMIT DECIDE cmd_3 PREPARE PRE-COMMIT COMMIT

PRE-COMMIT COMMIT DECIDE cmd_4 PREPARE PRE-COMMIT

- Back to the question, how does this effect QC?
- What if QC wasn’t reached?
- Can the pipeline stall?
- How is proof (justify) for a command to be committed prepared, when the previous command didn’t have QC?

Chained HotStuff

- Other commands are processed simultaneously
- System is ‘pipelined’
- Increased throughput, as previously, latency of one command was 3 phases, now it’s just one



cmd_1 PREPARE PRE-COMMIT COMMIT DECIDE cmd_5 PREPARE

DECIDE cmd_2 PREPARE PRE-COMMIT COMMIT DECIDE

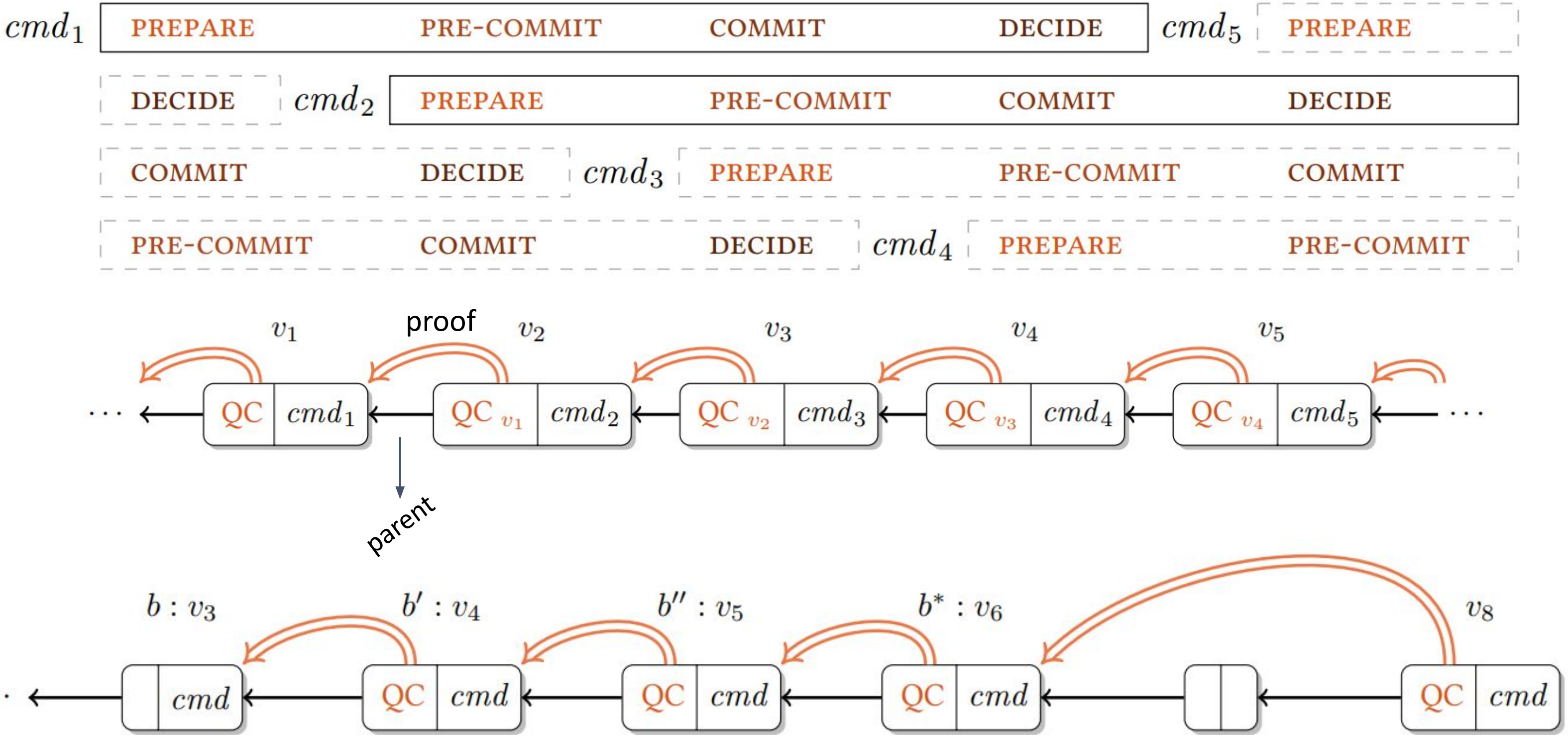
COMMIT DECIDE cmd_3 PREPARE PRE-COMMIT COMMIT

PRE-COMMIT COMMIT DECIDE cmd_4 PREPARE PRE-COMMIT

- Back to the question, how does this effect QC?
- What if QC wasn’t reached?
- Can the pipeline stall?
- How is proof (justify) for a command to be committed prepared, when the previous command didn’t have QC?

Chained HotStuff

- Each node uses the previous node's QC as justify (proof)



cmd_1 PREPARE PRE-COMMIT COMMIT DECIDE
cmd_5 PREPARE
DECIDE
cmd_2 PREPARE PRE-COMMIT COMMIT DECIDE
COMMIT DECIDE
cmd_3 PREPARE PRE-COMMIT COMMIT
PRE-COMMIT COMMIT DECIDE
cmd_4 PREPARE PRE-COMMIT
v_1
proof
v_2
v_3
v_4
v_5
QC cmd_1
QC $_{v_1}$ cmd_2
QC $_{v_2}$ cmd_3
QC $_{v_3}$ cmd_4
QC $_{v_4}$ cmd_5
parent
$b : v_3$
$b' : v_4$
$b'' : v_5$
$b^* : v_6$
v_8
cmd
QC cmd
QC cmd
QC cmd
QC cmd