Cristina Nita-Rotaru

7680: Distributed Systems

Quorums. Paxos. Viewstamped replication. BFT. Raft

Required reading for this topic…

- Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. F. B. Schneider
- Quorums
  - Quorum Systems. Chapter in The Encyclopedia of Distributed Computing, D. Malkhi
- Paxos
  - Paxos Made Simple, L. Lamport
  - **Paxos for System Builders, J. Kirsch and Y. Amir (the technical report) – need for project**
  - The Part-time Parliament, L. Lamport
- Viewstamped Replication Revisited, B. Liskov and J. Cowling
- From Viewstamped replication to Byzantine replication. B Liskov.

1: The State Machine Approach

The State Machine Approach

- The system consists of clients that invoke commands on deterministic state machines
- The state of a state machine depends only on its initial state and the sequence of (deterministic) commands it has been given
- All non-faulty state machines, being deterministic, will give the same response to a command

How it works …

- All replicas start in the same initial state
- Every replica apply operations in the same order
- All operations must be deterministic
- All replicas end up in the same state

Consensus

- Each process sends its value (proposal) to all the other processes, all processes have the same set they can make a decision

- Each process sends its value (proposal) to a leader which makes the decision and informs the other processes

Challenges

- Can nodes trust each other
- Can a node crash and/or recover
- Can a network partition occur
- Can messages be delayed or lost
- How to detect that
  - processes crashed
  - network partitions
  - messages are lost

Crash, Recovery, and Network Partition

- Process crashes: the leader may not have enough values to reach a decision
- Network partition: leader may not have enough values, some components will have no leaders and have to select one
- Leader crashes: if it crashes after deciding but before announcing results everybody is blocked
- Process or leader recovers: they will not know what they were doing (unless they write some information on the disk)

Process Groups Approach

- One way of building distributed fault-tolerant systems by organizing them in a group:
  - Ensure group membership
  - Ensure group multicast, with different ordering properties.
- Easier to work with when providing in the form of a toolkit

Limitations of Process Groups Approach

- Need to respond to leader failure
  - Costly agreement on membership
- Virtual synchrony: simplify recovery from partitioned views
- Servers need to monitor for failures (correct but slow participants may be excluded)
- Reconfiguration

2: Quorums

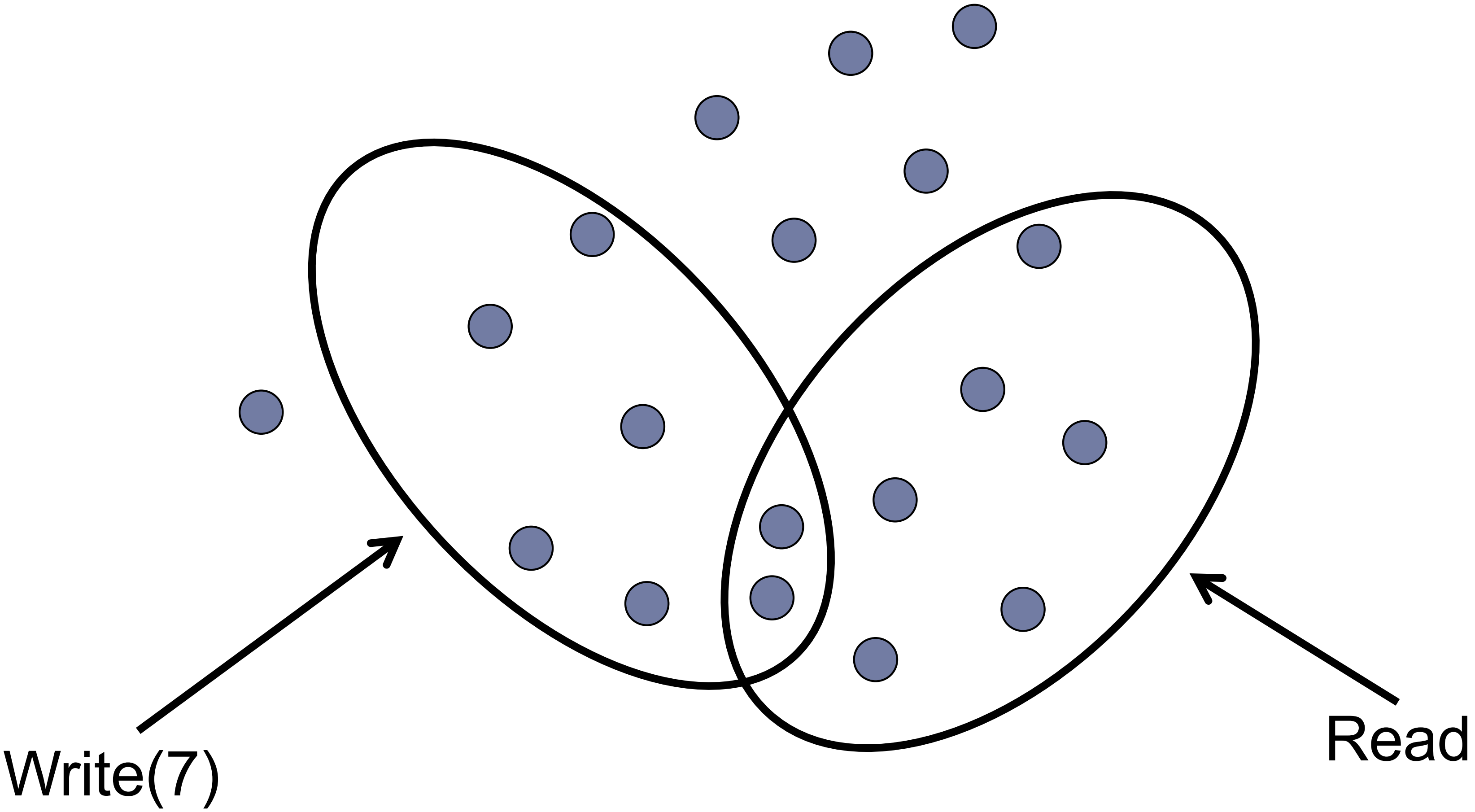
A different approach: Quorum Systems

- In law, a quorum is the minimum number of members of a deliberative body necessary to conduct the business of that group
- When quorum is not met, a legislative body cannot hold a vote, and cannot change the status quo

Quorum Systems

- Increase availability and efficiency of replicated services
- **Availability**: Operations succeed in spite of failures; quorum systems can be defined to tolerate both benign and arbitrary/malicious failures
- **Efficiency**: Can significantly reduce communication complexity, do not require all servers in order to perform an operation, requires a subset of them for each operation

Using Quorums to Read and Write



The set of processors from which a variable is *read* must intersect the set of processors to which a variable was *written*.

Shared Variable with Quorums

- Use a quorum system to implement a multi-reader multi-writer shared variable, replicated across n servers

Write:

- Client queries each server in some quorum (writing quorum) to obtain a set A of value/timestamp pairs
- Client chooses a timestamp greater than the highest value in the set A and updates the value and the timestamp at each server in the writing quorum

Shared Variable with Quorums (II)

Read:

- Client queries each server in a quorum to obtain a set A of value/timestamp
- Then chooses the pair with the highest timestamp

- For both read and write each server updates its local variable and timestamp to the received values, only if received timestamp is greater than the one they had for that value

Replication with Quorums

- Replicated data items have “versions”, and these are numbered
  - I.e. can’t just say “Xp=3”. Instead say that Xp has timestamp [7,q] and value 3
  - Timestamp must increase monotonically and includes a process id to break ties

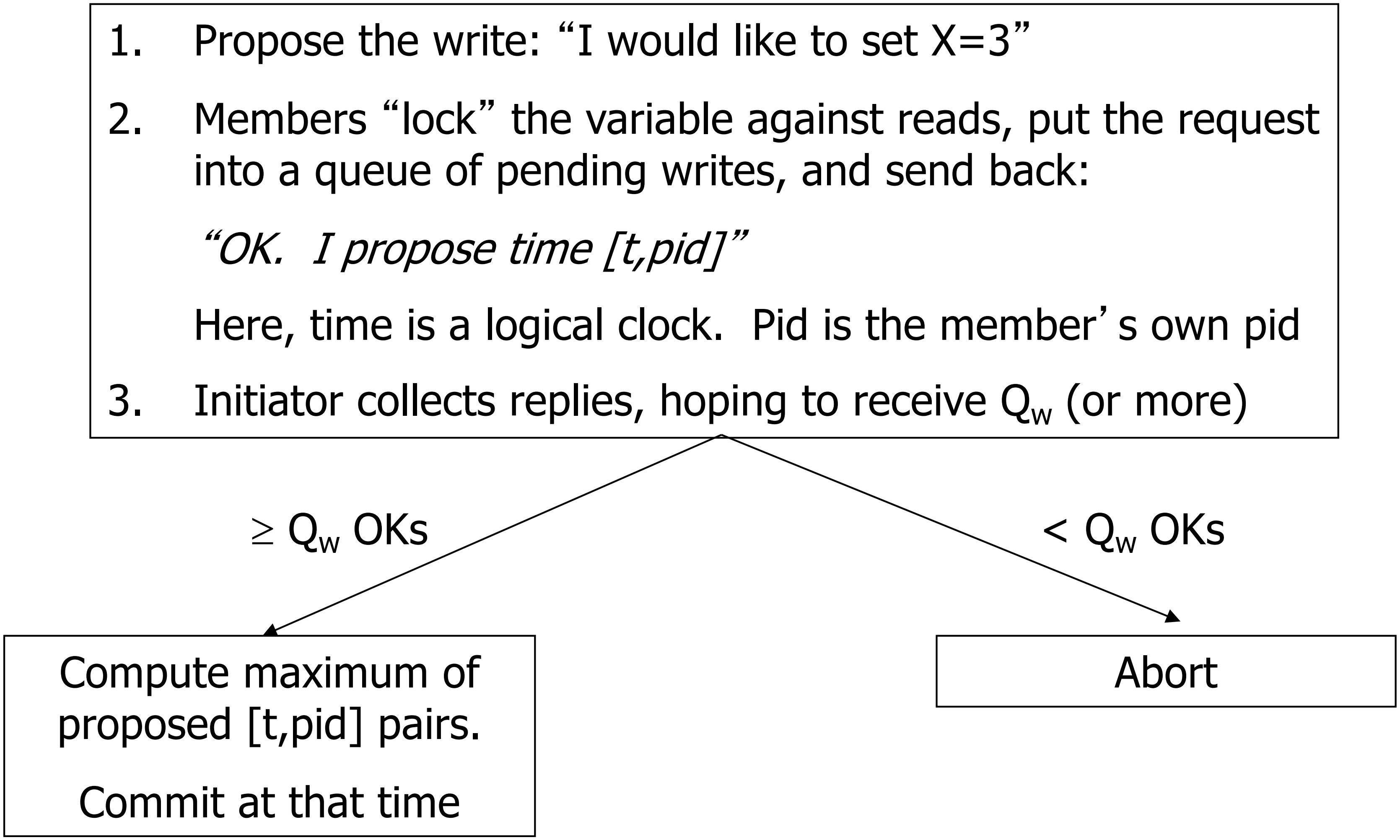
Read Operation

- Send request and wait until Qr (read quorum) processes reply
- Then use the value with the largest timestamp
  - Break ties by looking at the process id
  - For example
    - [6,x] < [9,a] (first look at the “time”)
    - [7,p] < [7,q] (but use process id as a tie-breaker)

Write Operation

- When a process initiates a write, it does not know if it will succeed in updating a quorum (writing quorum) of processes
  - Need to use a commit protocol
- Moreover, must implement a mechanism to determine the version number as part of the protocol.

Write Operation: Details



1. Propose the write: “I would like to set X=3”
2. Members “lock” the variable against reads, put the request into a queue of pending writes, and send back:

   “OK. I propose time [t,pid]”

   Here, time is a logical clock. Pid is the member’s own pid
3. Initiator collects replies, hoping to receive Q_w (or more)

$\geq Q_w$ OKs

Compute maximum of proposed [t,pid] pairs.

Commit at that time

$< Q_w$ OKs

Abort

Quorum constructions: Weighted Majorities

- Assume that every server s in the universe U is assigned a number of votes ws.

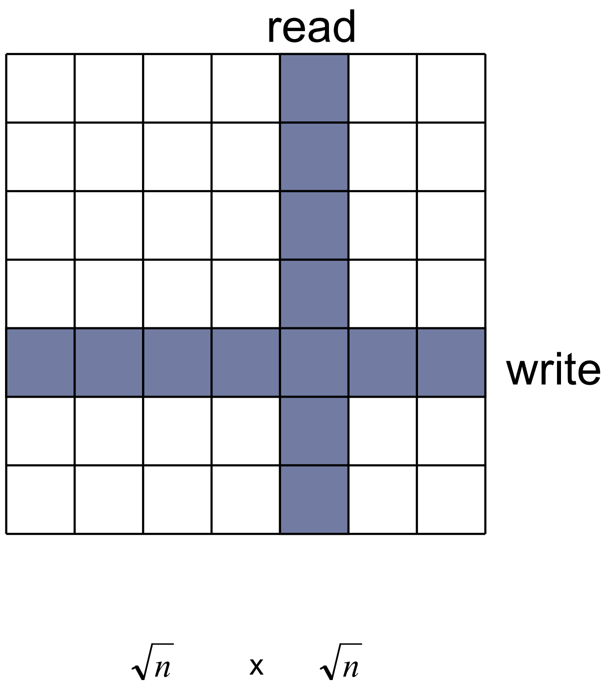
  Then, the set system

  $$\boldsymbol{Q} = \{Q \subseteq U: \Sigma_{q \in Q} w_q > 1/2 \Sigma_{q \in U} w_q\}$$

  is a quorum system called Weighted Majorities.
- When all the weights are the same, simply call this the system of Majorities

Quorum constructions: Grid

- Previous example was not very efficient, requiring more than half of the servers to be contacted
- Arrange servers into a logical grid, and use rows/columns for writes/reads, respectively
- Can cut the number of servers contacted in an operation
- Can change row/column sizes to optimize for write-heavy/read-heavy scenarios



2: Paxos

Paxos

- More robust that process groups approach
- Safety does not require synchronous communication
- Liveness requires that we are able to detect failures within a certain timeout
  - Requires a stable-enough network to elect a leader that will stay stable for a while
  - Requires a (potentially changing) majority of members to support the leader (in order to make progress)

Goals of Paxos

- Provides a solution to consensus (agreement) in an **asynchronous** model
- Safety - If two servers execute the ith update, then these updates are the same.
  - Only a value that has been proposed may be chosen
  - Only a single value is chosen
  - A node never learns that a value has been chosen unless it actually has been
- Liveness
  - Some proposed value is eventually chosen
  - If a value has been chosen, a node can eventually learn the value

The Model

- N servers, uniquely identified
- Messages:
  - **Can take arbitrarily long to be delivered (Asynchronous communication)**
  - Can be duplicated
  - Can be lost
- Nodes:
  - **Operate at arbitrary speed**
  - May fail by stopping, and may restart
  - Must remember what they were doing (in case they fail and restart, they have to know what to do next)

Paxos: Main Idea

- One node is chosen as the leader (we will see later how this is done)
- Leader proposes a value and asks the other nodes to accept it
- Leader announces result to the rest of the nodes or tries again if he could not have reached a decision

- Why this algorithm?

Terminology (from the Lamport paper) (Paxos made simple)

- In any consensus nodes perform three different types of actions:
  - Propose values
  - Accept values
  - Learn values
- We can classify nodes as:
  - Proposer: proposes a value and solicits acceptance from acceptors
  - Acceptor: accepts a value
  - Learner: finds out the outcome
- A node can have all three roles

Need for Multiple Acceptors

- Assume a single node A acts as acceptor
  - Each proposer sends its value to A
  - A decides on one of the values
  - A announces its decision to all learners
- What can go wrong?
  - If the acceptor fails, the protocol will block since nobody will decide
- Solution: We need multiple acceptors

Solution with Multiple Acceptors

- Each proposer proposes to all acceptors
- Each acceptor accepts the first proposal it receives and rejects the rest
- If the proposer receives positive replies from a majority of acceptors, it chooses its own value (that' s what he proposed)
- Proposer sends chosen value to all learners
- What if multiple proposers propose simultaneously so there is no majority accepting?

Dealing with Multiple Proposals

- Proposals are ordered by proposal number
- We can allow multiple proposals but we must guarantee that all chosen proposals have the same value
- Each acceptor may accept multiple proposals
  - If a proposal with value v is chosen, all higher-numbered proposals have value v

Invariant

- For any v and n, if a proposal with value v and number n is issued then there is a set S consisting of a majority of acceptors such that:
  - No acceptor in S has accepted any proposal numbered less than n, or
  - v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S

How to Ensure the Invariant

- Can not predict the future, what acceptor will accept
- Can obtain promise from acceptors with respect to what they will accept:
- “The proposer requests that the acceptors not accept any more proposals numbered less than n”

Issuing Proposals

- A proposer chooses a new proposal number n and sends a request to a set of acceptors, asking them:
  - Promise to never accept a proposal numbered less than n
  - Send the proposal with the highest number less than n that it has accepted, if any.
- If the proposer receives the requested responses from a majority of acceptors it can issue a proposal with number n and value v, where v is:
  - the value of the highest-numbered proposal among the responses, or
  - any value selected by the proposer if the responders reported no proposals.

Optimization

- Given that an acceptor receives a request numbered n, but it has already responded to a request numbered greater than n, thereby promising not to accept any new proposal numbered n.
- Acceptor can ignore
  - such a request
  - a request for a proposal it has already accepted

Accepting Proposals

- Phase 1 (Prepare)
  - A proposer selects a proposal number n and sends a Prepare_Request<n> to a majority of acceptors.
  - If an acceptor receives a Prepare_Request<n> with n greater than that of any Prepare_Request to which it has already responded, then it responds to the request with an ACK which promises not to accept any more proposals numbered less than n and includes the highest-numbered proposal (if any) that it has accepted.
- Phase 2 (Accept)
  - If the proposer receives an ACK to its Prepare_Request<n> from a majority of acceptors, then it sends an Accept_Request<, n, v> to each of those acceptors, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.
  - If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

Learning about Accepted Proposals

- Lower cost by using a leader for learners
- Acceptors send their accepts to the leader for the learners
- It is possible that a value has been accepted and some learners did not learn it
- They will learn it when a new proposal is issued

Need for one Leader for Proposers

- Scenario where there will be no progress with two proposers
- Proposer p completes phase 1 for a proposal number n1
- Proposer q then completes phase 1 for a proposal number n2 > n1
- Proposer p’s phase 2 accept requests for a proposal numbered n1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n2
- Proposer p then begins and completes phase 1 for a new proposal number n3 > n2, causing the second phase 2 accept requests of proposer q to be ignored

Leader

- One leader, it' s the leader for the proposers and for the learners: issues proposals and informs all learners of the outcome
- How to select leader: FLP implies that to select a leader we need to use timeouts or randomization
- A new leader must not violate previously established ordering!
  - The new leader must know about all updates that may have been ordered.
  - If a new leader gets information from any majority of acceptors, it can determine what may have been ordered!

Implementation: Node states

- Acceptor:
  - na, va: highest accepted proposal number and its corresponding accepted value
  - np: highest proposal number seen
- Proposer:
  - myn: the current proposal number

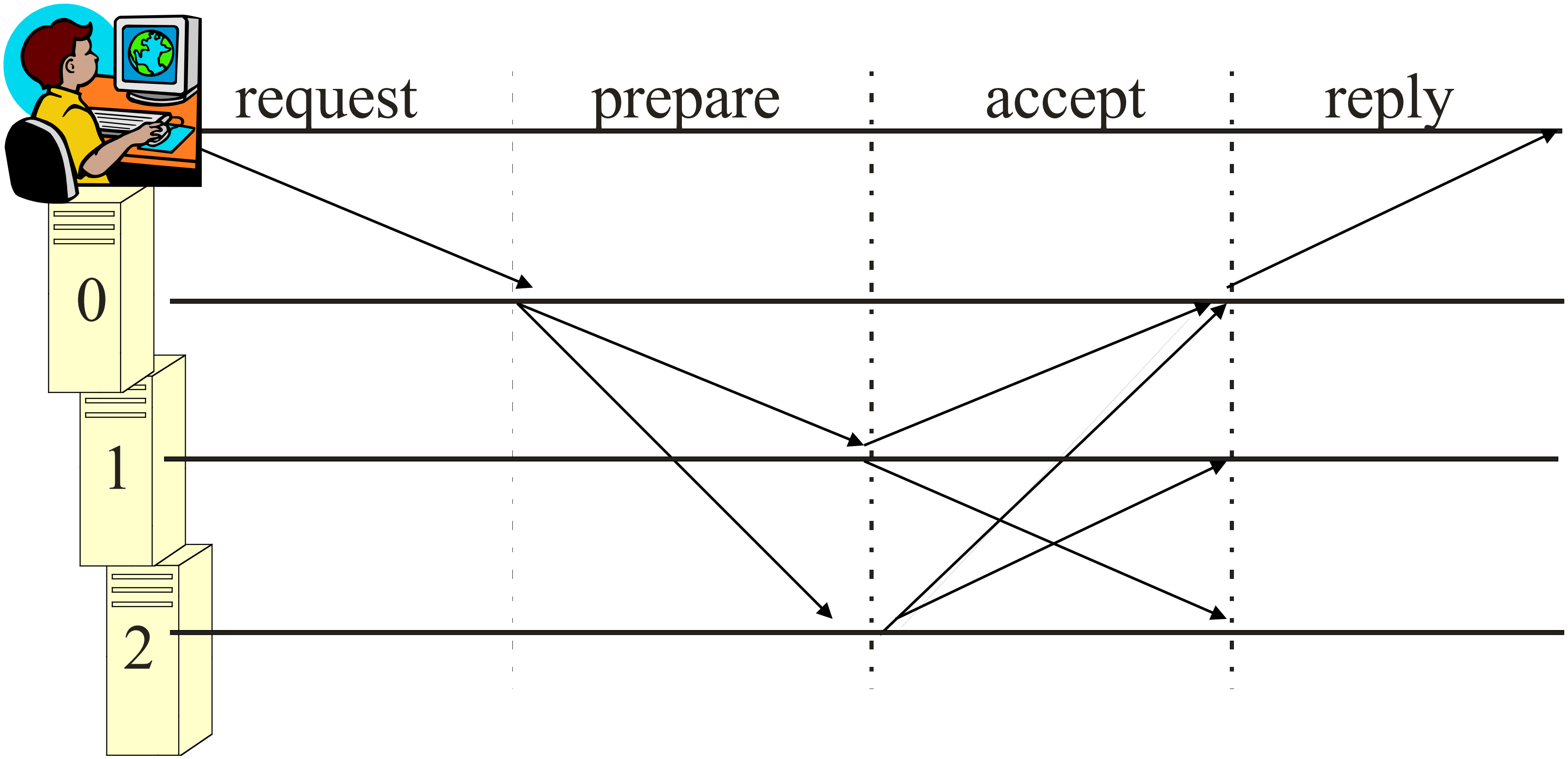
Proposer Algorithm for Value v

```
select $my_n > n_p$
send PREPARE($my_n$) to all nodes


if received PREPARE_OK($n_a$, $v_a$) from majority then
   $v_a$  = $v_a$ with highest $n_a$, or choose own v
otherwise
   send ACCEPT ($n_a$, $v_a$) to all

if received ACCEPT_OK($n_a$) from majority then
     send DECIDED($v_a$) to all
```

Acceptor Algorithm

```
If received PREPARE(n)
 If n > np
   $n_p$= n
     send PREPARE_OK ($n_a$,$v_a$)

If received ACCEPT(n, v)
 If n >= $n_p$
   $n_a$ = n
   $v_a$ = v
     send ACCEPT_OK
```

Optimization



f servers can crash, f = 1 in this example

Recovery Case

- If leader fails, new leader is elected, known as *view change*
- The new leader cannot propose updates until it collects information from a majority of servers!
  - New leader must learn the outcome of all pending requests
  - For all pending requests, the leader sends accept messages
  - New leader sends a sequence n
  - Every nodes sends a higher ni representing proposals it has ordered or ackd
  - Leader collects f+1 responses, eliminates duplicates, selects proposals with highest number and broadcasts it to everybody, computes also the list of missed messages

Questions

- What if more than one leader is active and issues two different proposal numbers, can both leaders see a majority of Prepare_Ok?
- What if leader fails while sending accept?
- What if a node fails after receiving Accept?
- What if a node fails after sending Prepare_Ok?

Liveness

- A leader can get conflicting Proposal messages!
- Why? Some nodes might think somebody else is the leader
- How to fix? Add view numbers, each proposal is made within a view, view number is attached to each proposal
- Leader selects the proposal with the highest view id

View Change

- A set of 2f+1 replicas, replica id: 0,1,…,2f
- Each view: one and only one leader
- Initially replica 0 assumes the leader role for view=0
- Subsequently, replicas take the primary role in a round-robin fashion
- To ensure liveness
  - A replica starts a view change timer on the initiation of each instance of Paxos
  - If the replica does not learn the request chosen before timer expires => suspect the primary

View Change: Start

- On suspecting the leader, a replica broadcasts a **View Change** message to all
- Current leader, if it is wrongly suspected, joins the view change anyway (i.e., it steps down from leader role)
- A replica joins the view change even if it's view change timer has not expired yet
- On joining view change, a replica stops accepting normal control msgs and respond to only checkpointing and view change msgs

View Change: Installing a new view

- New view # Seq# of last stable checkpoint
- A set of accepted records since last stable checkpoint
- Each record consists of view#, seq#, request msg
- On receiving f+1 View Change msgs, new leader sends **New View** msg
- Include a set of accept msgs
- Include all accepted msgs as part of View Change msg
- When a gap in seq# is detected, create an accept request with no op
- A replica accepts new view msg if it has not installed a newer view

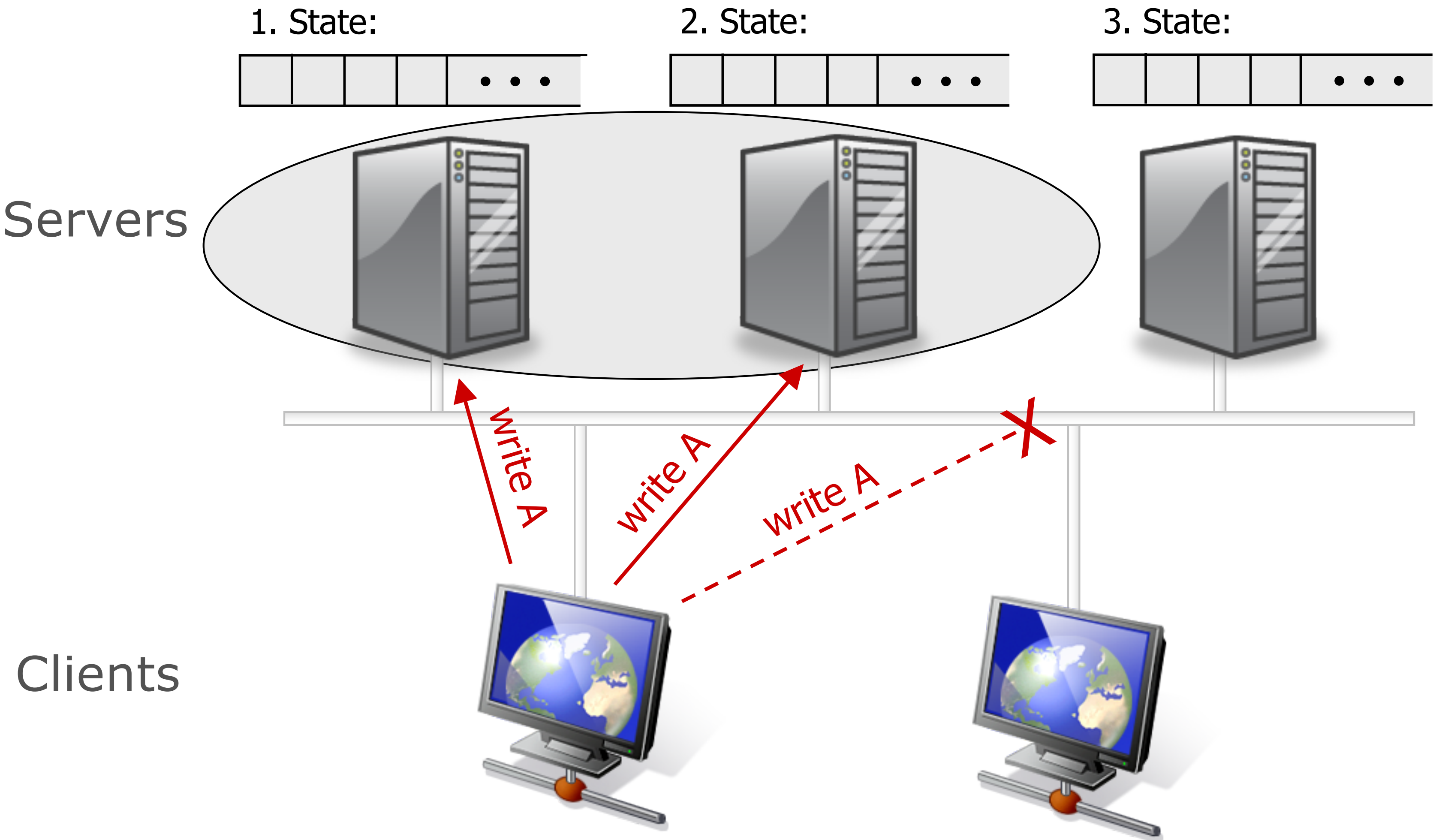
3: Viewstamped replication

Slides by B. Liskov

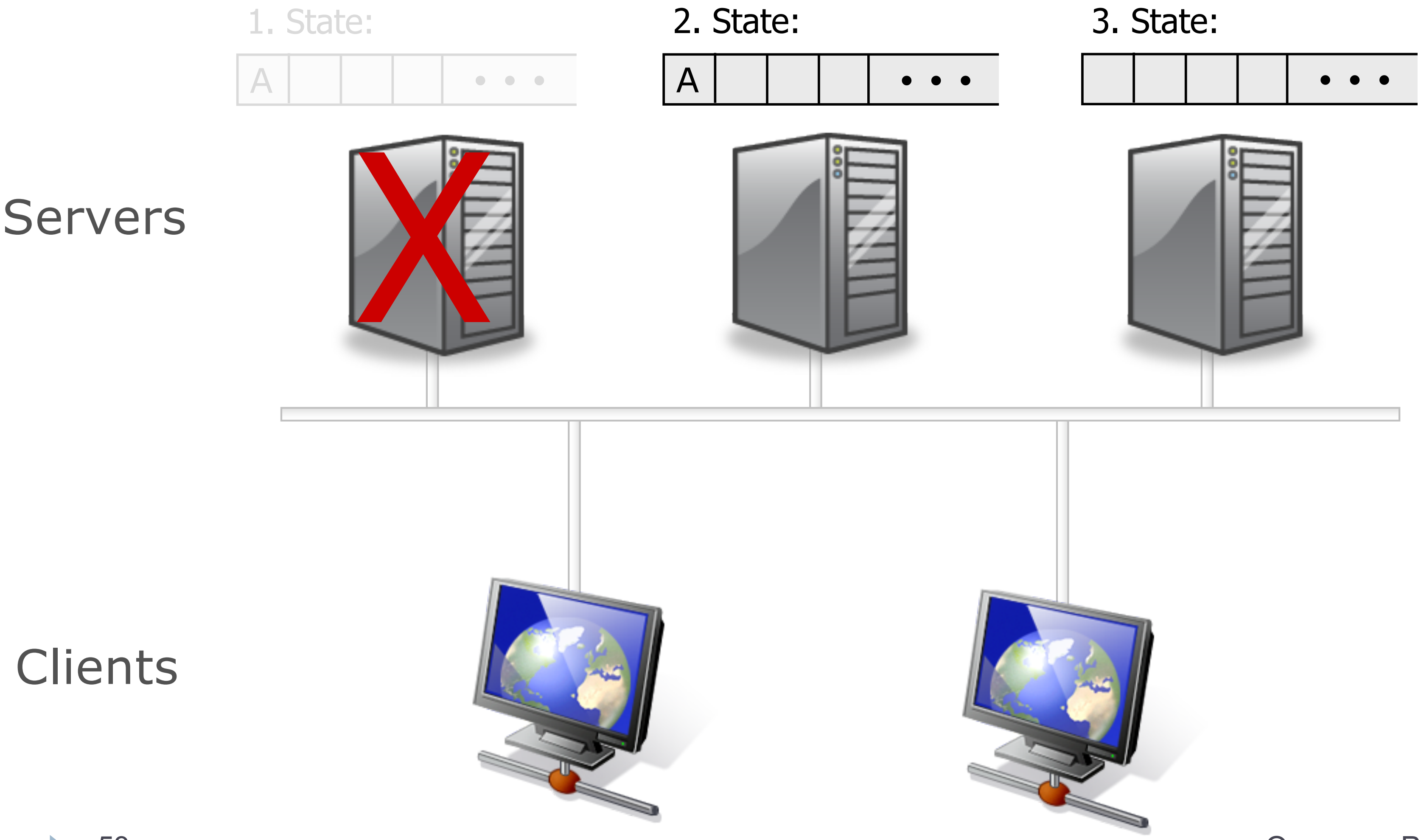
Viewstamped replication

- Problem it solves: replication protocol
- Model:
  - Failstop failures
  - Asynchronous communication
- Uses quorums and ideas from 2PC
  - 2f+1 replicas to tolerate f failures
  - Operations must intersect at at least one replica
  - Availability for both reads and writes
  - Read and write quorums of f+1 nodes
- Appeared in PODC 1988, SOSP 1991, independent from Paxos.

Quorums


1. State:
2. State:
3. State:
• • •
Servers
Clients
write A
write A
write A
X

Quorums



Servers

1. State:

A • • •

2. State:

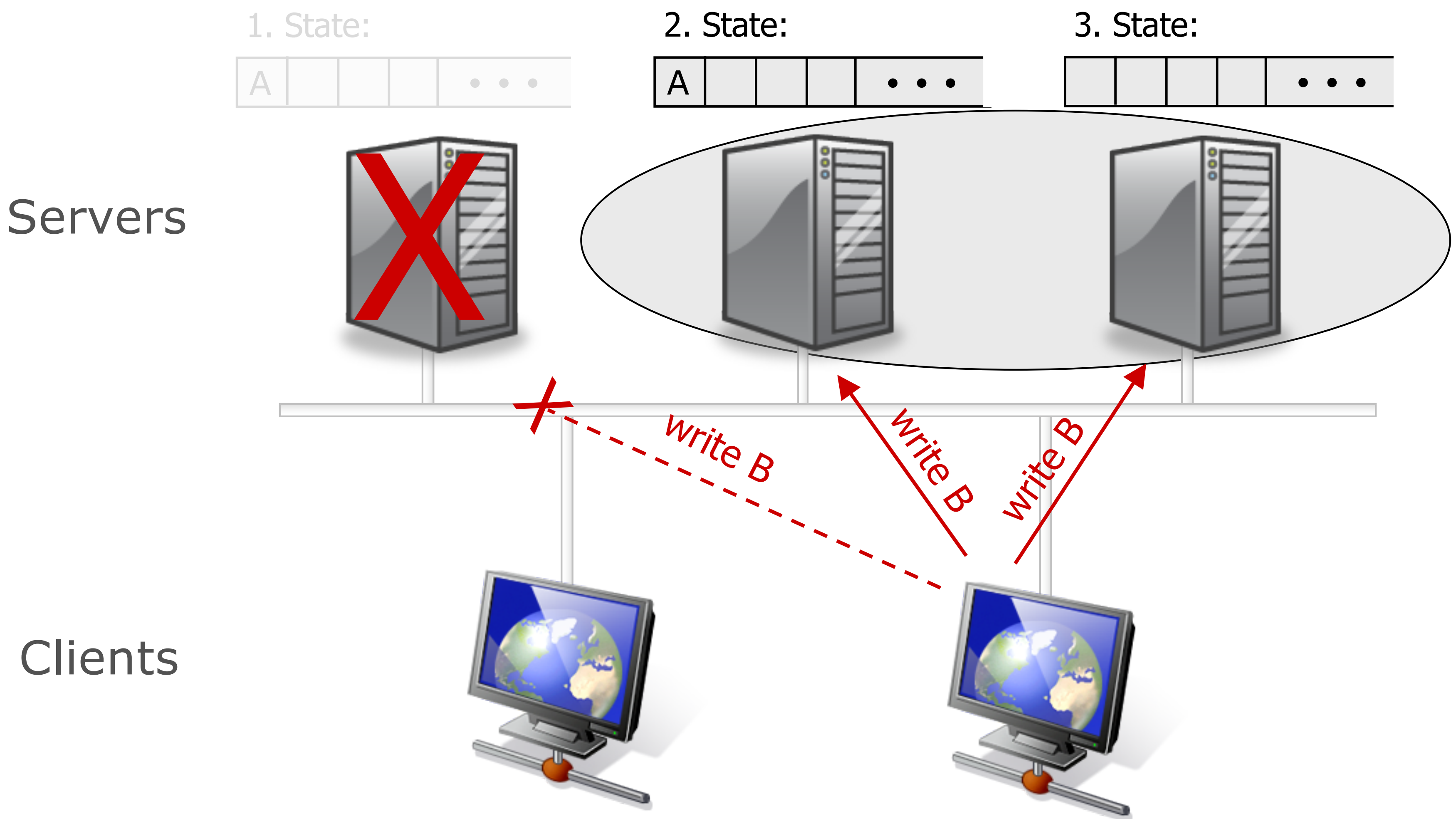
A • • •

3. State:

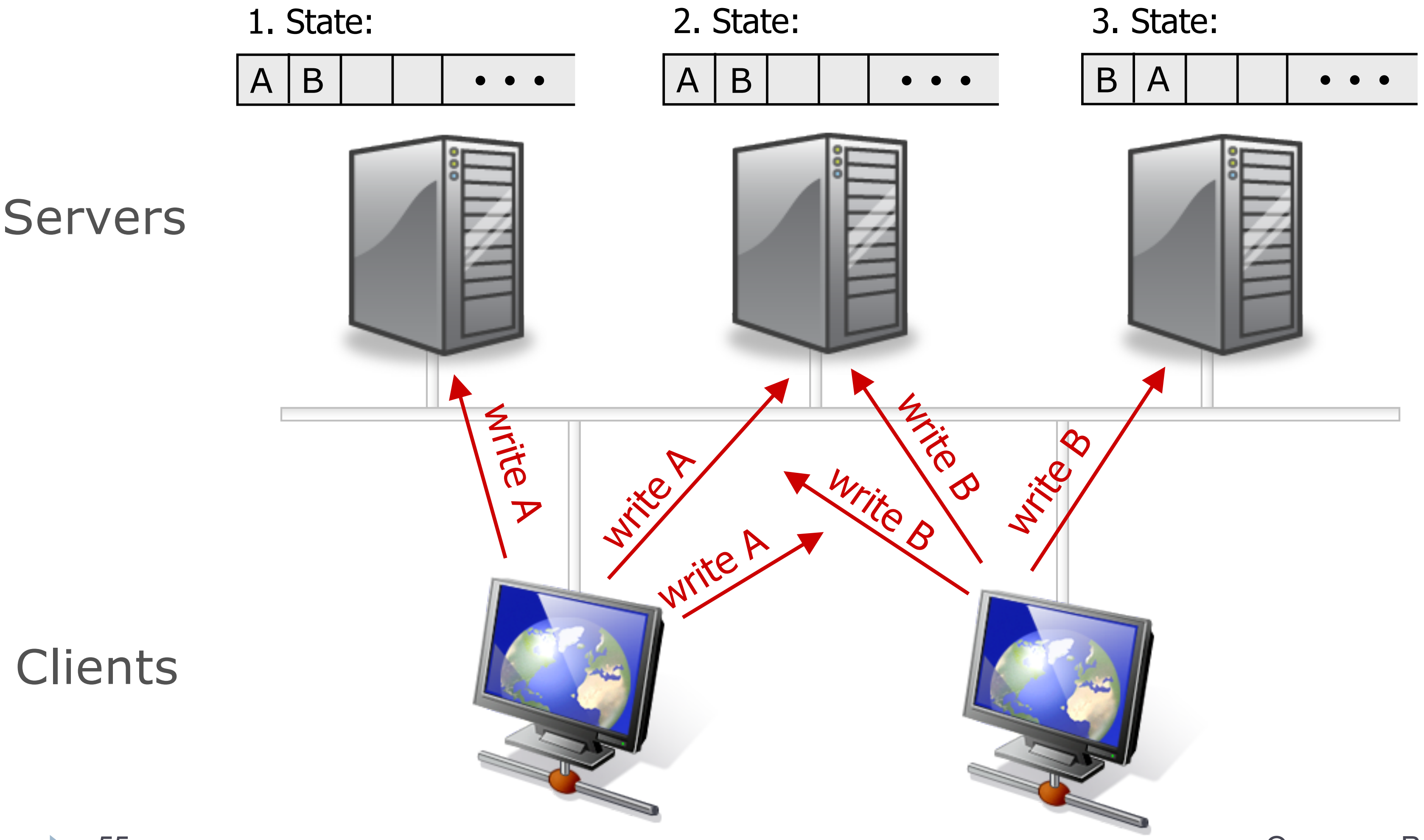
• • •

Clients

Quorums


1. State:
A
• • •
2. State:
A
• • •
3. State:
• • •
Servers
write B
write B
write B
Clients

Concurrent Operations



Overview

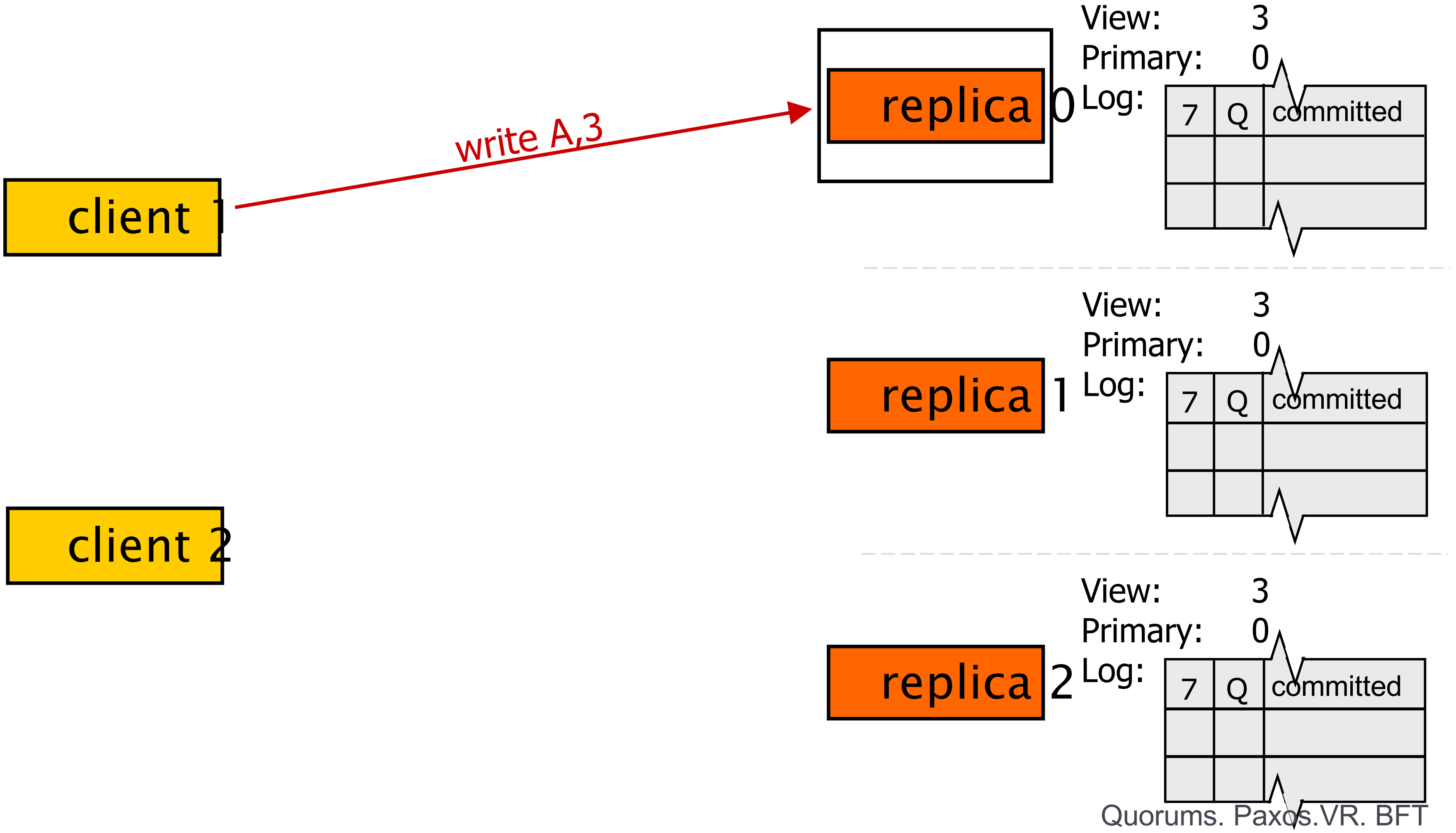
- Replicas must execute operations in the same order
- Implies replicas will have the same state, assuming
  - replicas start in the same state
  - operations are deterministic
- Uses a primary to order operations
- Uses views to address primary failures, system moves through a sequence of views
  - Primary runs the protocol
  - Replicas watch the primary and do a view change if it fails

Replica State

- A replica id i (between 0 and N-1)
  - Replica 0, replica 1, …
- A view number v#, initially 0
- Primary is the replica with id

  i = v# mod N
- A log of <op, op#, status> entries
  - Status = prepared or committed

Normal Case



client 1

client 2

write A,3

replica 0

View: 3
Primary: 0
Log:

7	Q	committed

replica 1

View: 3
Primary: 0
Log:

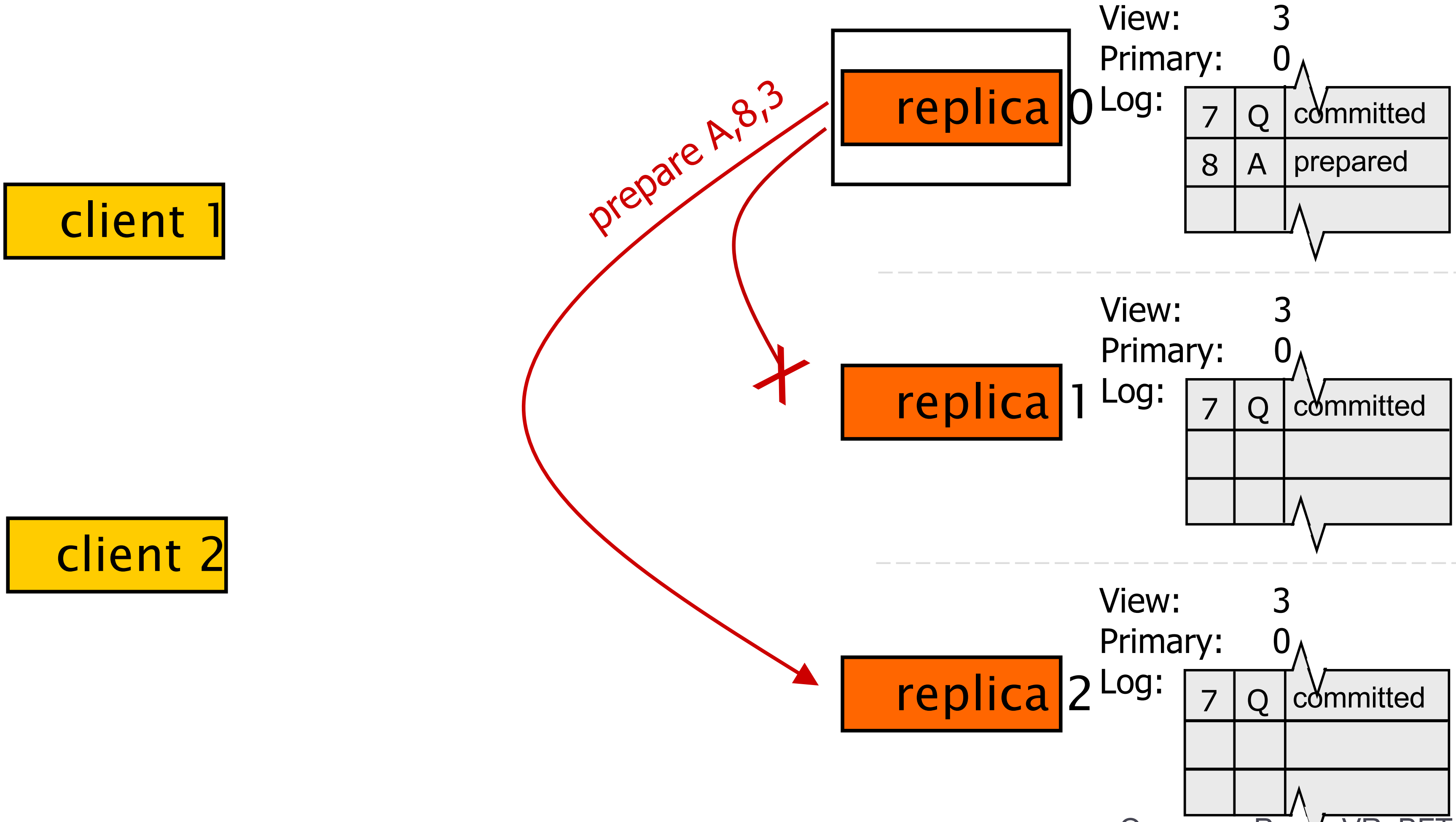
7	Q	committed

replica 2

View: 3
Primary: 0
Log:

7	Q	committed

Normal Case



View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

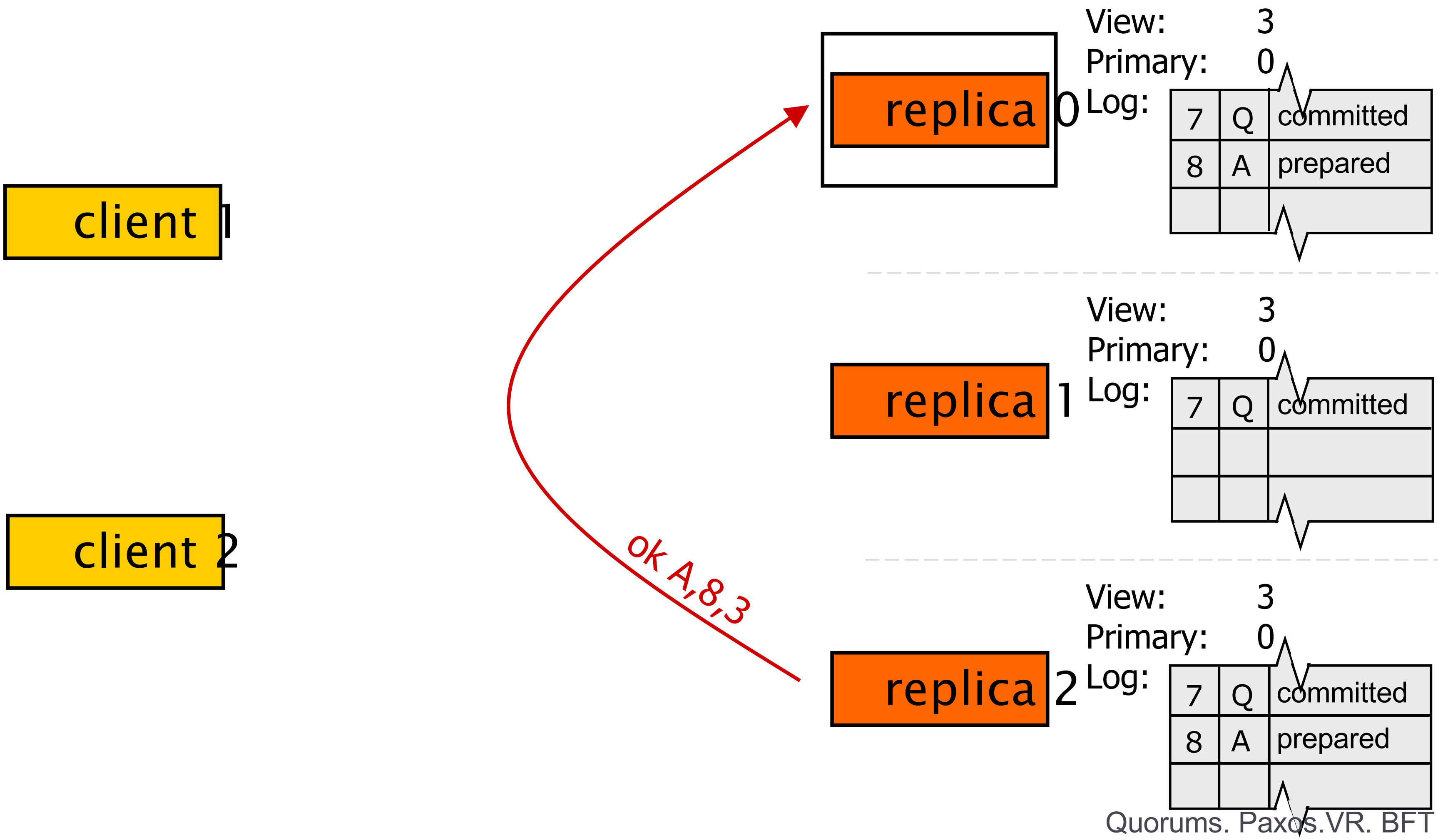
View: 3
Primary: 0
Log:

7	Q	committed

View: 3
Primary: 0
Log:

7	Q	committed

Normal Case



client 1

client 2

ok A,8,3

replica 0

View: 3

Primary: 0

Log:

7	Q	committed
8	A	prepared

replica 1

View: 3

Primary: 0

Log:

7	Q	committed

replica 2

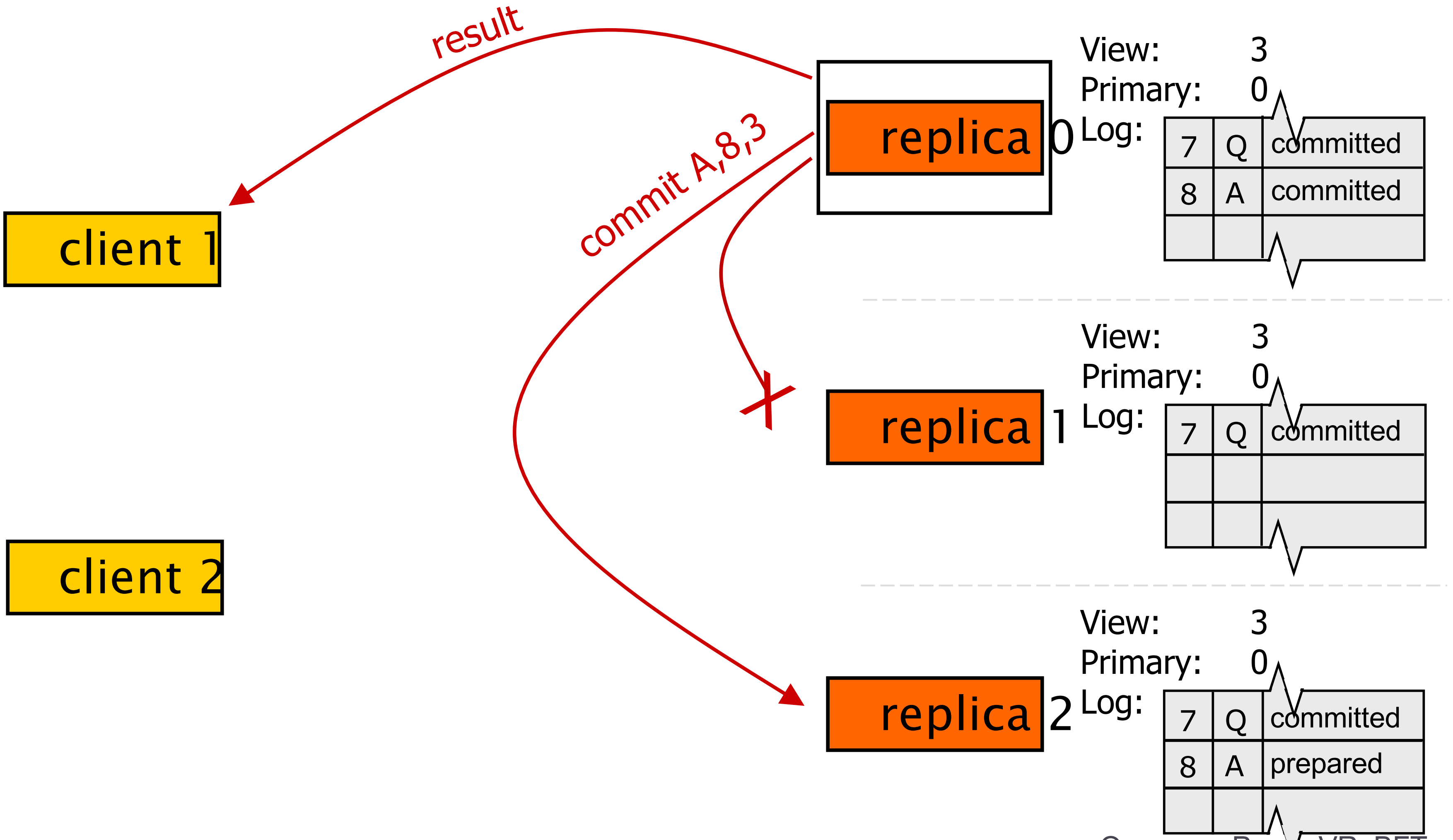
View: 3

Primary: 0

Log:

7	Q	committed
8	A	prepared

Normal Case



View: 3
Primary: 0
Log:

7	Q	committed
8	A	committed

View: 3
Primary: 0
Log:

7	Q	committed

View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

View Changes

- Used to mask primary failures
- Replicas monitor the primary
  - Client sends request to all
- Replica requests next primary to do a view change

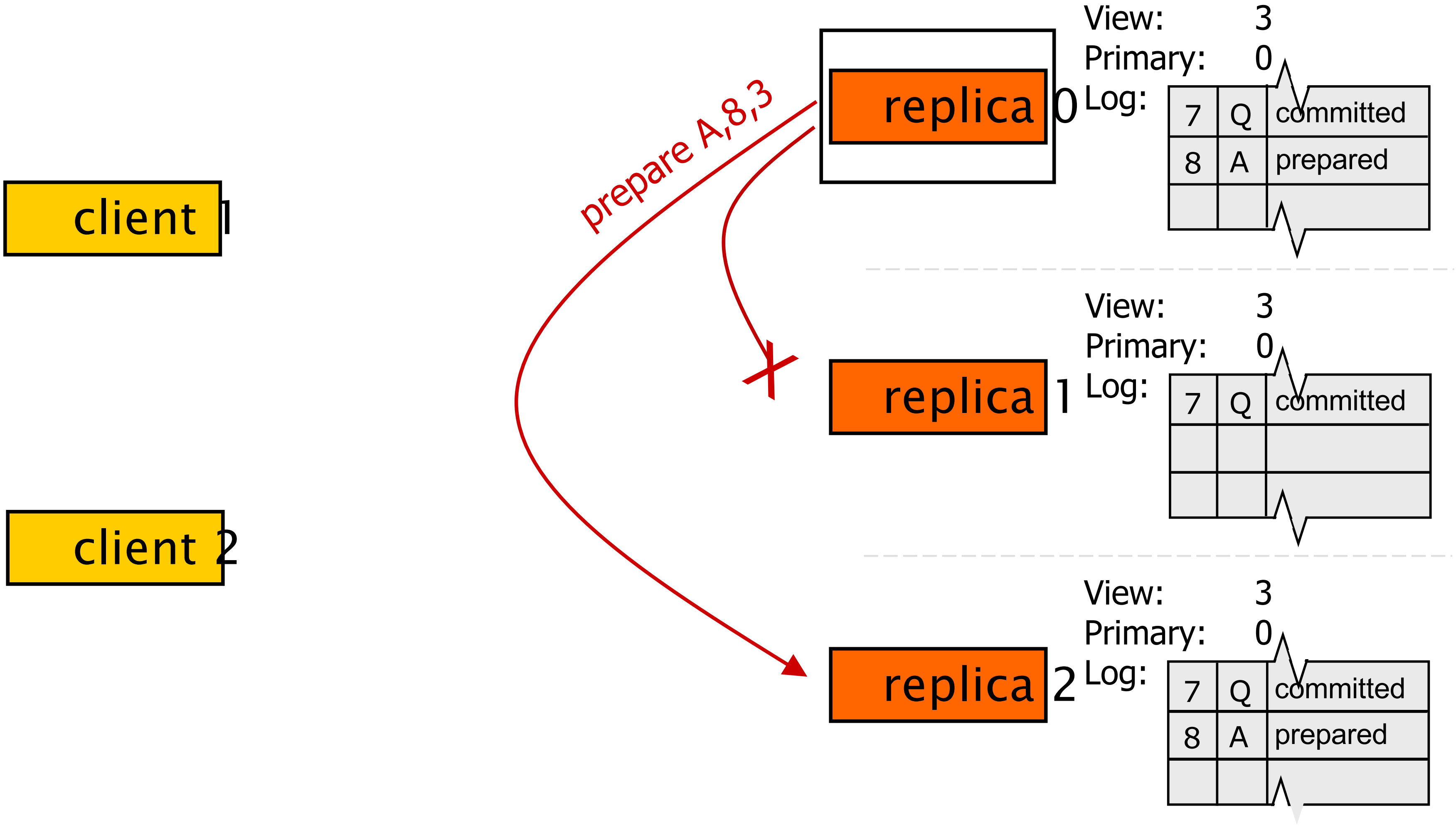
Correctness Requirement

- Operation order must be preserved by a view change

- For operations that are visible
  - executed by server
  - client received result

- An operation could be visible if it prepared at f+1 replicas
  - this is the commit point

View Change



View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

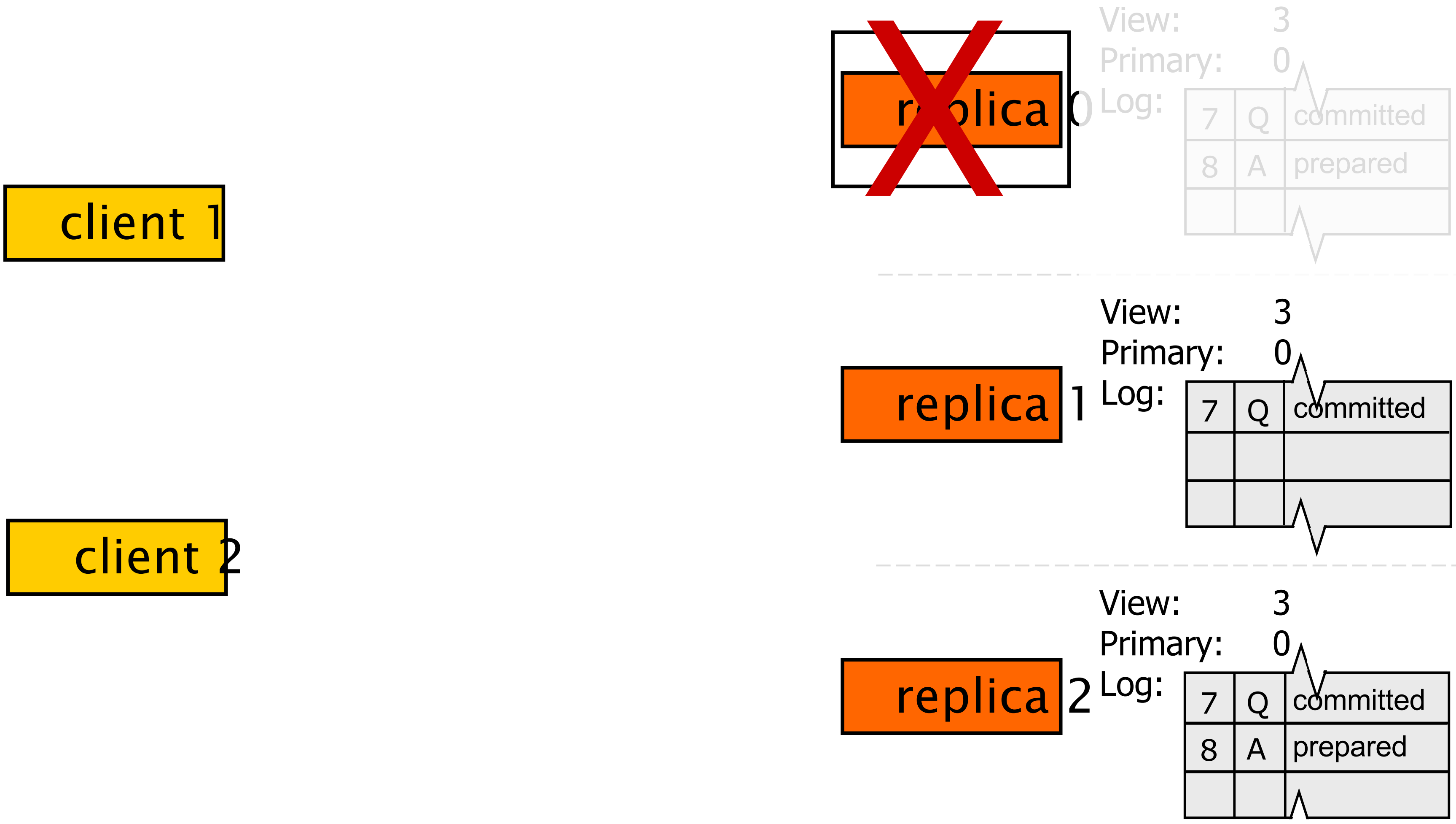
View: 3
Primary: 0
Log:

7	Q	committed

View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

View Change



client 1

client 2

replica 0

View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

replica 1

View: 3
Primary: 0
Log:

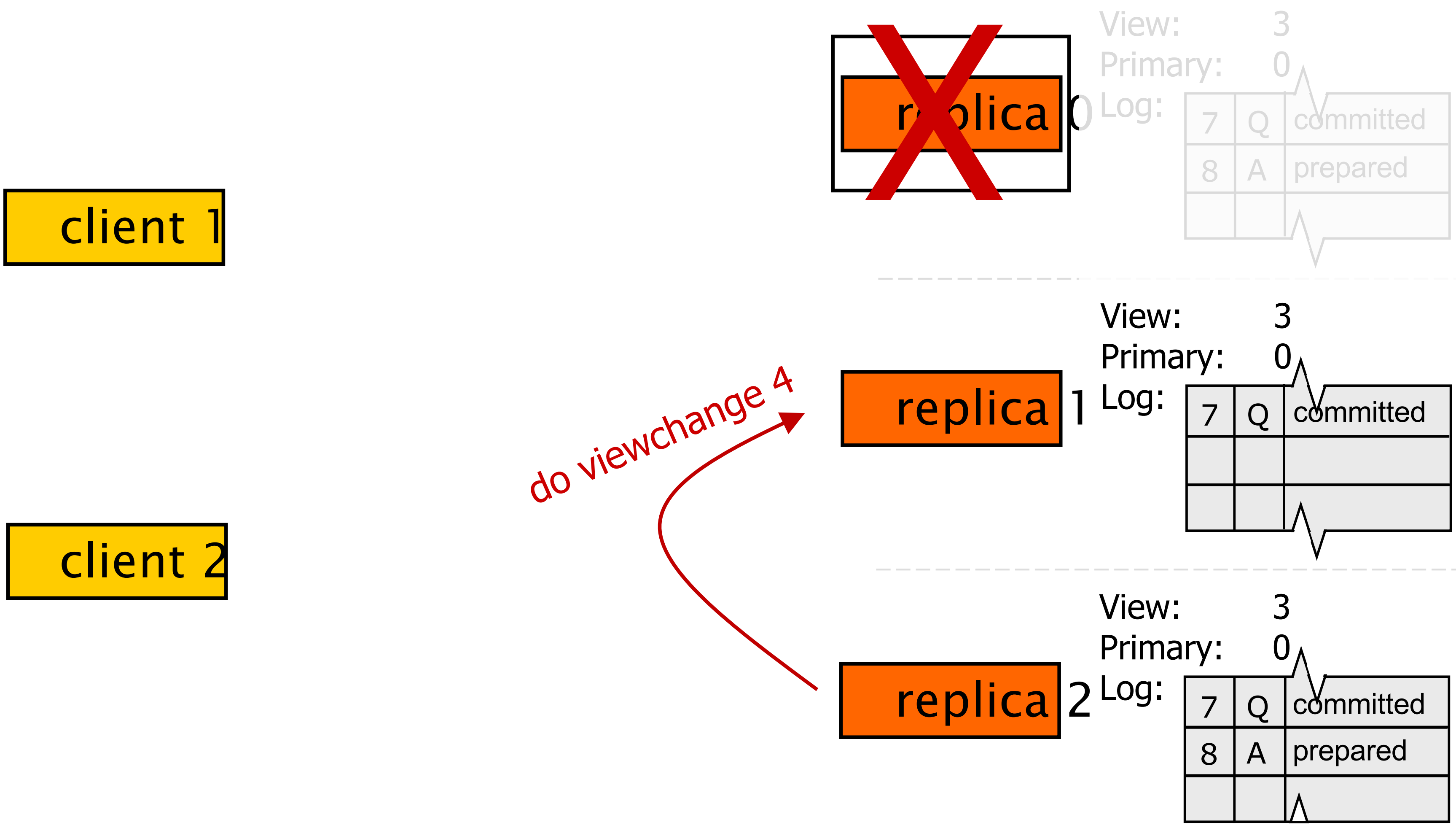
7	Q	committed

replica 2

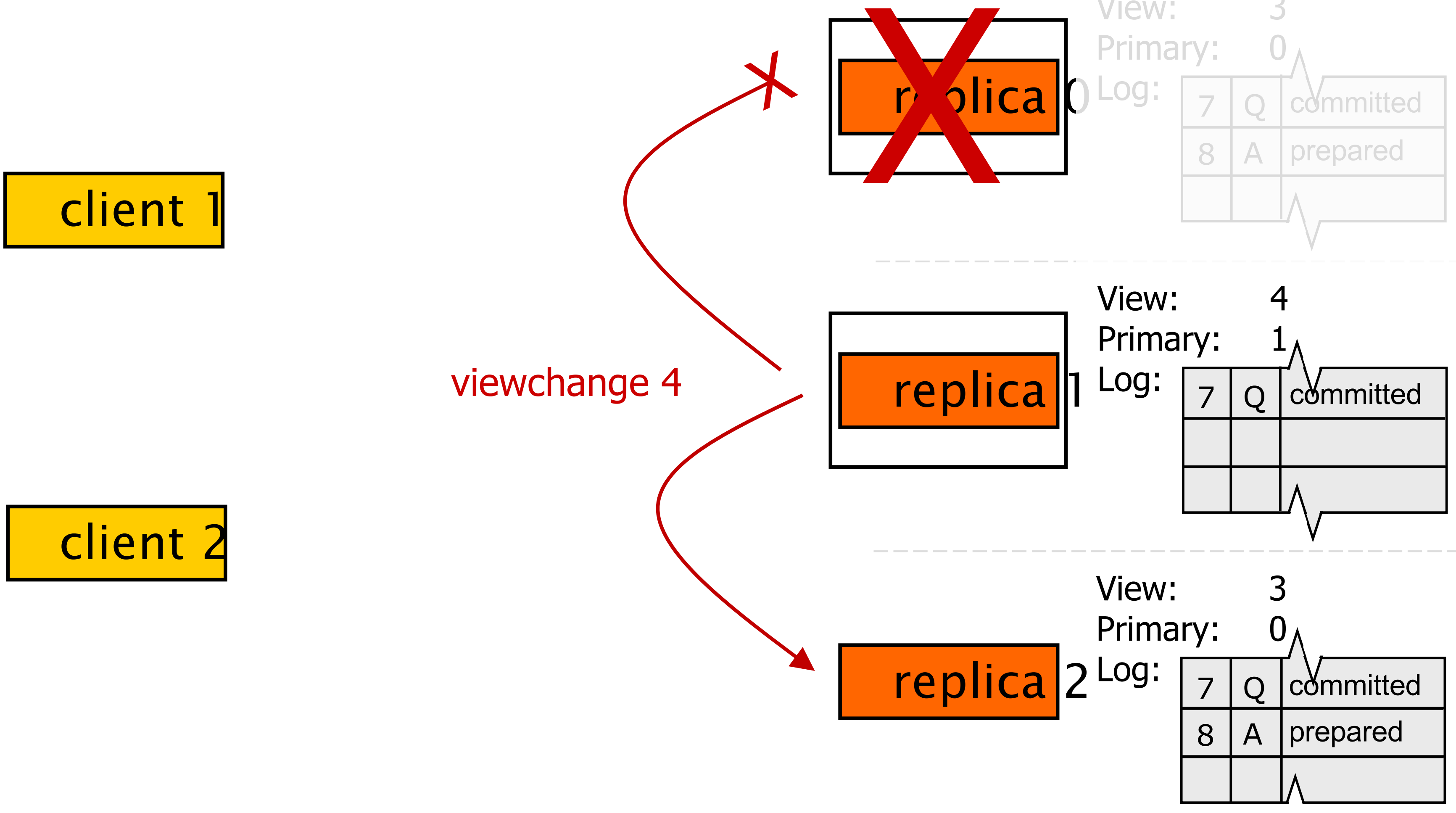
View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

View Change


replica 0
View: 3
Primary: 0
Log:
7
Q
committed
8
A
prepared
client 1
replica 1
View: 3
Primary: 0
Log:
7
Q
committed
do viewchange 4
client 2
replica 2
View: 3
Primary: 0
Log:
7
Q
committed
8
A
prepared

View Change



client 1

client 2

viewchange 4

replica 0

View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

replica 1

View: 4
Primary: 1
Log:

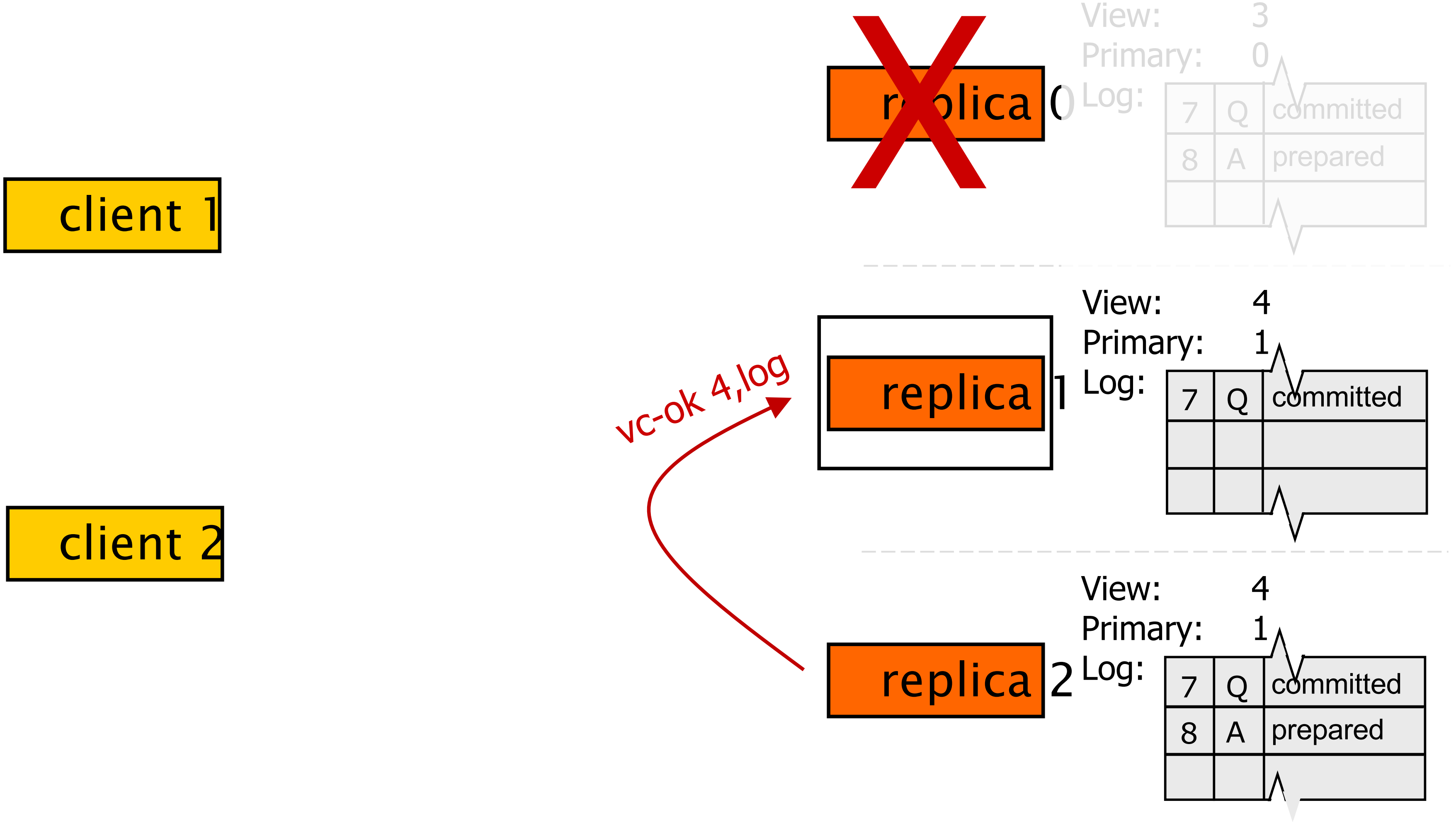
7	Q	committed

replica 2

View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

View Change



View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

View: 4
Primary: 1
Log:

7	Q	committed

View: 4
Primary: 1
Log:

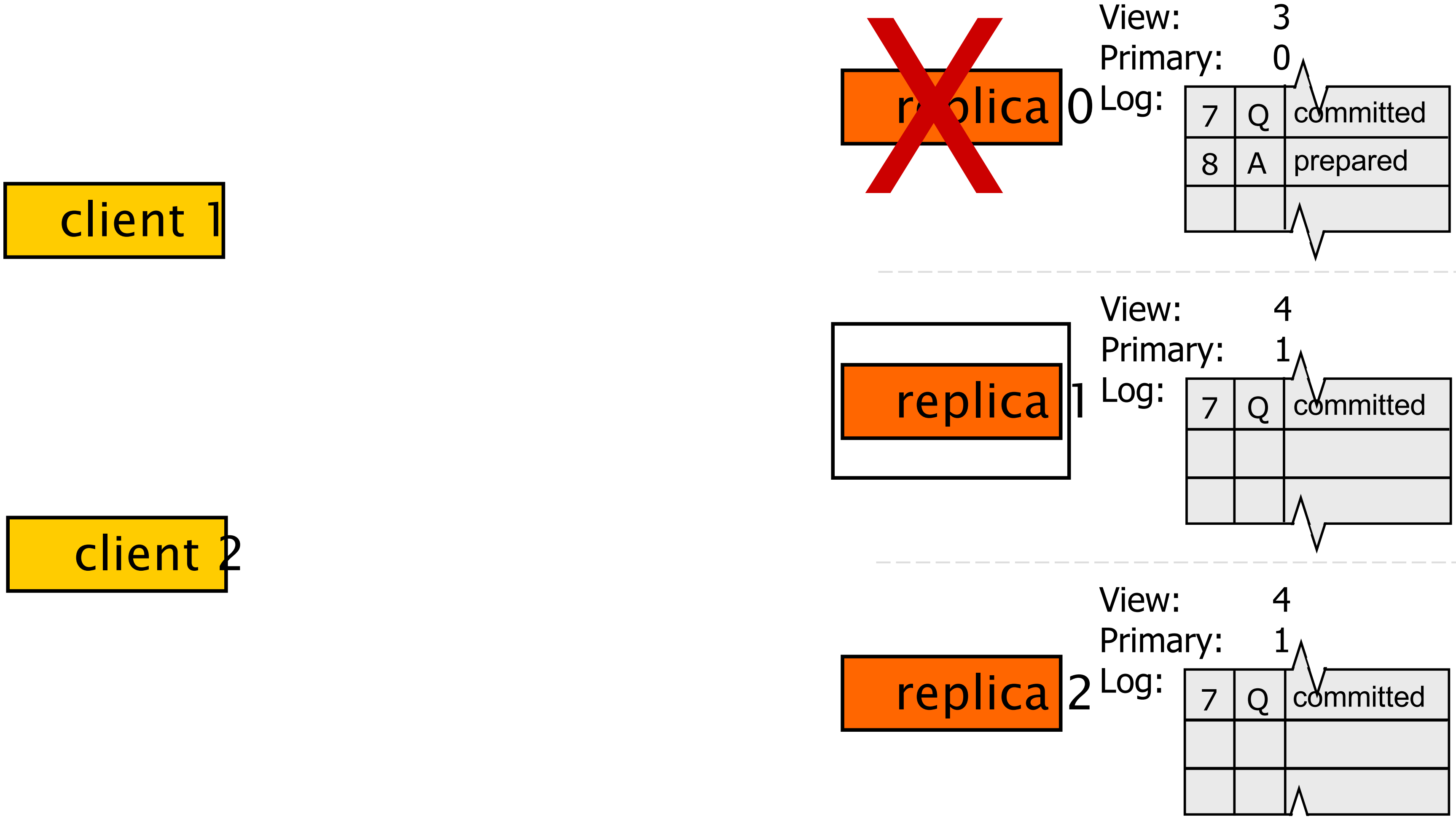
7	Q	committed
8	A	prepared

Double booking

- Sometimes more than one operation is assigned the same number
  - In view 3, operation A is assigned 8
  - In view 4, operation B is assigned 8

- Viewstamps
  - op number is <v#, seq#>

Example



client 1

client 2

replica 0

View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

replica 1

View: 4
Primary: 1
Log:

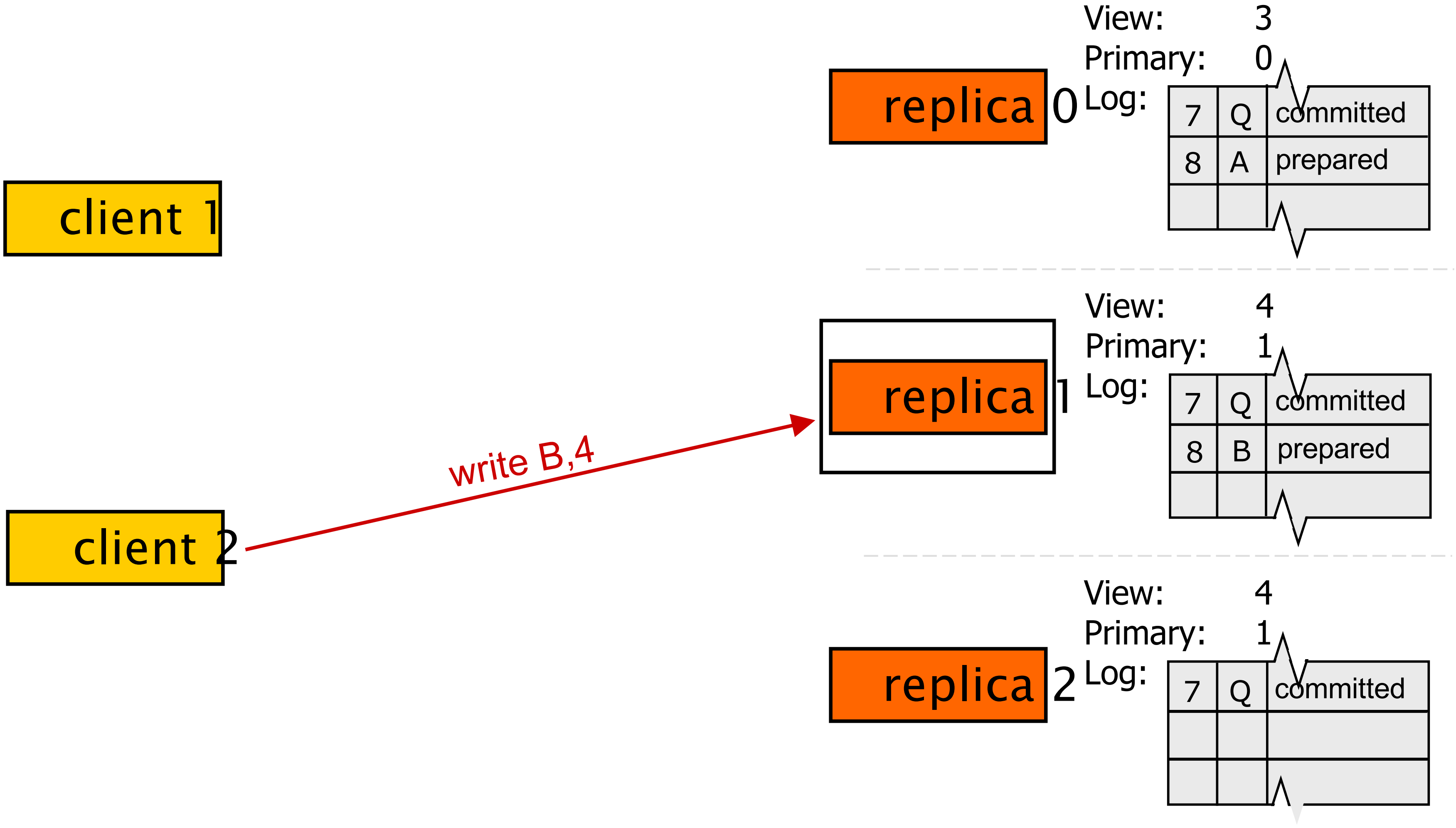
7	Q	committed

replica 2

View: 4
Primary: 1
Log:

7	Q	committed

Example



View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

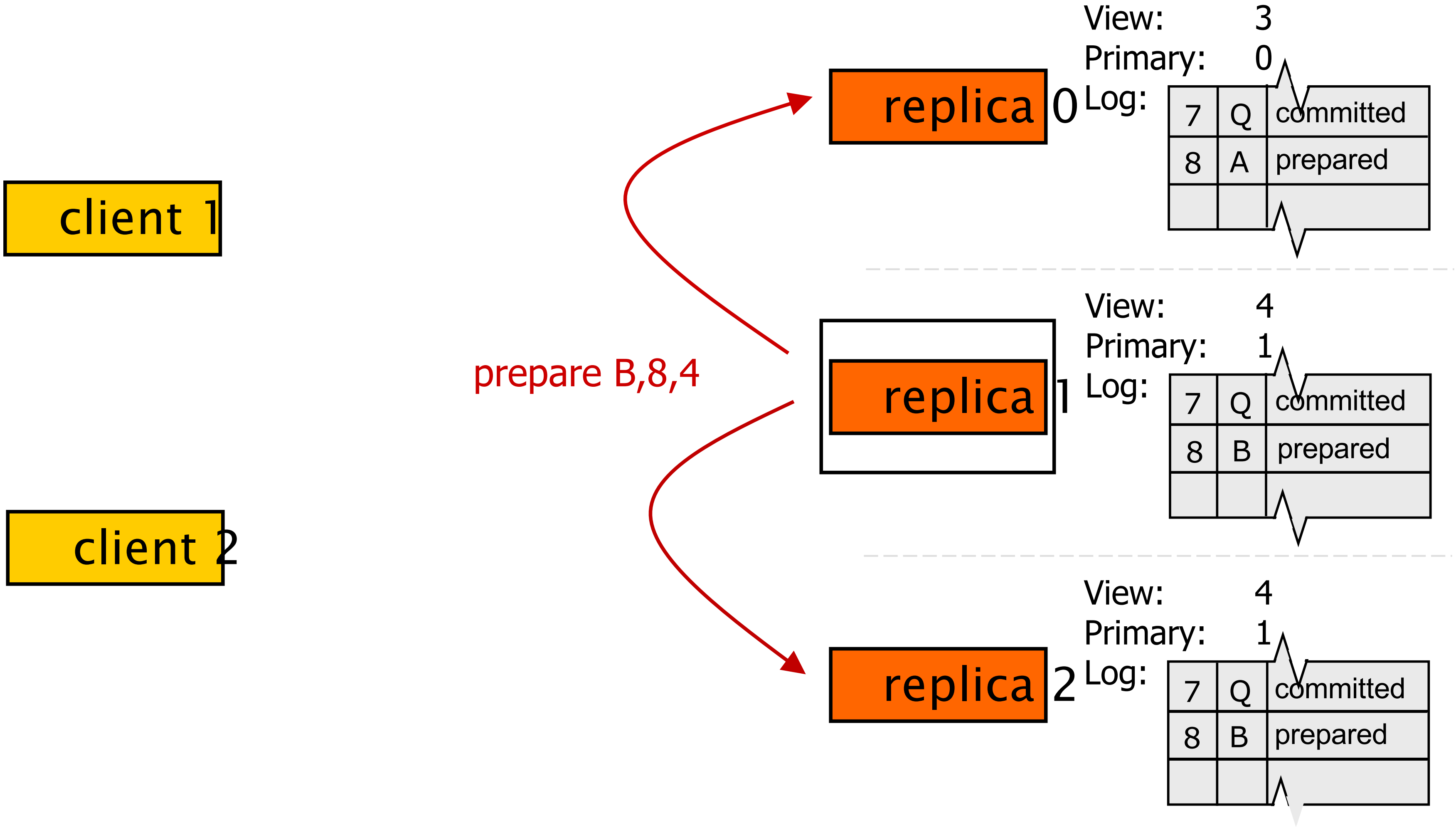
View: 4
Primary: 1
Log:

7	Q	committed
8	B	prepared

View: 4
Primary: 1
Log:

7	Q	committed

Example



View: 3
Primary: 0
Log:

7	Q	committed
8	A	prepared

View: 4
Primary: 1
Log:

7	Q	committed
8	B	prepared

View: 4
Primary: 1
Log:

7	Q	committed
8	B	prepared

Additional Issues

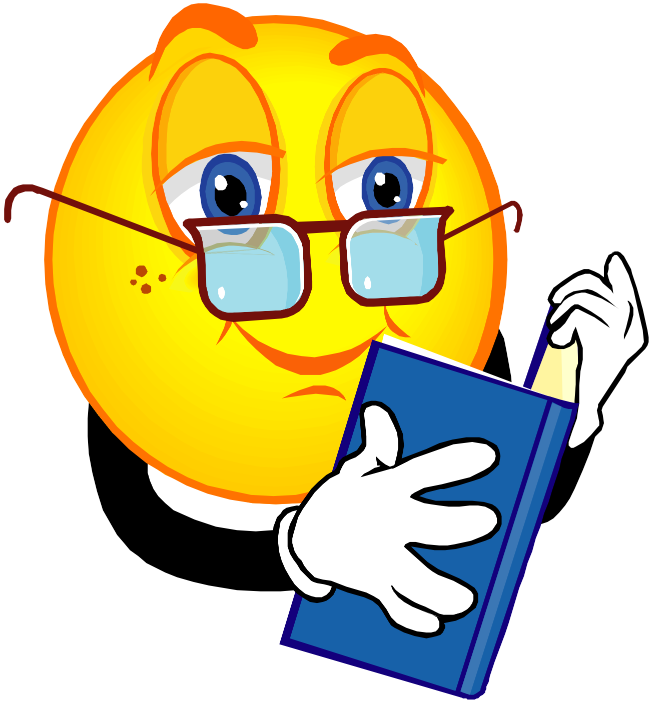
- State transfer
- Garbage collection of the log
- Selecting the primary

Improved Performance

- Lower latency for writes (3 messages)
  - Replicas respond at prepare
  - client waits for f+1
- Fast reads (one round trip)
  - Client communicates just with primary
  - Leases
- Witnesses (preferred quorums)
  - Use f+1 replicas in the normal case

Summary so far …

- State machine replication: approach to implementing fault-tolerant services
- Process groups: membership and VS
- Quorums: no membership, but leaders (or view), each operation requires a quorum,
- Paxos and VR
  - Approaches that rely on quorums for consensus and replication
  - One can use Paxos to further build a state machine replication



4: Byzantine replication

Byzantine-Resilient Replication

- How to design replication protocols that do not block and can tolerate malicious participants
- Use ideas from both Byzantine agreement and replication protocols (Viewstamped Replication)
- Ensure safety and liveness

BFT: Assumptions

- Provides safety without synchrony: ensures correct replies in spite of malicious servers
- Assumes eventual time bounds for liveness: messages will make it when network is stable

Assumes asynchronous communication for safety

BFT: Assumptions

- Servers can be malicious, arbitrary behavior, f malicious servers
- Failures are independent.
- Crypto options
  - Digital signatures
  - HMACs, requires n^2 symmetric keys

BFT: Overview

- Deterministic replicas start in same state
- Replicas execute same requests in same order
- Correct replicas produce identical replies
- Uses a leader to coordinate the protocol; each leader associated with a view
- Ensure ordering is not easy!
- What to do when the leader fails?

Dealing with Malicious Behavior

- Require 2f+1 out of 3f+1 participants to agree at each step
- This ensures that any 2 sets of 2f+1 will intersect in a correct replica
- Require at each step a proof that the 2f+1 agreed on the issue to ensure safety
- When leader (primary) fails, new leader (view) elected

Client-Server Interaction

- Client submits a request to the primary
- If timeout occurs, suspects the primary and sends to every server
- Servers order the request
- Client waits for answers from servers. How many identical answers should a client wait for?

f+1 identical responses to be guaranteed that at least one correct server returned the correct value.

BFT: Components

- Normal case operation:
  - primary is not faulty (does not fail and it is not malicious)
- View changes:
  - how to deal with view changes
- Garbage collection:
  - when it is time to garbage collect information maintained by each server

Safety and Liveness

- Safety:
  - ensure ordering of requests within a view and across view
- Liveness:
  - there is progress at each step, including selecting a new leader

BFT: Normal Case

- Primary goal of normal case is to ensure ordering of requests within a view; also has a phase that works in combination with the view change protocol
- Three phases:
  - **pre-prepare** assigns order to requests
  - **prepare** ensures servers agree on order within views
  - **commit** ensures servers agree on order across views
- Messages are logged and authenticated

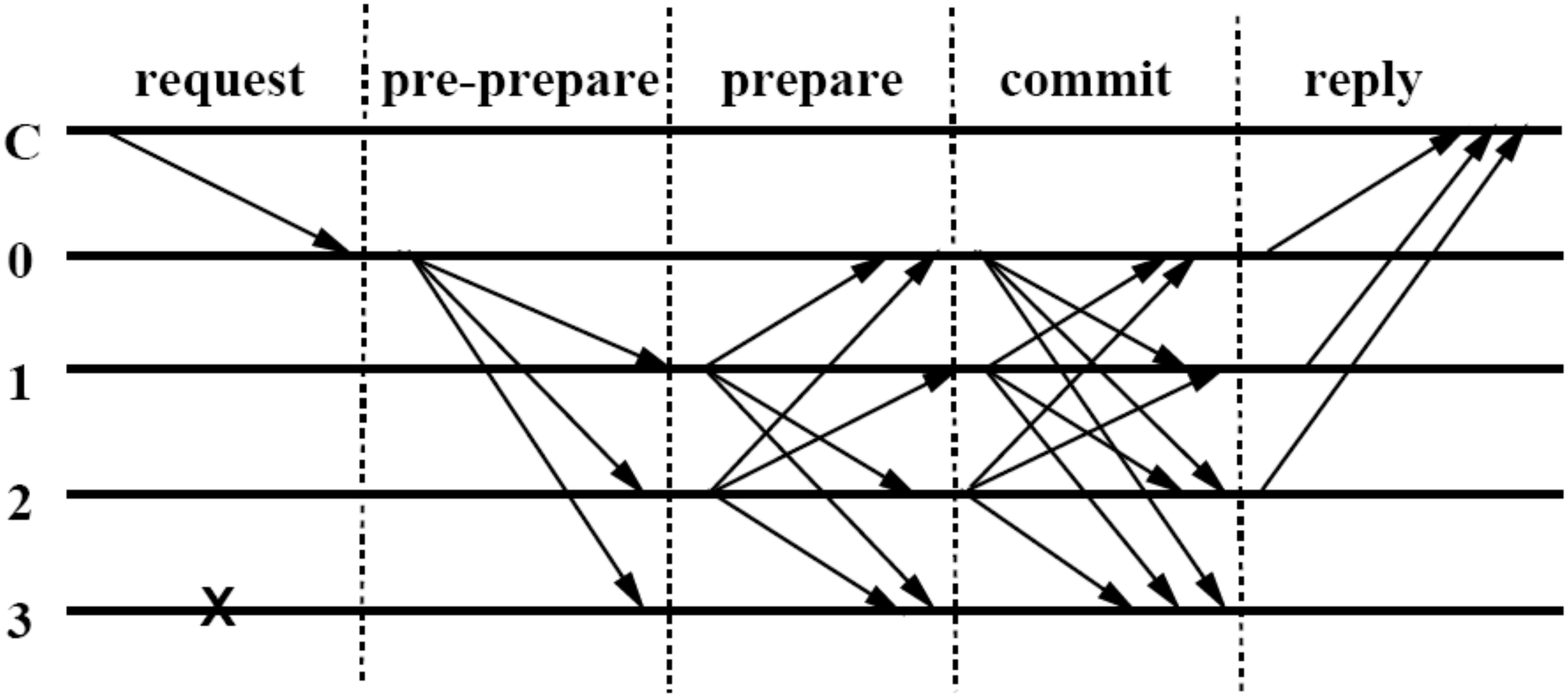
- Matching means: same view, sequence numbers, and message digest

Normal Case Details

- Client *c* sends ***m* = <REQUEST,o,t,c>**$_c$ to the *primary*. (o=operation,t=timestamp)
- Primary *p* assigns sequence *n* to *m* and sends

  <PRE-PREPARE,v,n,m>$_p$ to other replicas, v is the current view; unique identifier is given by **n and v**
- If server *i* **accepts** the message, it sends **<PREPARE,v,n,d,i>**$_i$ to other replicas. (d is hash of the request). Signals that *i* agrees to assign *n* to *m* in *v*.
- Once server *i* has a pre-prepare and 2*f* matching prepare messages, it sends **<COMMIT,v,n,d,i>**$_i$ to other replicas. At this point, correct replicas agree on an order of requests within view v.
- Once server *i* has 2*f*+1 matching prepare and commit messages (2f+1 prepare and 2f+1 commit), it executes m, then sends **<REPLY,v,t,c,i,r>**$_i$ to the client.

BFT: How Does It Work?



More Details

- Servers accept pre-prepare <PRE-PREPARE,v,n,m> if
  - Their current view is view v
  - They did not accept pre-prepare for v,n with different request

- All collected pre-prepare and 2f matching prepares serve as a certificate for the next step: P-certificate(m,v,n)

- Request m executed after:
  - having C-certificate(m,v,n)
  - executing requests with sequence number less than n

BFT: View Change

- Provide liveness when primary fails:
  - Timeouts used to trigger view changes
  - Mapping between primary and view number
  - Increase current view number and select new primary (≡ view number mod 3f+1)
- Preserve safety
  - ensure replicas are in the same view long enough
  - prevent denial-of-service attacks

BFT: View Change Details

- When servers suspect the primary, they start a view change
  - A backup starts timer when it is waiting for executing a request and stops it when it is no longer waiting
  - If timer times out something is wrong with Primary
  - Change view so that Primary gets changed

- A backup sends <VIEW-CHANGE, *v+1, n, C, P, i* >$_{\sigma i}$
  - *C* is a proof of last stable check-point
  - *P* is a proof of due requests after the check-point

BFT: View Change Details

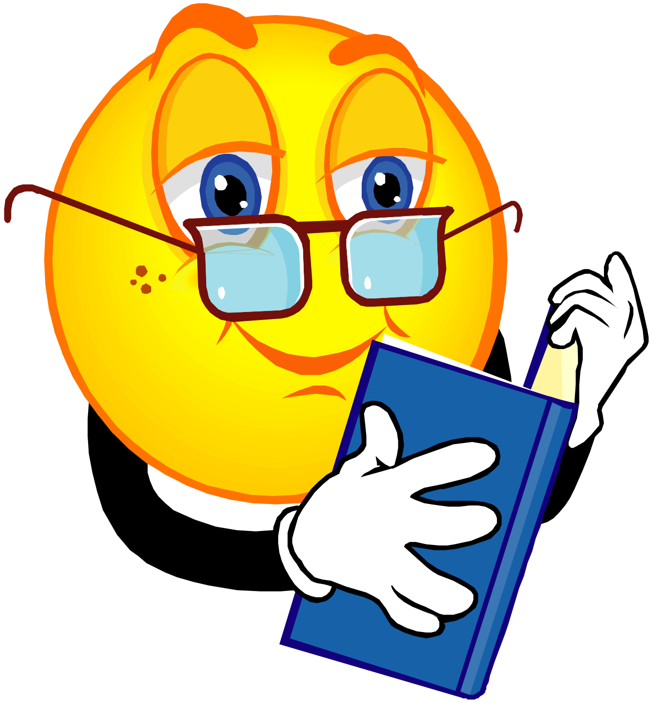
- When a primary of new view gets *2f* VIEW-CHANGE messages, it declares new view
- The new Primary sends < NEW-VIEW, $v+1, V, O$ >$_{\sigma i}$
- *V* is a proof containing valid VIEW-CHANGE messages
- *O* is a set containing PRE-PREPARE messages needed to carry the incomplete messages from previous view into new view

BFT: Garbage Collection

- The logs are cleaned periodically
- Before cleaning the logs a backup must be sure that all requests whose messages it is going to clean have been successfully executed
- After fixed number of requests replicas send check-point signals $< \text{CHECKPOINT}, n,d,i >_{\sigma i}$
- When a replica receives *2f+1* check-point messages, it clears all messages for requests up to *n.*

Summary

- BFT – requires a minimum of 3f+1 participant, 3 communication rounds and f+1 identical answers to the client
- Scaling beyond BFT, one can combine fault-tolerant approaches (like Paxos) with BFT to achieve better performance on wide area networks.



5: Raft

Slides from Diego Ongaro and John Ousterhout

Raft's Design Goals

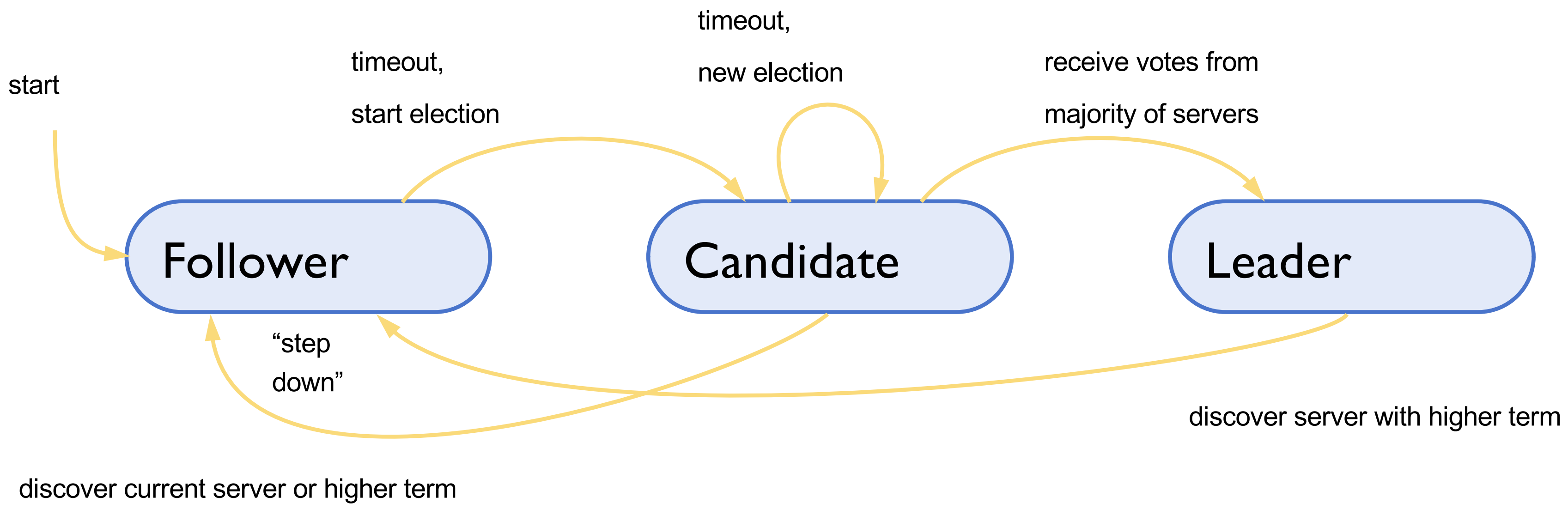
- Alternative to Paxos; Paxos is too complex and incomplete for real implementations
- Algorithm for building real systems
  - Must be correct, complete, and perform well
  - Must also be **understandable**
- "What would be easier to understand or explain?"
  - Fundamentally different decomposition than Paxos
  - Less complexity in state space
  - Less mechanism

Raft Overview

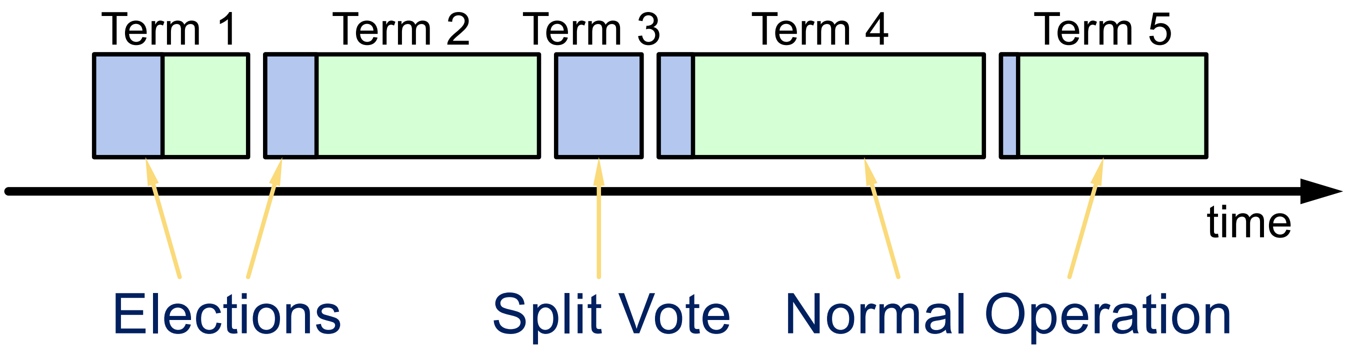
1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
   - Implementing linearizeable semantics
6. Configuration changes:
   - Adding and removing servers

Server States

- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
  - Follower: completely passive
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers



Terms



- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most 1 leader per term
- Some terms have no leader (failed election)
- Each server maintains current term value
- Key role of terms: identify obsolete information

Heartbeats and Timeouts

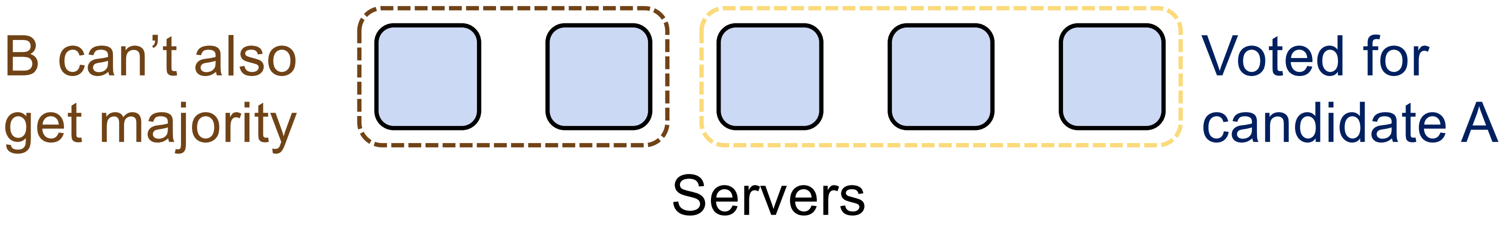
- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send **heartbeats** to maintain authority
- If **electionTimeout** elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

Election Basics

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

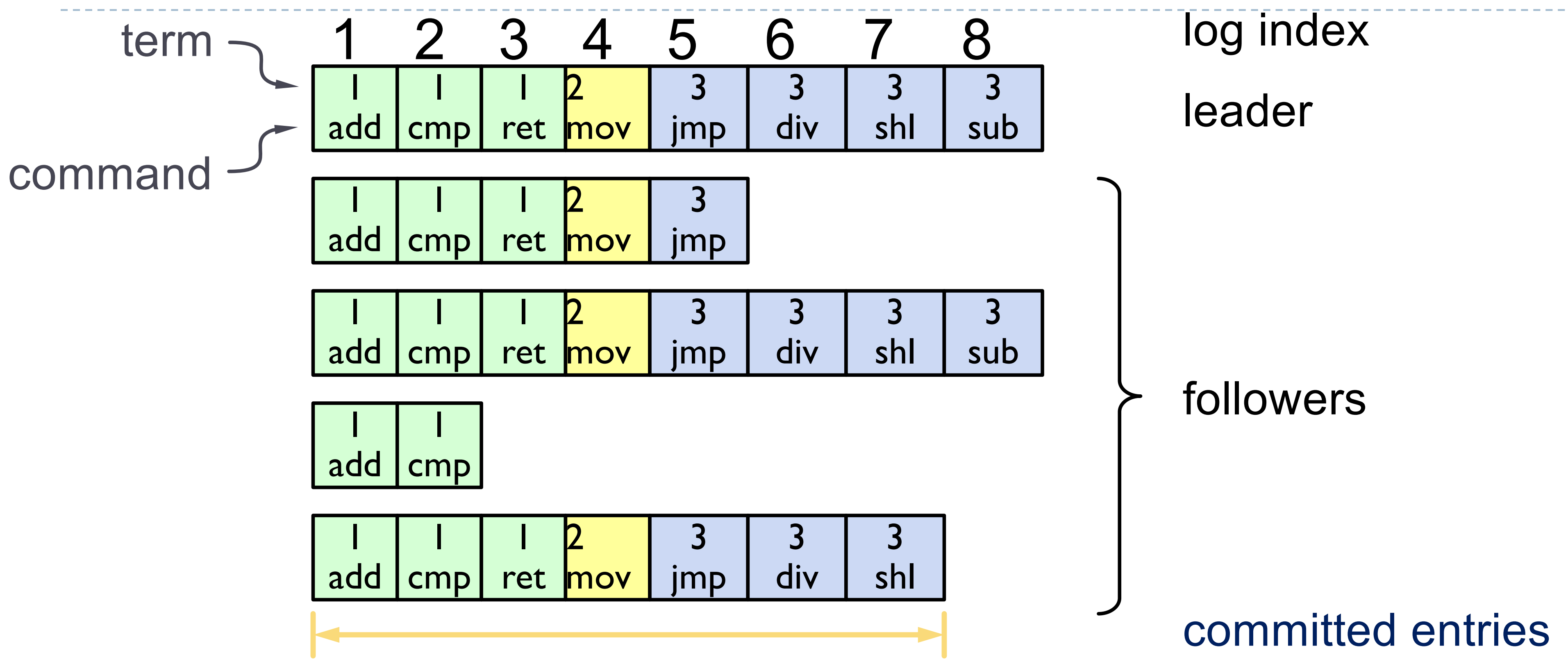
Elections, cont'd

- Safety: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in same term



- Liveness: some candidate must eventually win
  - Choose election timeouts **randomly** in [T, 2T]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time

Log Structure



log index	1	2	3	4	5	6	7	8
leader	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	3 shl	3 sub
followers	1 add	1 cmp	1 ret	2 mov	3 jmp			
	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	3 shl	3 sub
	1 add	1 cmp						
	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	3 shl	

- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable, will eventually be executed by state machines

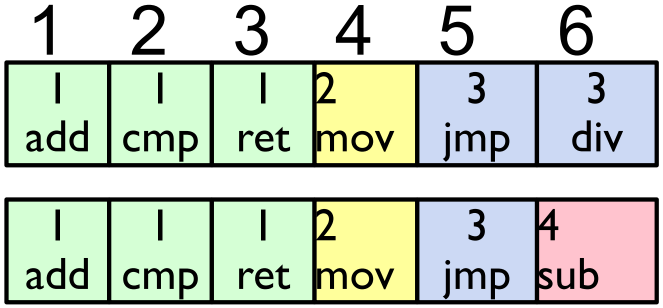
Normal Operation

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

Log Consistency

High level of coherency between logs:

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries

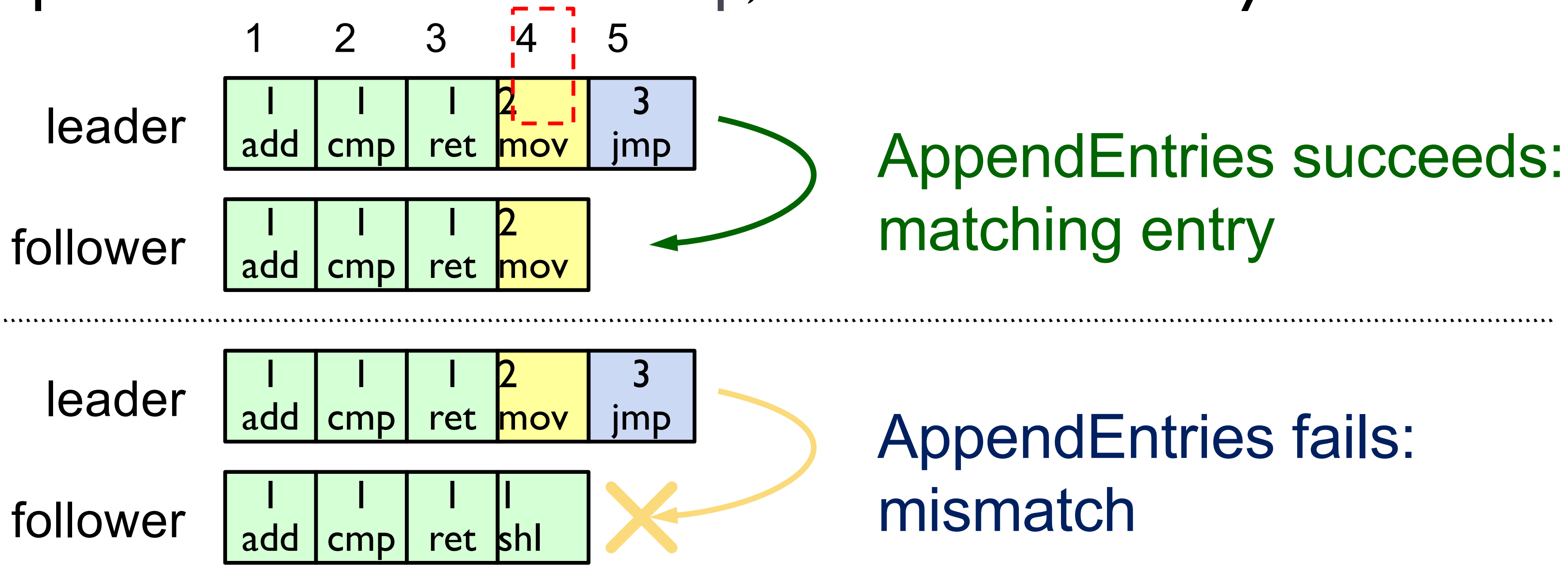


1	2	3	4	5	6
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

- If a given entry is committed, all preceding entries are also committed

AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency



	1	2	3	4	5
leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	2 mov	

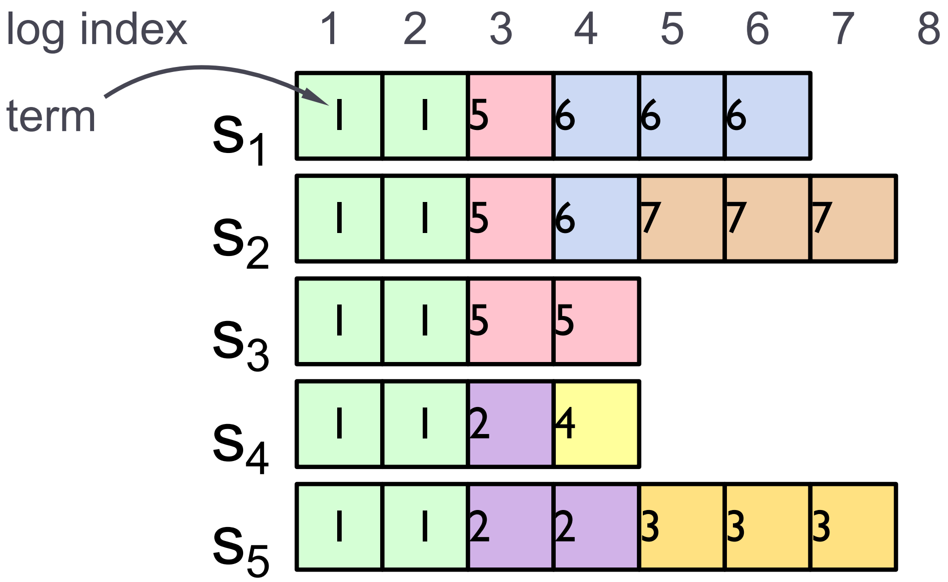
AppendEntries succeeds: matching entry

leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	1 shl	

AppendEntries fails: mismatch

Leader Changes

- At beginning of new leader's term:
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader's log is "the truth"
  - **Will eventually make follower's logs identical to leader's**
  - Multiple crashes can leave many extraneous log entries:

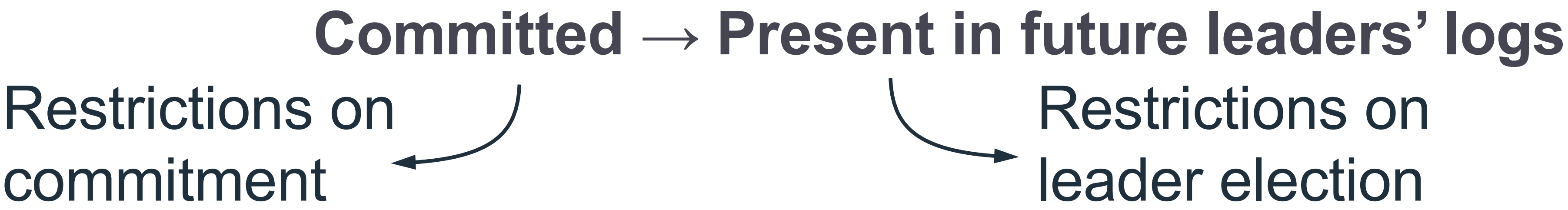


log index	1	2	3	4	5	6	7	8
s_1	1	1	5	6	6	6		
s_2	1	1	5	6	7	7	7	
s_3	1	1	5	5				
s_4	1	1	2	4				
s_5	1	1	2	2	3	3	3	

Safety Requirement

Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- Raft safety property:
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
- This guarantees the safety requirement
  - Leaders never overwrite entries in their logs
  - Only entries in the leader's log can be committed
  - Entries must be committed before applying to state machine



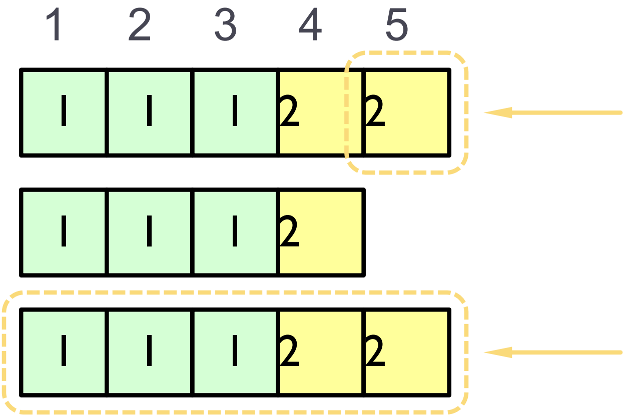
Committed → Present in future leaders' logs

Restrictions on commitment

Restrictions on leader election

Picking the Best Leader

- Can’t tell which entries are committed!



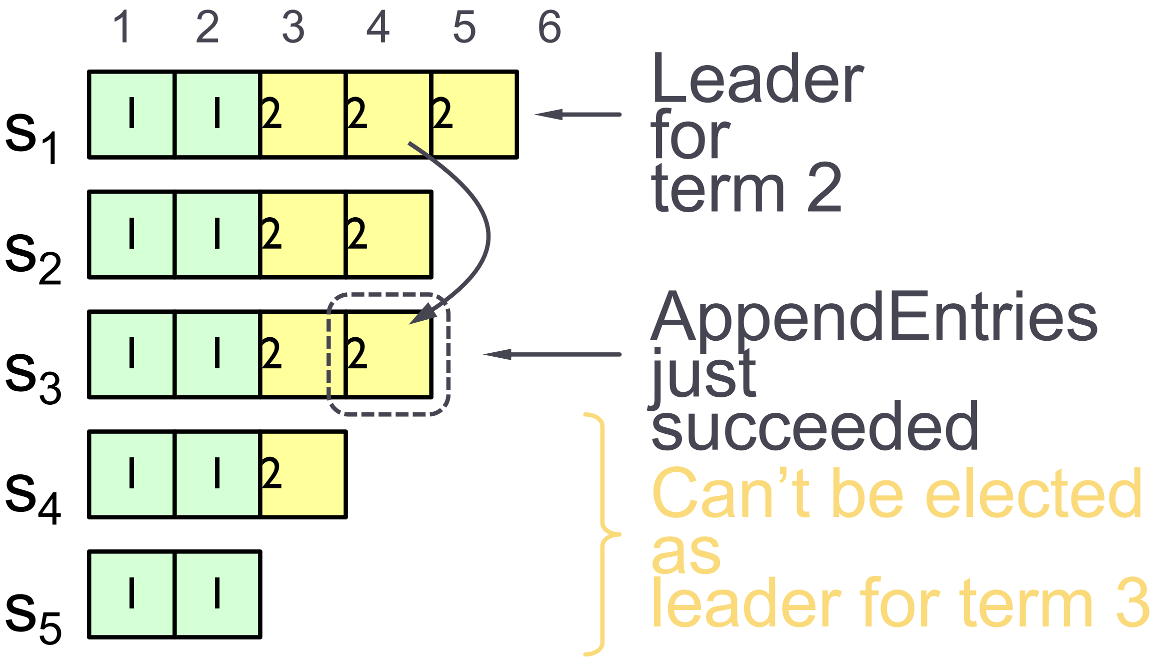
Committed ?

unavailable during
leader transition

- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include log info in RequestVote RPCs
    (index & term of last log entry)
  - Voting server V denies vote if its log is “more complete”:
    $(\text{lastTerm}_V > \text{lastTerm}_C) \;||$
    $(\text{lastTerm}_V == \text{lastTerm}_C) \;\&\&\; (\text{lastIndex}_V > \text{lastIndex}_C)$
  - Leader will have “most complete” log among electing majority

Committing Entry from Current Term

- Case #1/2: Leader decides entry in current term is committed



	1	2	3	4	5	6
s_1	1	1	2	2	2	
s_2	1	1	2	2		
s_3	1	1	2	2		
s_4	1	1	2			
s_5	1	1				

Leader for term 2

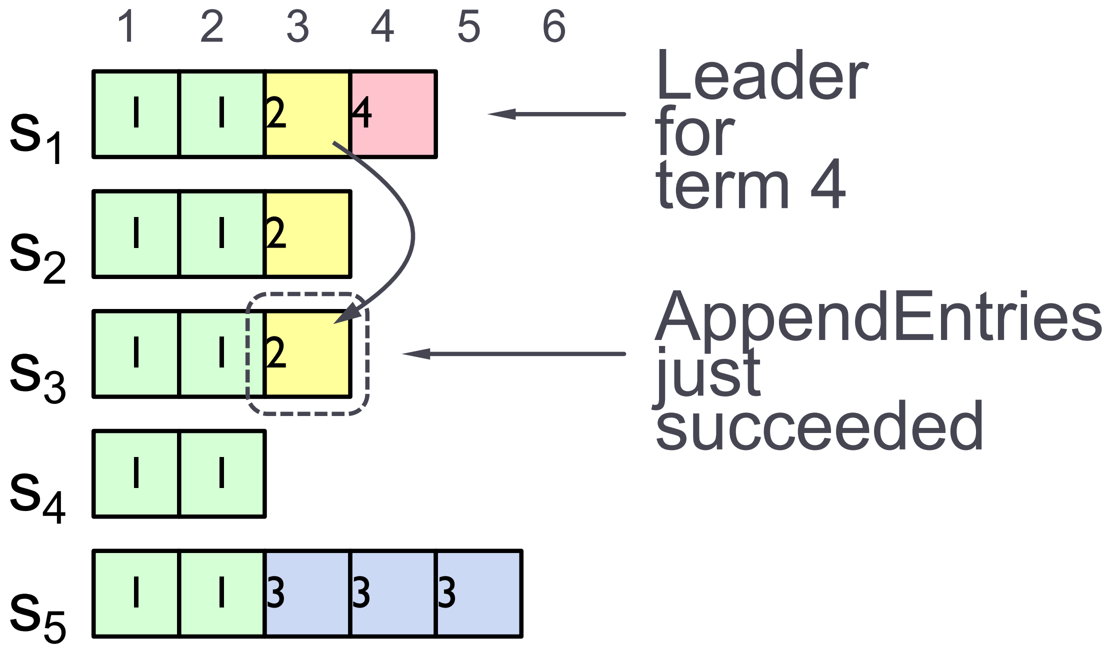
AppendEntries just succeeded

Can't be elected as leader for term 3

- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term

- Case #2/2: Leader is trying to finish committing entry from an earlier term



	1	2	3	4	5	6
s_1	1	1	2	4		
s_2	1	1	2			
s_3	1	1	2			
s_4	1	1				
s_5	1	1	3	3	3	

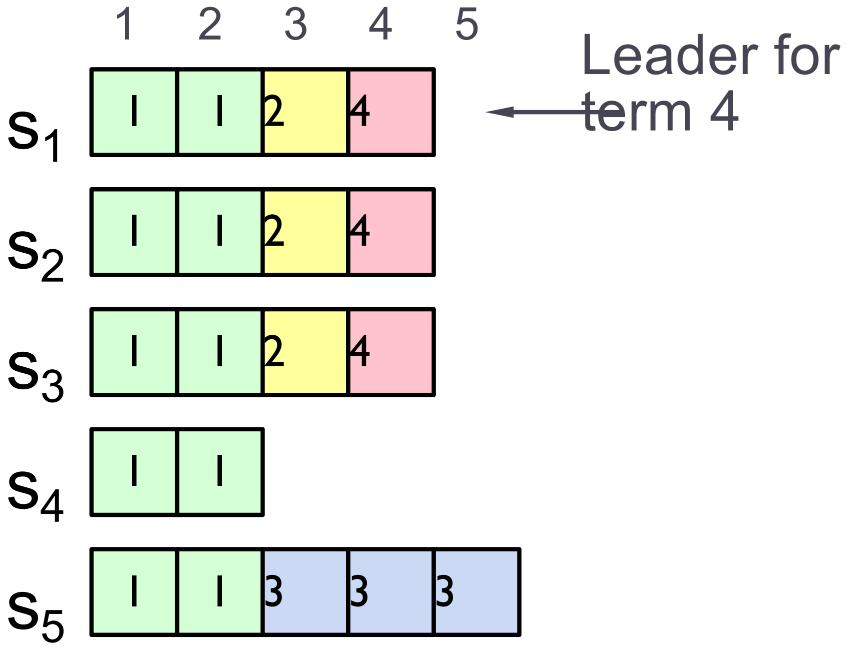
Leader for term 4

AppendEntries just succeeded

- Entry 3 not safely committed:
  - s_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3!

New Commitment Rules

- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



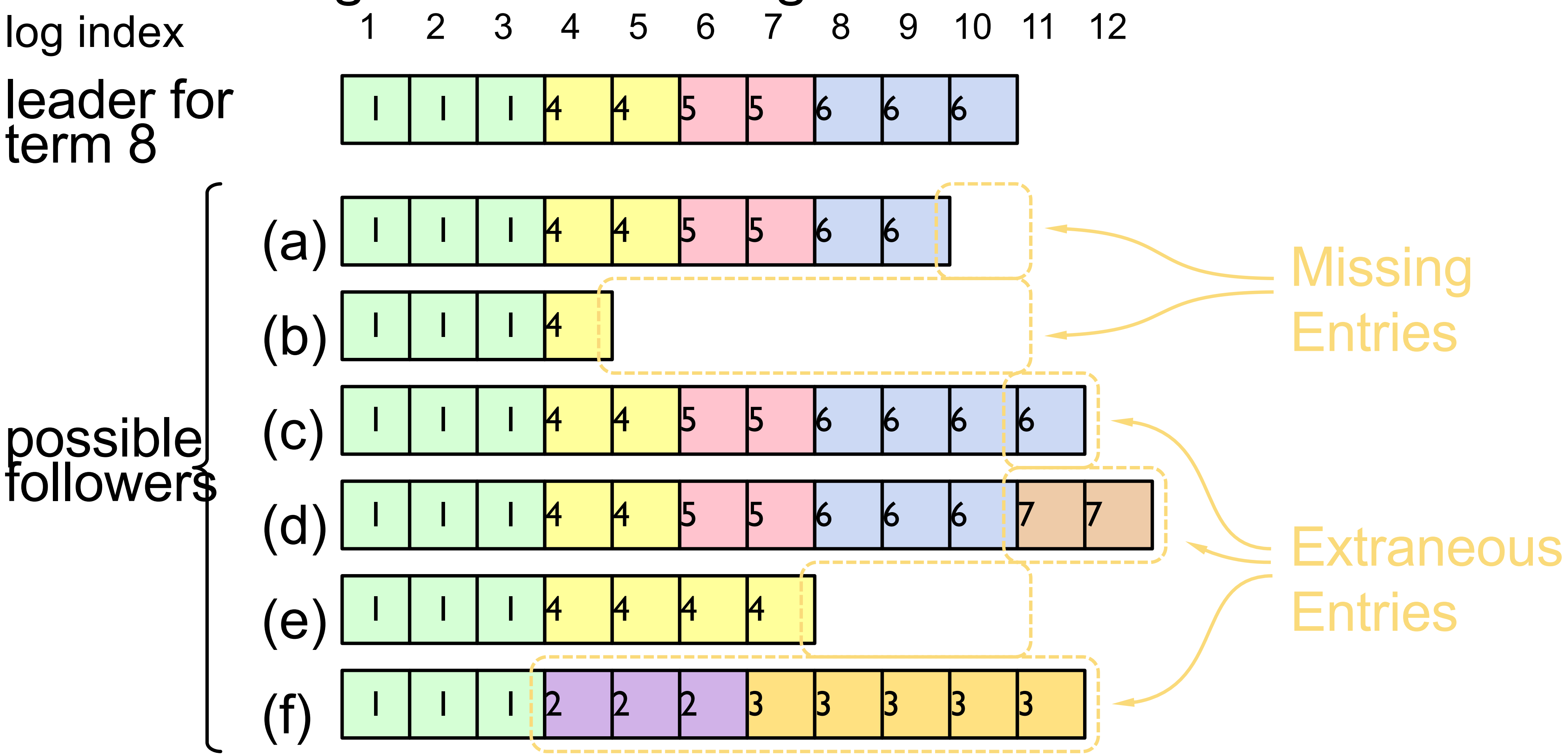
	1	2	3	4	5
s_1	1	1	2	4	
s_2	1	1	2	4	
s_3	1	1	2	4	
s_4	1	1			
s_5	1	1	3	3	3

Leader for term 4

Combination of election rules and commitment rules makes Raft safe

Log Inconsistencies

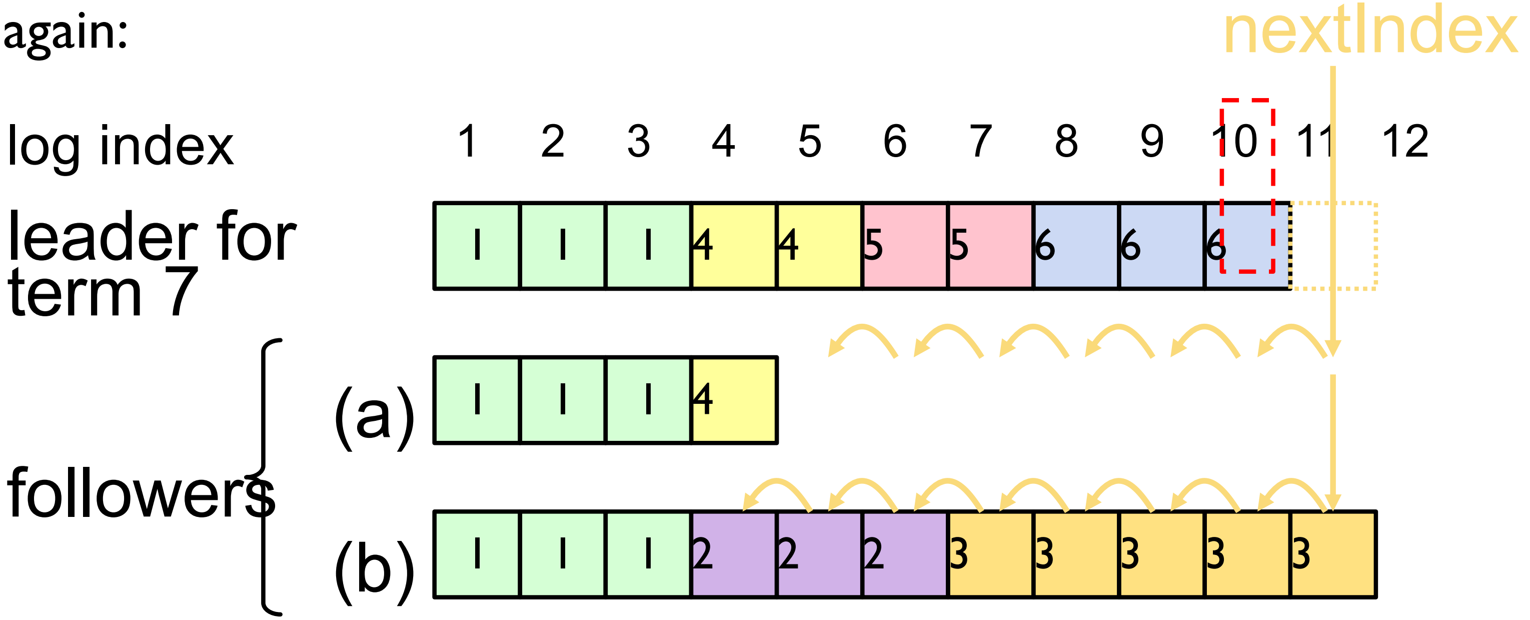
Leader changes can result in log inconsistencies:



log index	1	2	3	4	5	6	7	8	9	10	11	12
leader for term 8	1	1	1	4	4	5	5	6	6	6		
(a)	1	1	1	4	4	5	5	6	6			
(b)	1	1	1	4								
(c)	1	1	1	4	4	5	5	6	6	6	6	
(d)	1	1	1	4	4	5	5	6	6	6	7	7
(e)	1	1	1	4	4	4	4					
(f)	1	1	1	2	2	2	3	3	3	3	3	

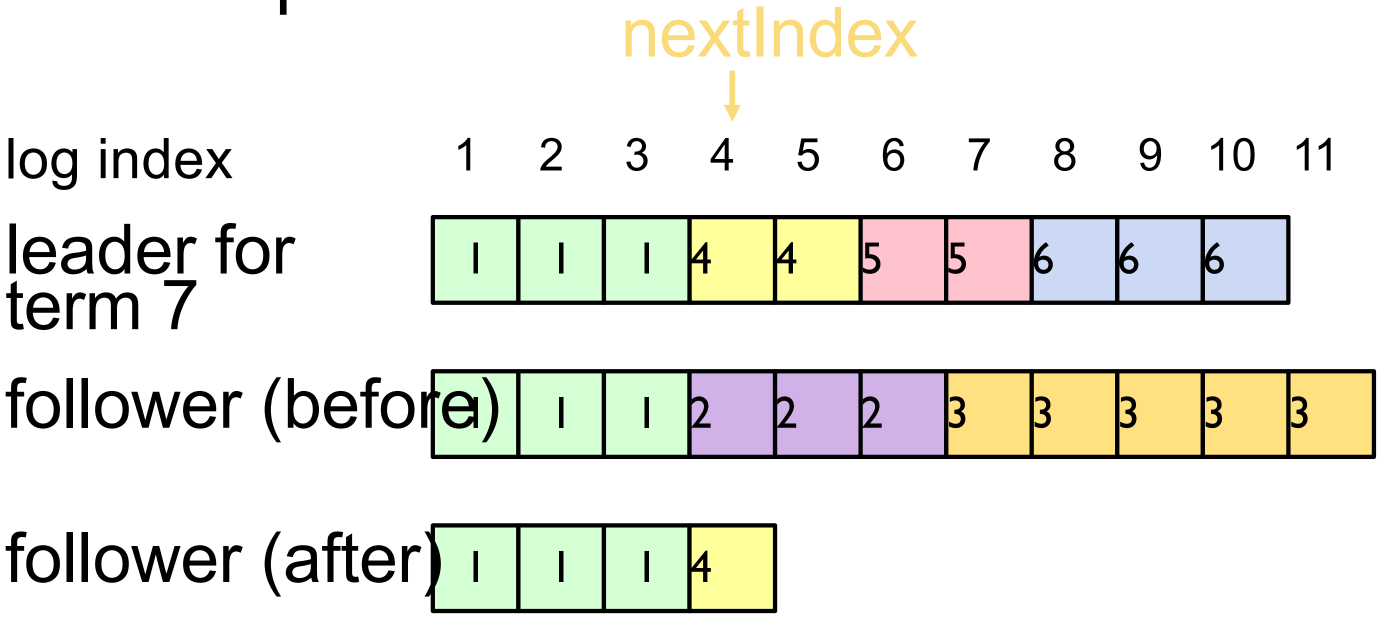
Repairing Follower Logs

- New leader must make follower logs consistent with its own
  - Delete extraneous entries
  - Fill in missing entries
- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- When AppendEntries consistency check fails, decrement nextIndex and try again:



Repairing Logs, cont'd

- When follower overwrites inconsistent entry, it deletes all subsequent entries:



log index	1	2	3	4	5	6	7	8	9	10	11
leader for term 7	1	1	1	4	4	5	5	6	6	6	
follower (before)	1	1	1	2	2	2	3	3	3	3	3
follower (after)	1	1	1	4							

Neutralizing Old Leaders

- Deposed leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries
- **Terms** used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally
- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries

Client Protocol

- Send commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
- Leader does not respond until command has been logged, committed, and executed by leader's state machine
- If request times out (e.g., leader crash):
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader

Client Protocol, cont'd

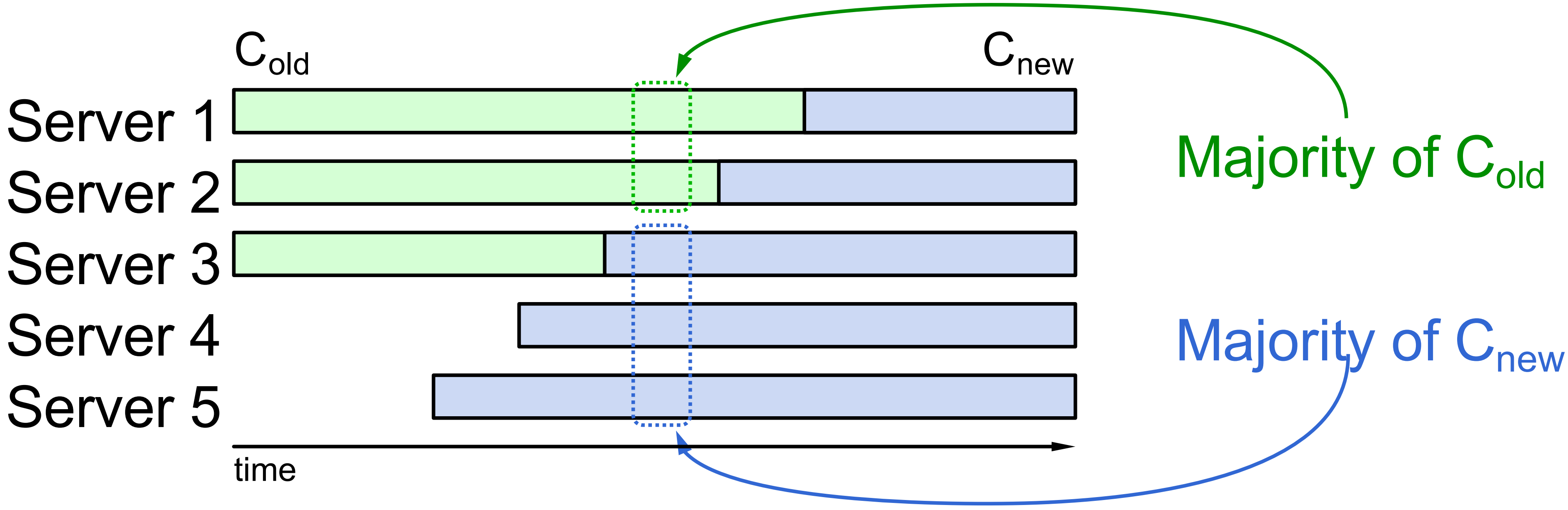
- What if leader crashes after executing command, but before responding?
  - Must not execute command twice
- Solution: client embeds a unique id in each command
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - If id found in log, ignore new command, return response from old command
- Result: exactly-once semantics as long as client doesn't crash

Configuration Changes

- System configuration:
  - ID, address for each server
  - Determines what constitutes a majority
- Consensus mechanism must support changes in the configuration:
  - Replace failed machine
  - Change degree of replication

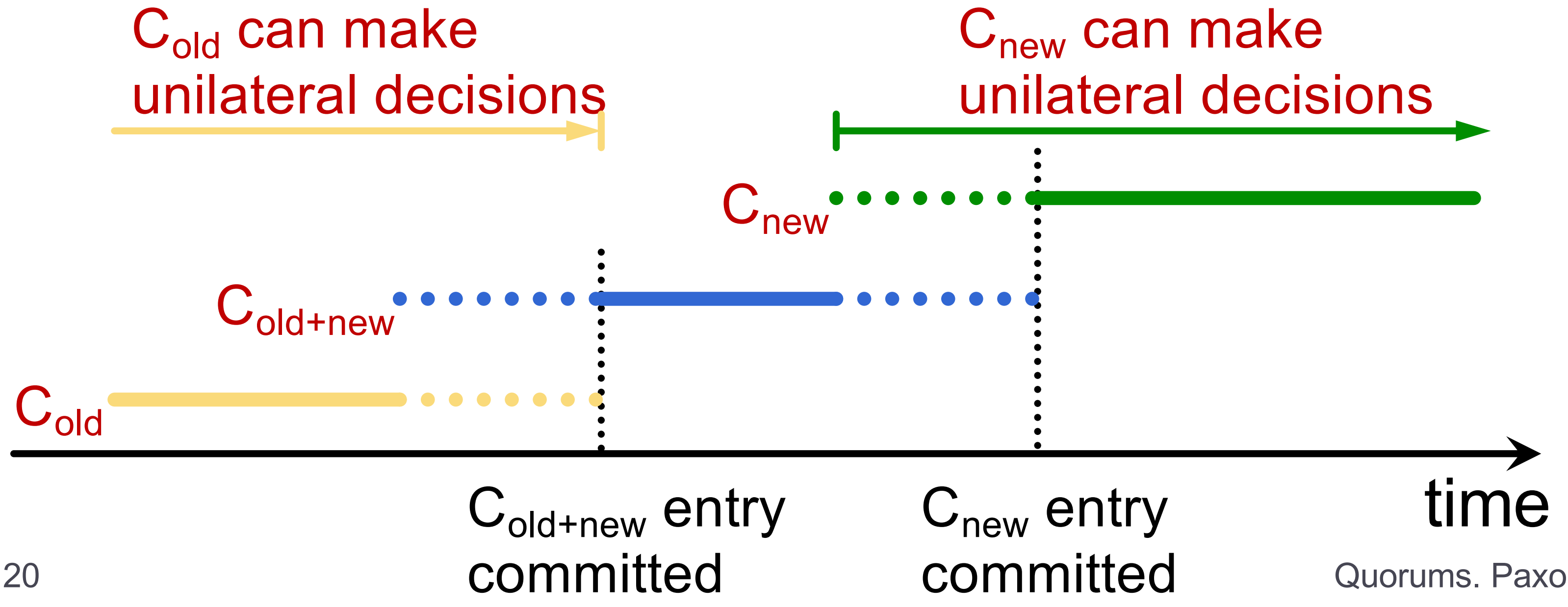
Configuration Changes, cont'd

Cannot switch directly from one configuration to another: conflicting majorities could arise



Joint Consensus

- Raft uses a 2-phase approach:
  - Intermediate phase uses joint consensus (need majority of both old and new configurations for elections, commitment)
  - Configuration change is just a log entry; applied immediately on receipt (committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration



C_{old} can make unilateral decisions

C_{new} can make unilateral decisions

C_{new}

$C_{old+new}$

C_{old}

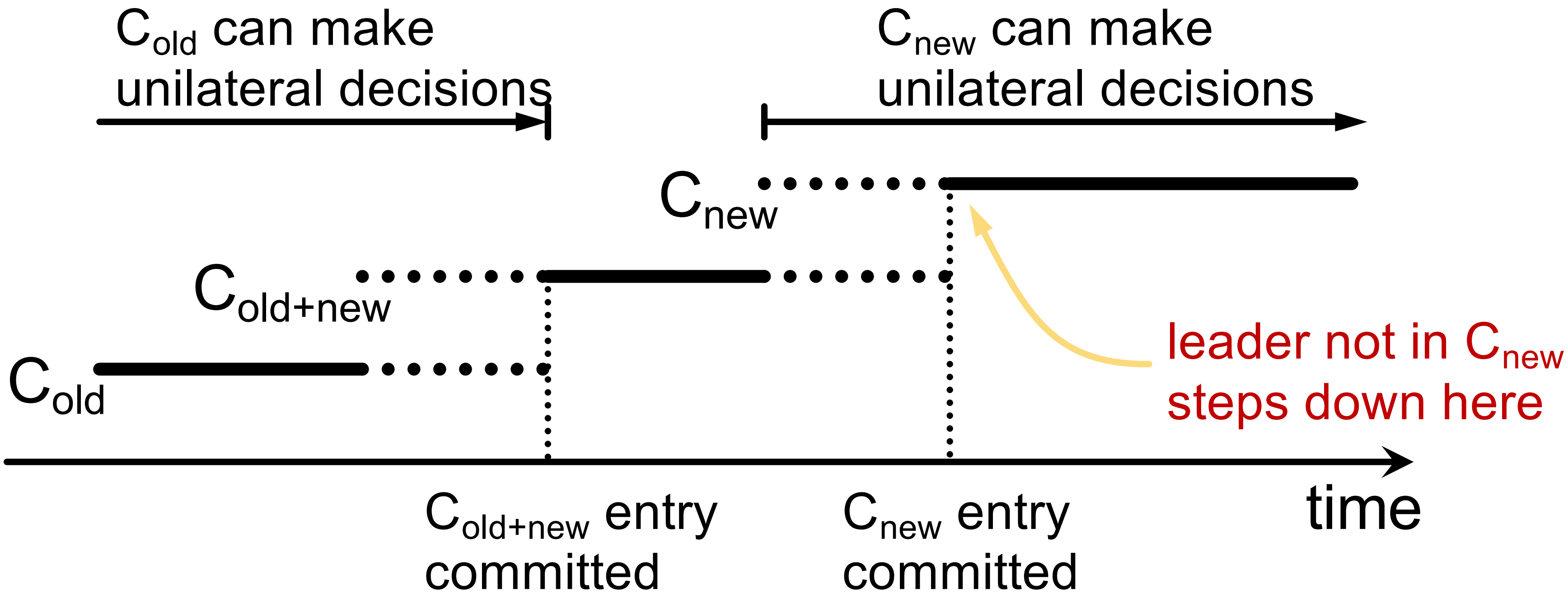
$C_{old+new}$ entry committed

C_{new} entry committed

time

Joint Consensus, cont'd

- Additional details:
  - Any server from either configuration can serve as leader
  - If current leader is not in C_{new}, must step down once C_{new} is committed.


C_{old} can make unilateral decisions
C_{new} can make unilateral decisions
C_{new}
$C_{old+new}$
C_{old}
leader not in C_{new} steps down here
$C_{old+new}$ entry committed
C_{new} entry committed
time