CS 795: Distributed Systems & Cloud Computing

Fall 2018

Lec 2: Distributed storage implementation & Consensus algorithms

Yue Cheng

Announcements

- Paper presentation schedule is out on course website
- Please sign-up for the paper scribes

Distributed key-value (KV) stores

- Interface
  - **put**(key, value); // insert/write “value” assoc. with “key”
  - value = **get**(key); // get/read data assoc. with “key”
- Abstraction used to implement
  - File systems: value content —> block
  - Sometimes as a simpler but more scalable “database”
- Can handle large volumes of data, e.g., PBs
  - Need to distribute data over hundreds, even thousands of machines

KV examples

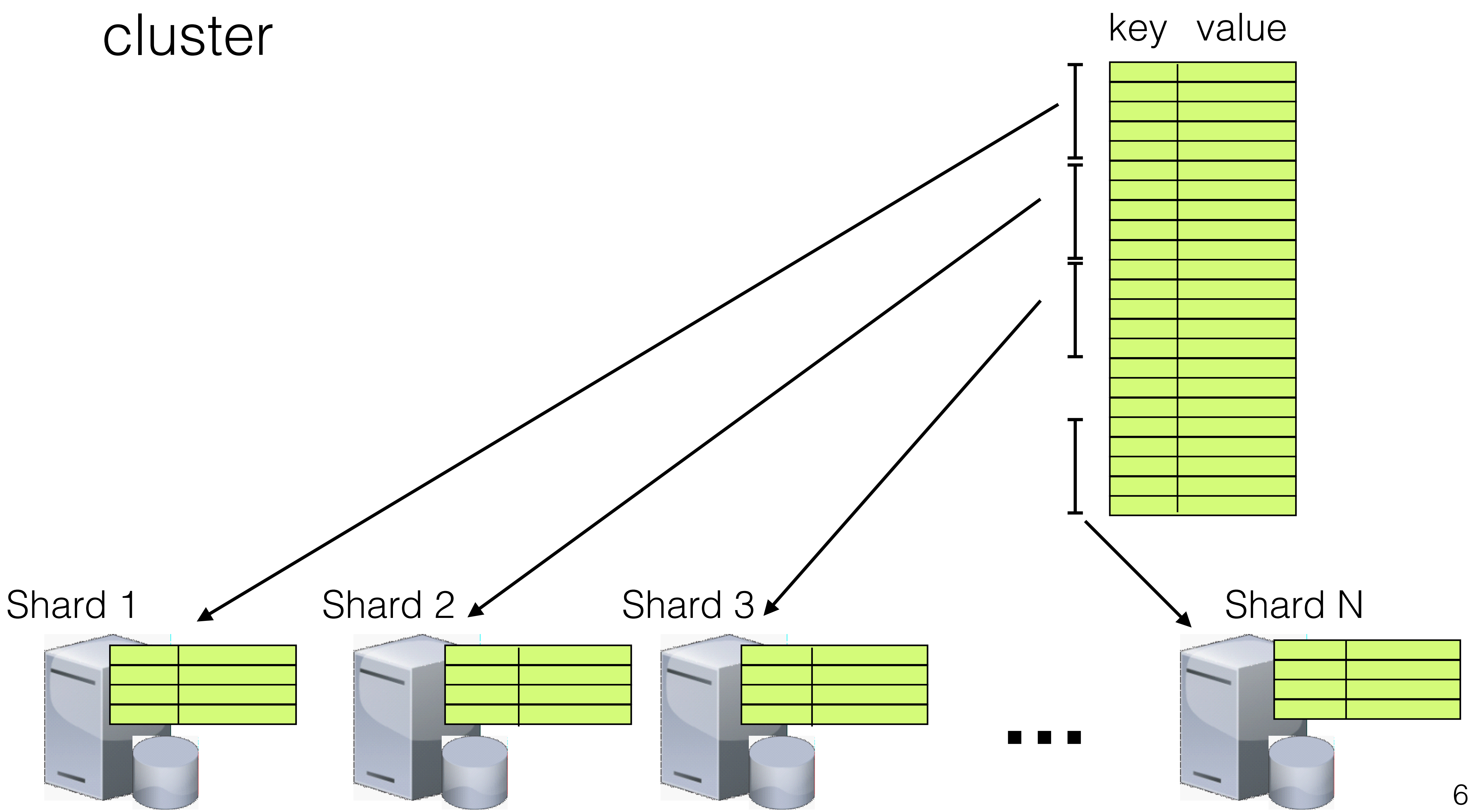
- Amazon
  - Key: CustomerID
  - Value: Customer profile (e.g., buying history, credit card, etc.)
- Facebook, Twitter
  - Key: UserID
  - Value: User profile (e.g., posting history, photos, friends, etc.)
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song file
- Distributed file systems
  - Key: BlockID
  - Value: Block

KV storage system examples

- **Google File Systems (GFS), Hadoop Distributed File System (HDFS)**
- **Amazon**
  - Dynamo: distributed KV store used to power the shopping cart in amazon.com
  - Simple Storage Service (S3)
- **Bigtable/HBase**: distributed NoSQL data store
- **Memcached/Redis**: distributed in-memory KV stores for small values (arbitrary strings)

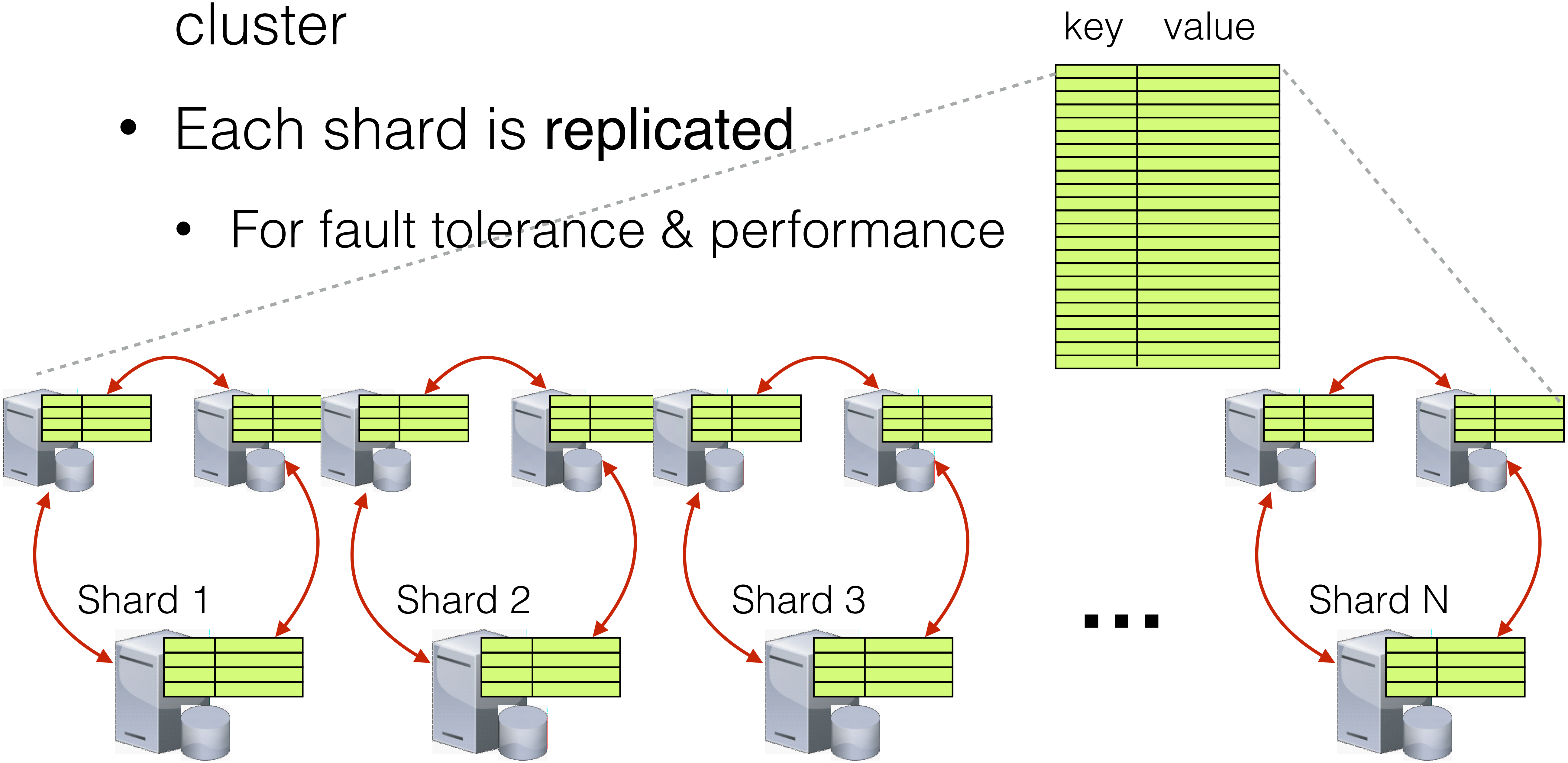
Data partitioning (sharding)

- Main idea: partition set of key-value data across many machines to form a **scale-out** data storage cluster



Data partitioning (sharding)

- Main idea: partition set of key-value data across many machines to form a **scale-out** data storage cluster
- Each shard is **replicated**
  - For fault tolerance & performance



Desired properties of a replicated KV store?

- **Scalability:** Horizontal scalability
    - Need to scale to thousands of machines
    - Need to allow easy addition of new machines

- **Consistency:** Maintain data consistency in face of node failures and message losses

- **Fault tolerance:** Handle machine failures without losing data and without degradation in performance

Key questions of implementation

- **put**(key, value): where does the system store a new key-value tuple?
- **get**(key): how does the system route the read request with a given “key”?

- And, do the above while providing:
  - Scalability
  - Consistency
  - Fault tolerance

Case study: BespoKV*

- BespoKV is a **versatile** distributed key-value store that decouples the control and data plane:
  - To support configurable data consistency, network topology, and fault tolerance
  - To support configurable backend data structures (how data is organized in storage medium)

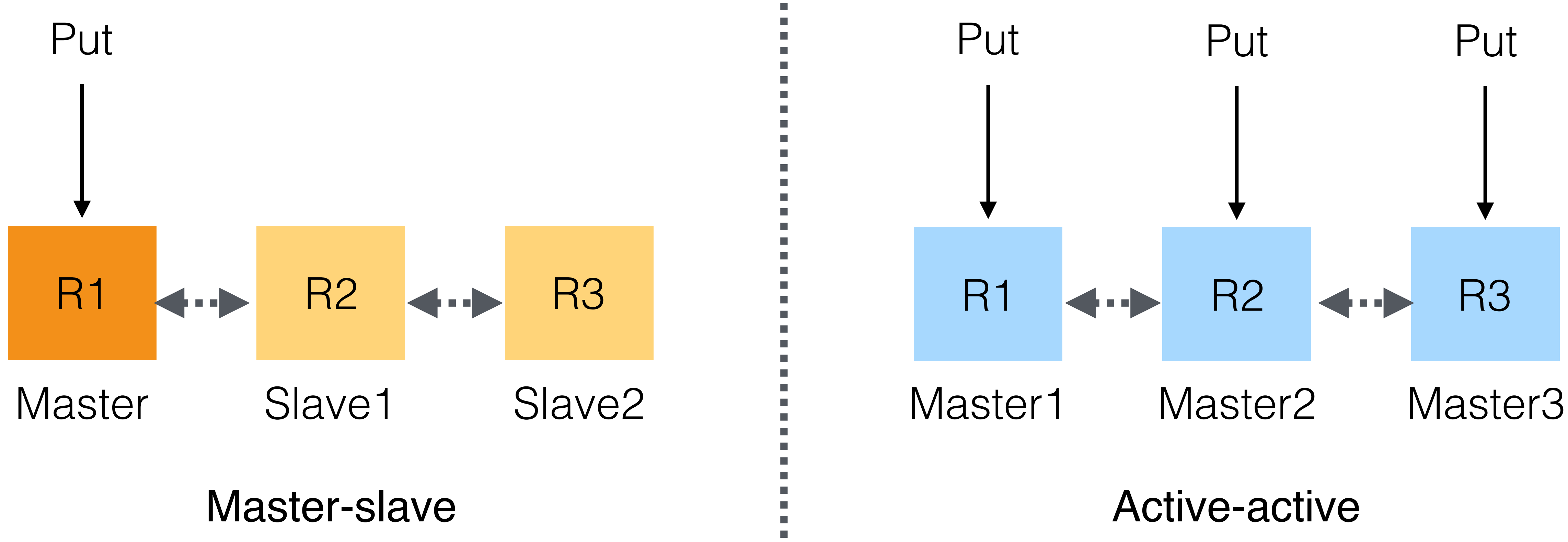
- Programmable **controlets:** responsible for distributed system management
- Pluggable **datalets:** responsible for managing local data storage

*: BespoKV: Application Tailored Scale-out Key-Value Stores [IEEE SC '18]

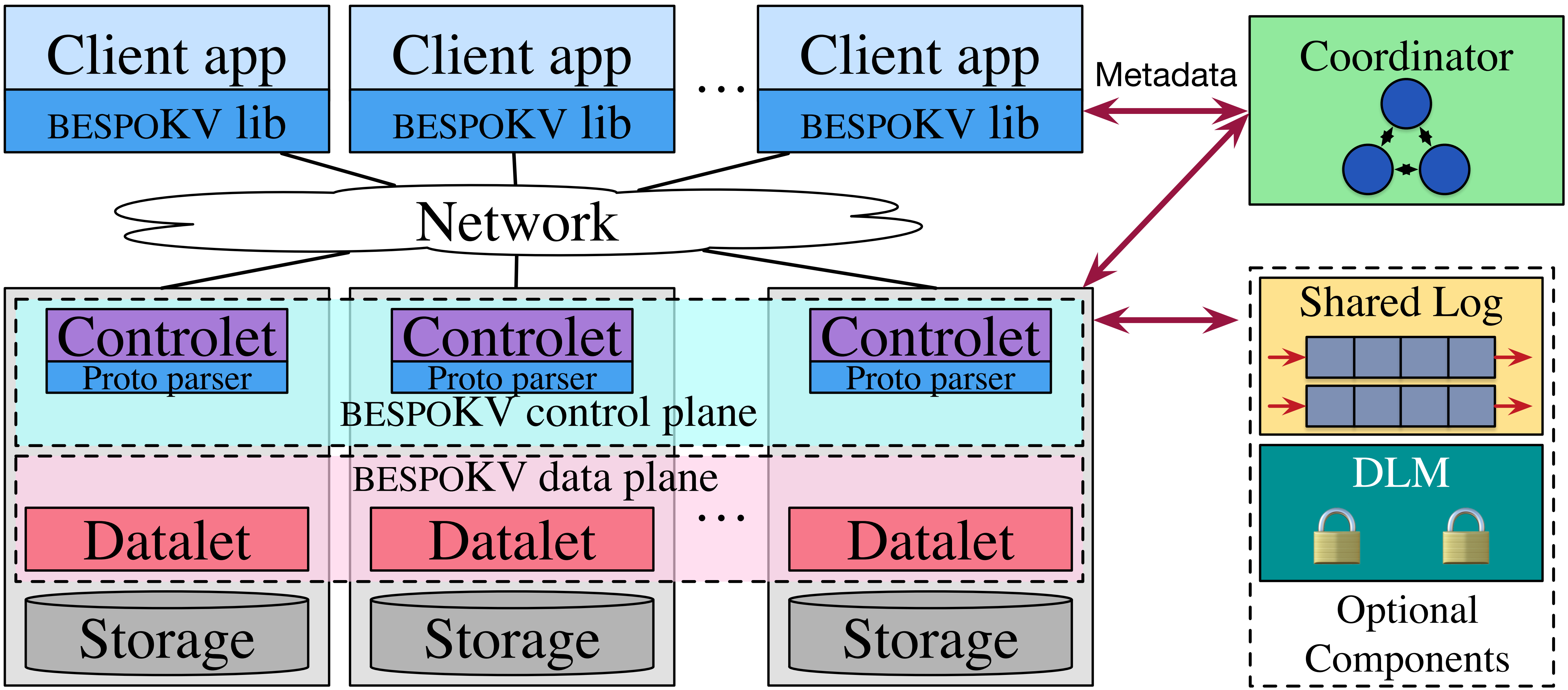
Configurable consistency levels & network topologies

- Consistency levels: Strong consistency (SC) / eventual consistency (EC)

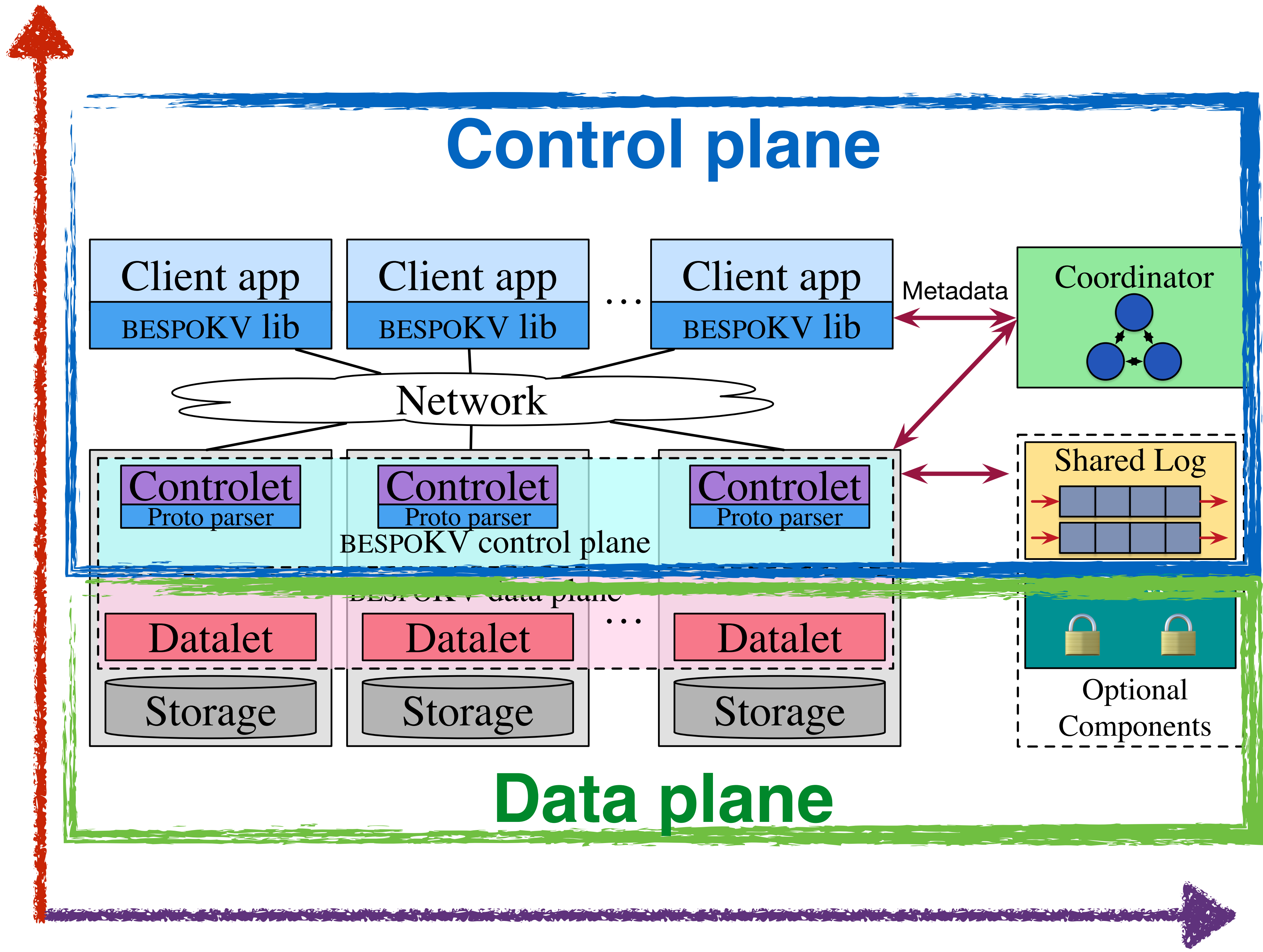
- Network topologies: Master-slave (MS) / active-active (AA)



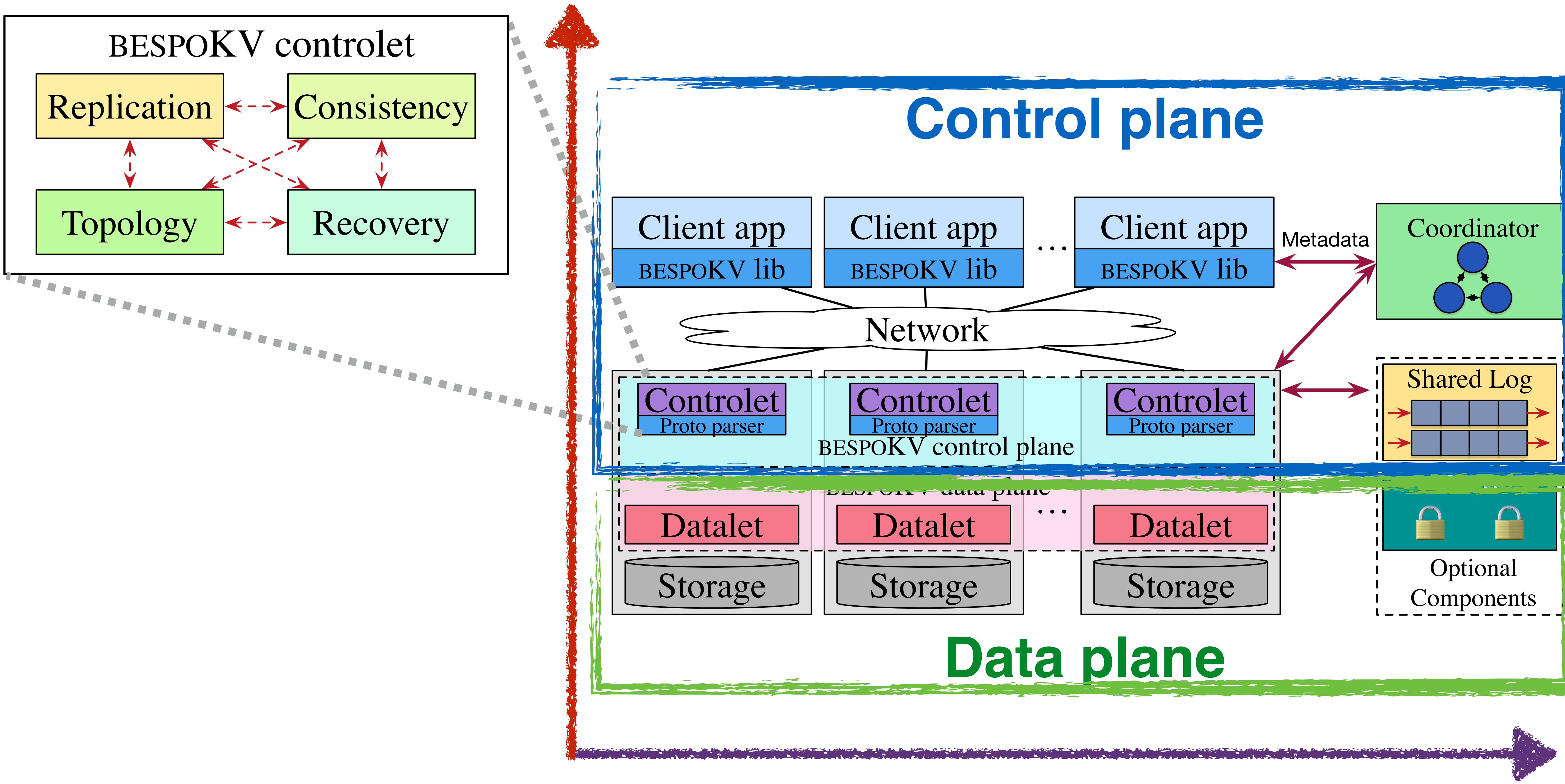
BespoKV overview



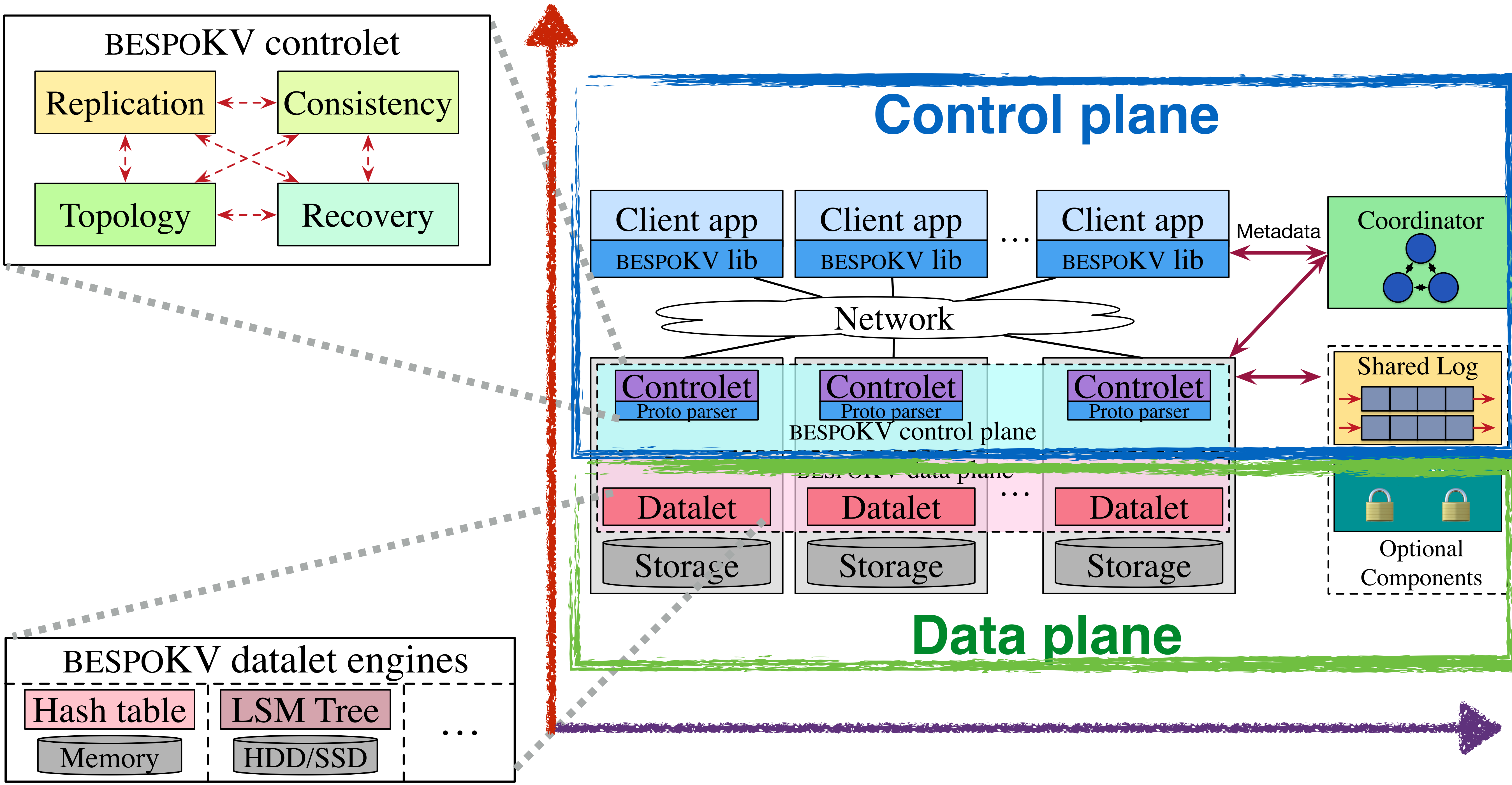
BespoKV's 2D architecture



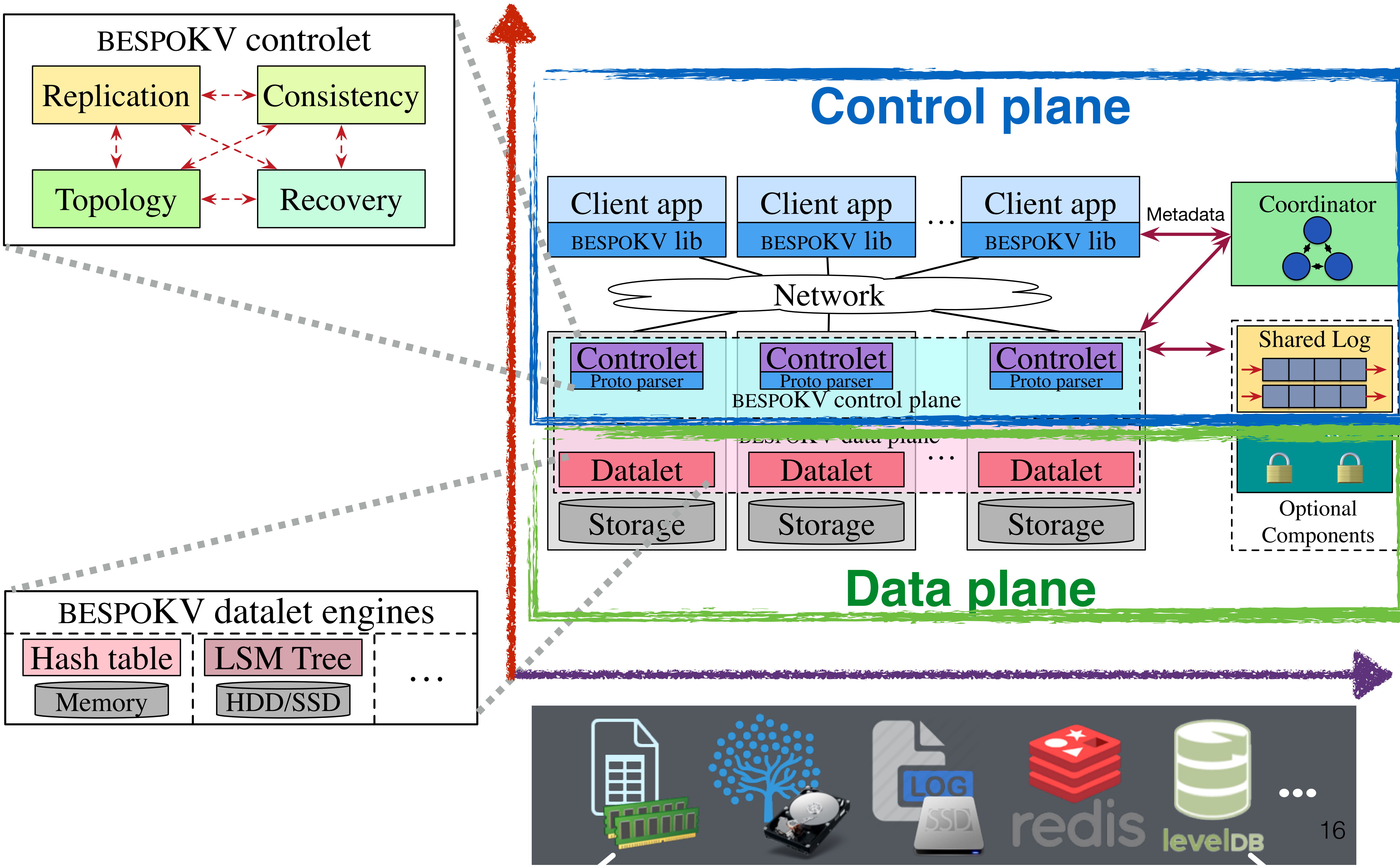
BespoKV's 2D architecture



BespoKV's 2D architecture



BespoKV's 2D architecture

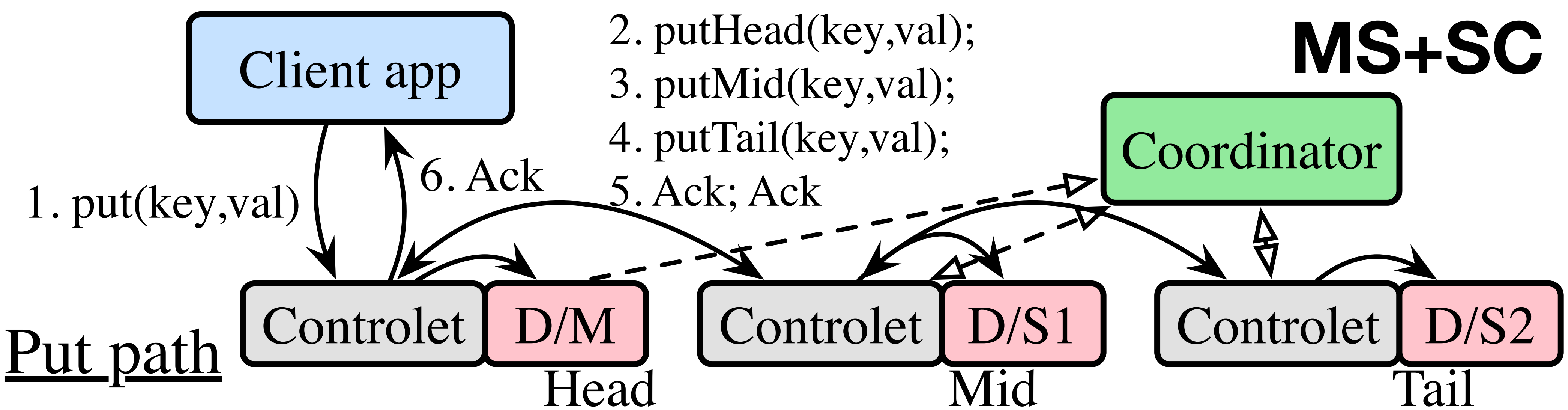


BespoKV API

- **Datalet API**: provided by datalet app developers
  - put(key, value)
  - value = get(key)
  - delete(key)
- **Client API**: provided by BespoKV
  - createTable(T)
  - put(key, value, T)
  - value = get(key, T)
  - delete(key, T)
  - deleteTable(T)

Supporting SC+MS

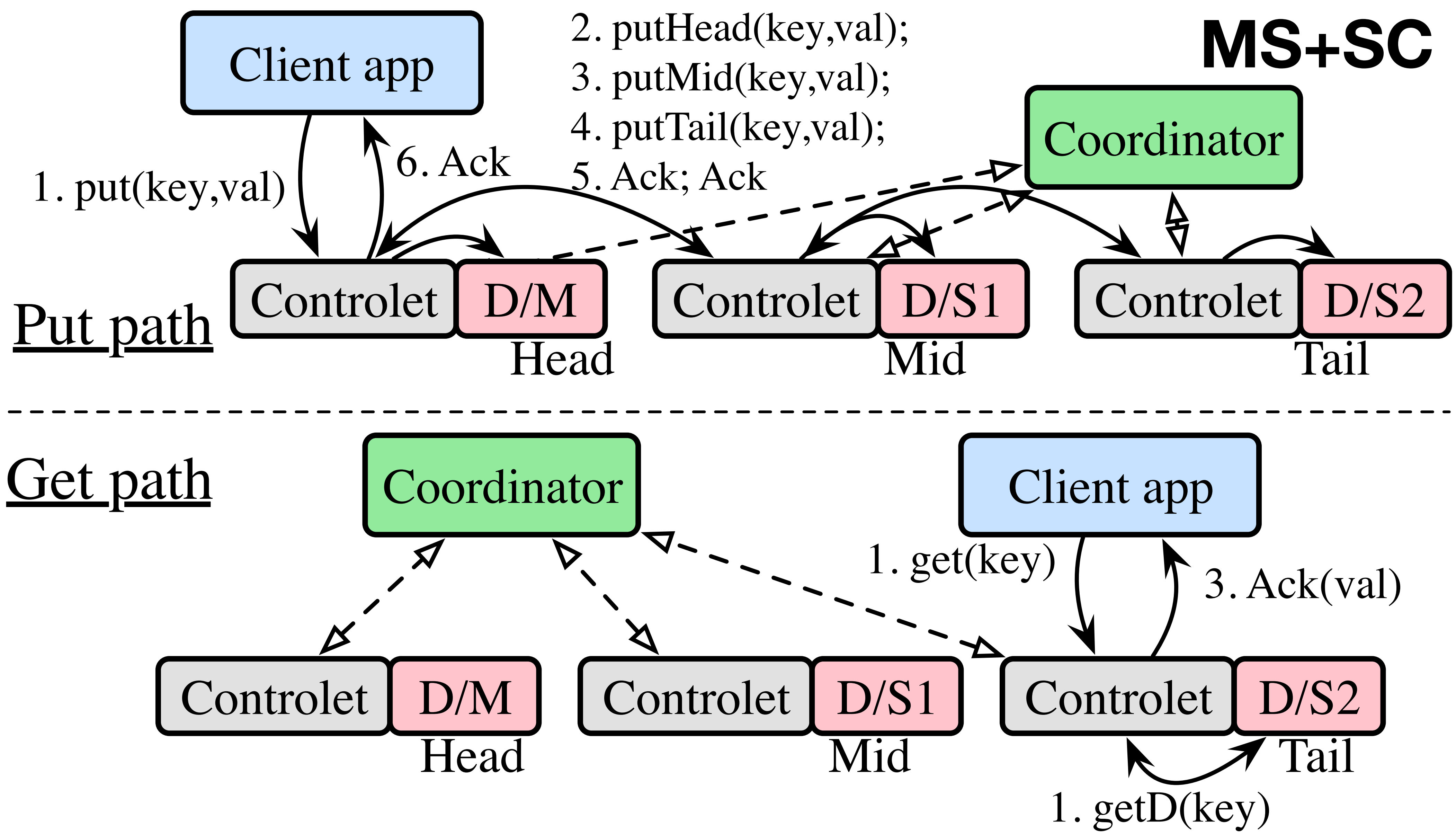
- Based on chain replication*



*: Chain replication for supporting high throughput and availability [USENIX OSDI 04]

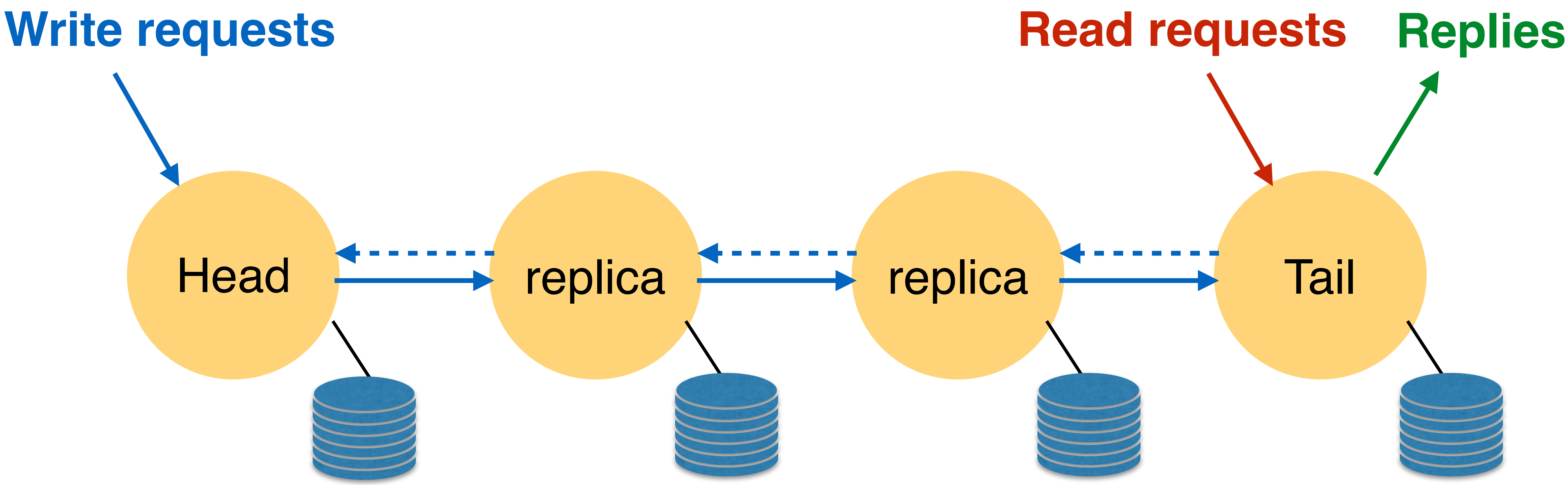
Supporting SC+MS

- Based on chain replication*



*: Chain replication for supporting high throughput and availability [USENIX OSDI '04]

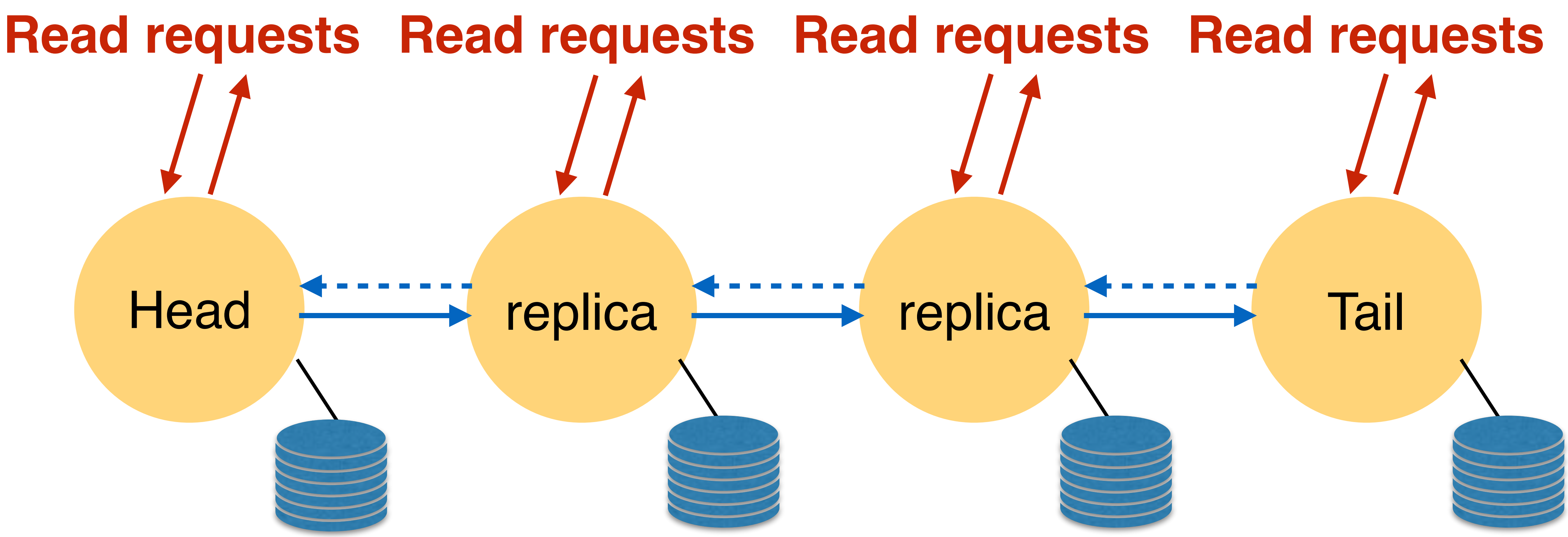
Chain replication*



- Writes to head, which orders all writes
- When write reaches tail, implicitly committed rest of chain
- Reads to tail, which orders reads w.r.t. committed writes
- Replies to both writes/reads from tail

*: Chain replication for supporting high throughput and availability [USENIX OSDI ’04]

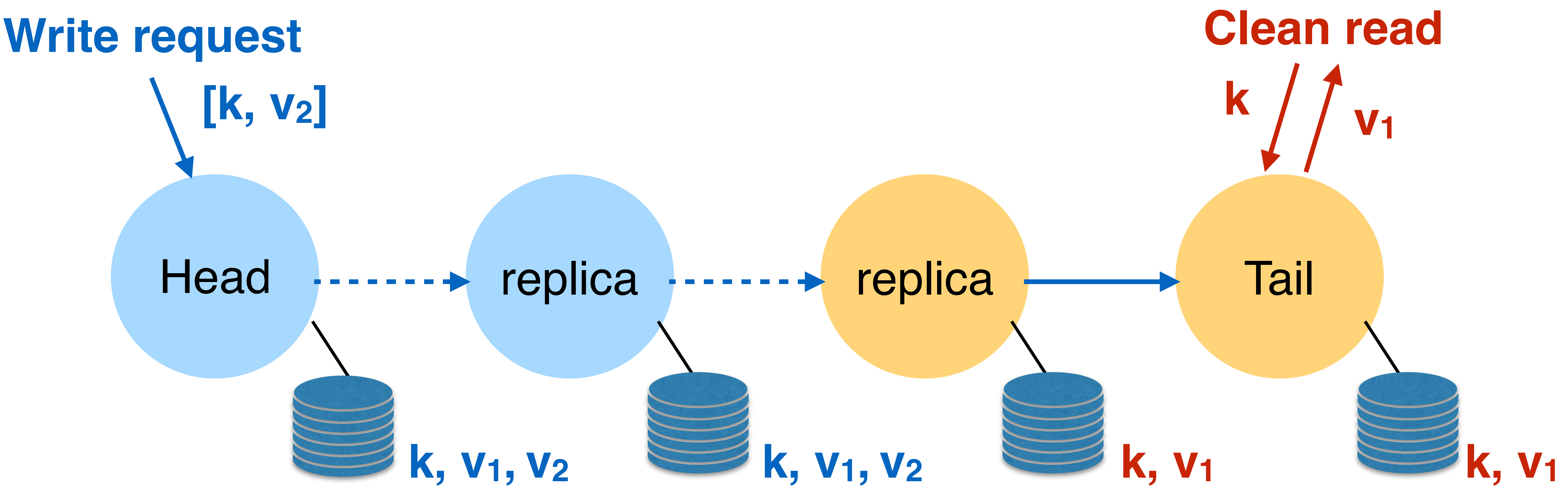
Chain replication for read-heavy* (CRAQ)



- Goal: If all replicas have same version, read from any one
- Challenge: They need to know they have correct version

*: Object storage on CRAQ: High-throughput chain replication for read-mostly workloads [USENIX ATC '09]

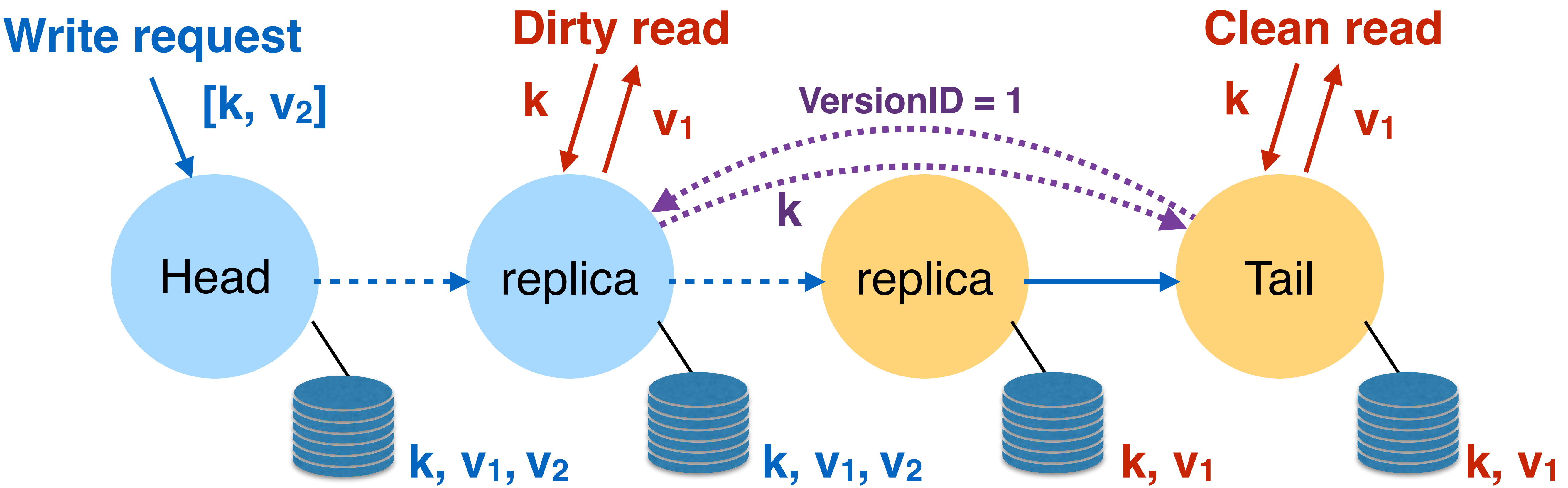
Chain replication for read-heavy* (CRAQ)



- Replicas maintain multiple versions of objects while "dirty", i.e., contain uncommitted writes
- Commitment sent "up" chain after reaches tail

*: Object storage on CRAQ: High-throughput chain replication for read-mostly workloads [USENIX ATC '09]

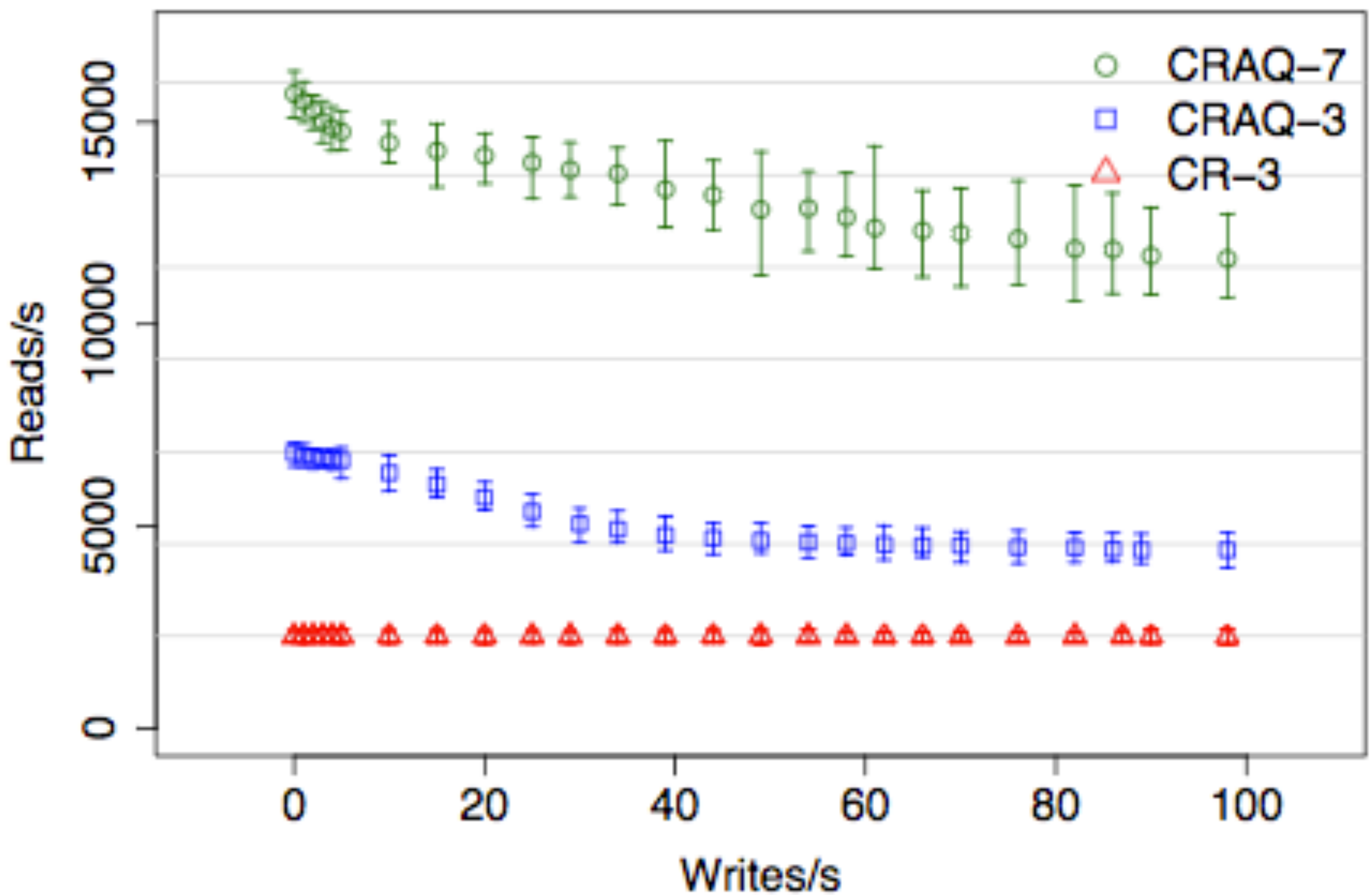
Chain replication for read-heavy* (CRAQ)



- Reads to dirty object must check with tail for proper version
- This orders read with respect to global order, regardless of replica that handles

*: Object storage on CRAQ: High-throughput chain replication for read-mostly workloads [USENIX ATC '09]

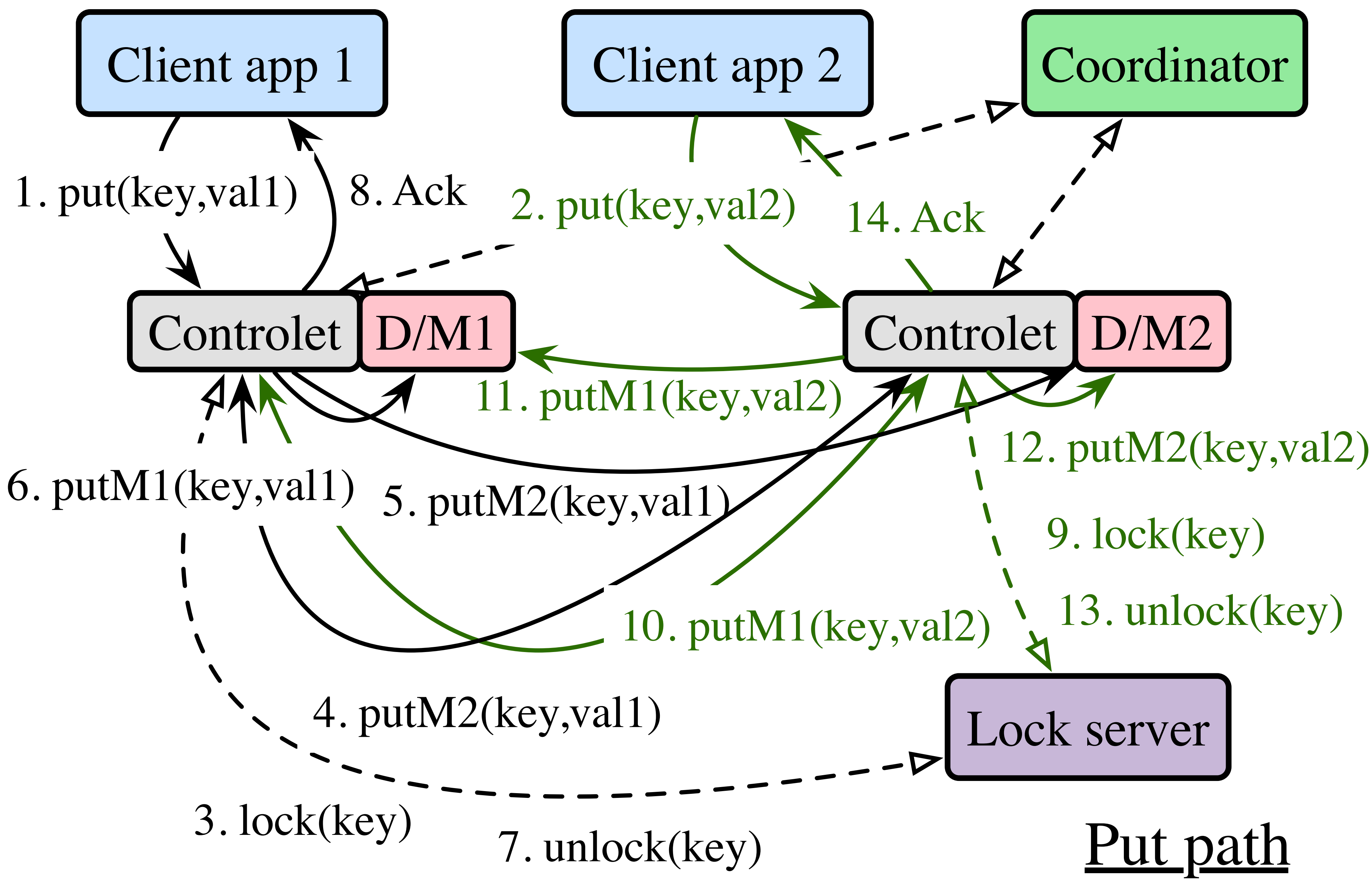
Chain replication for read-heavy* (CRAQ)



*: Object storage on CRAQ: High-throughput chain replication for read-mostly workloads [USENIX ATC '09]

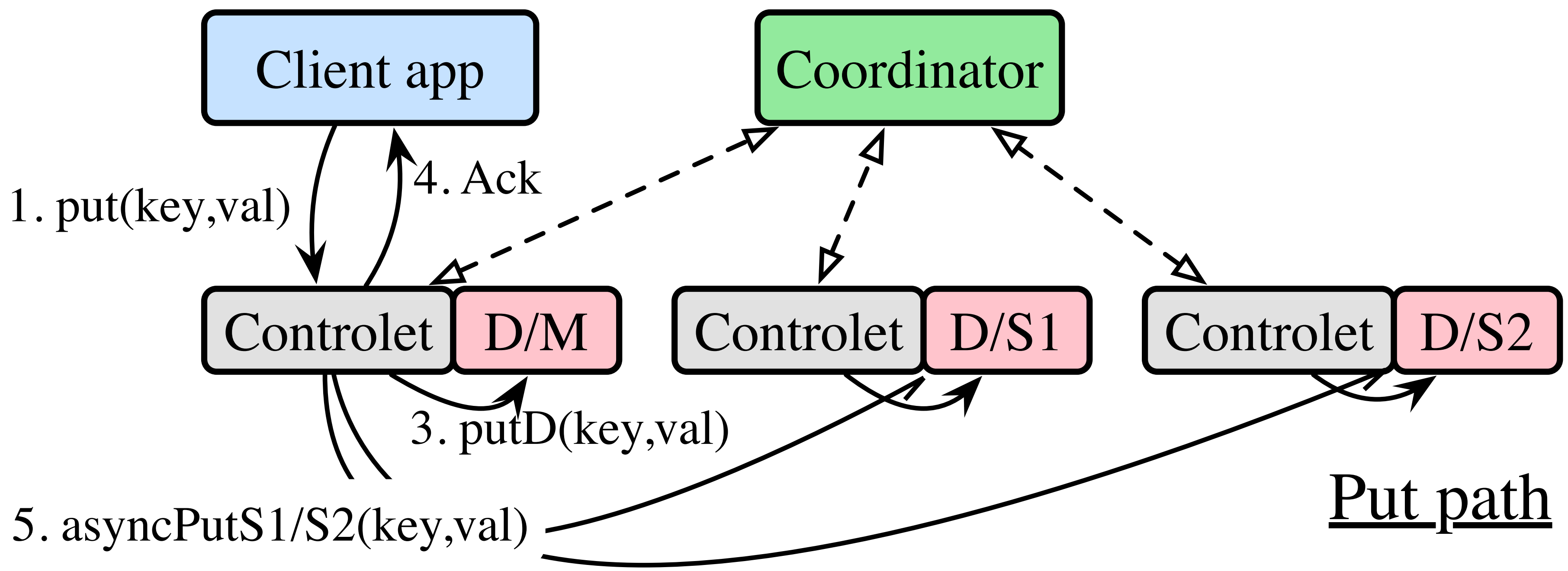
Supporting SC+AA

- Leverage a distributed lock server



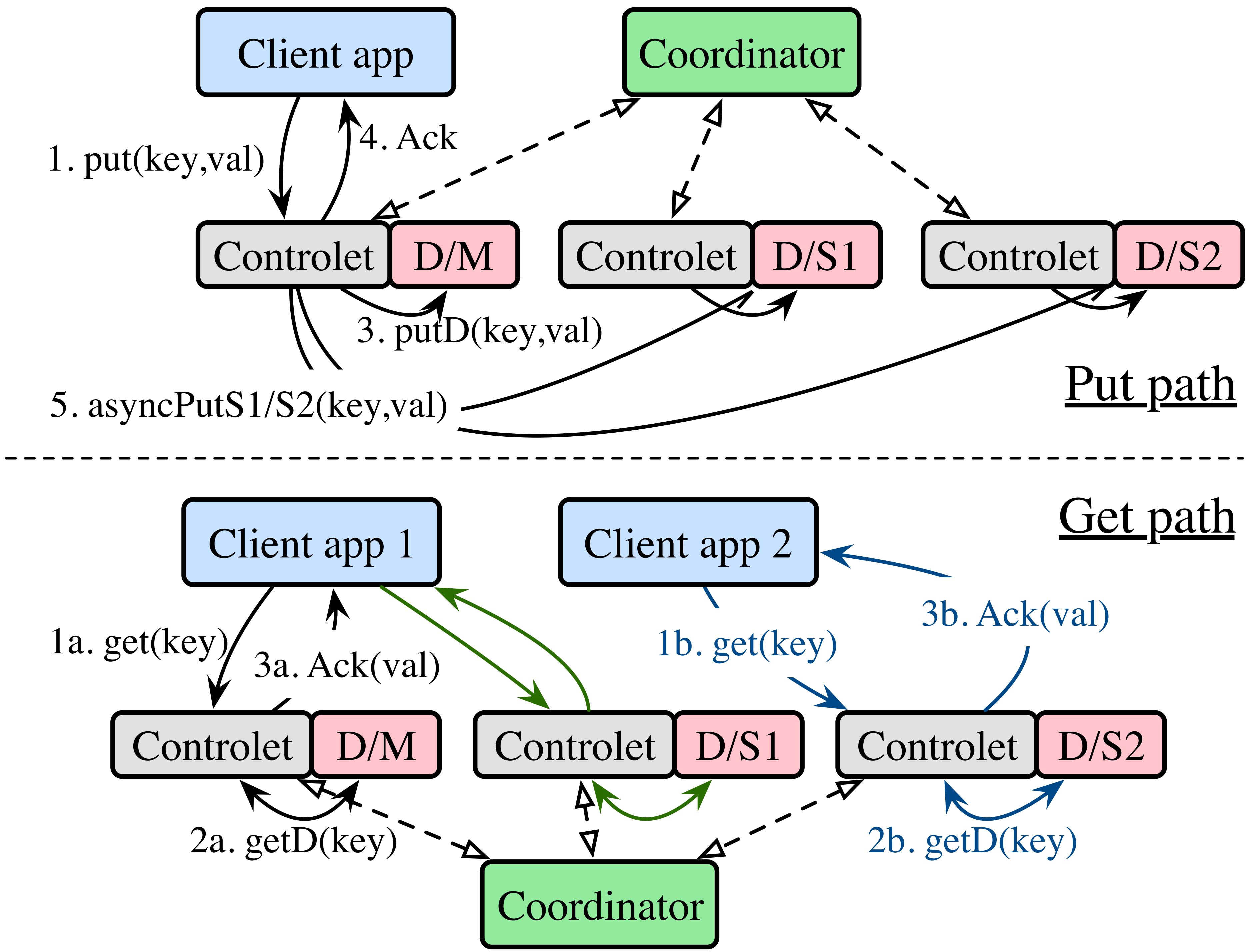
Supporting EC+MS

- Asynchronous writes to slave replicas



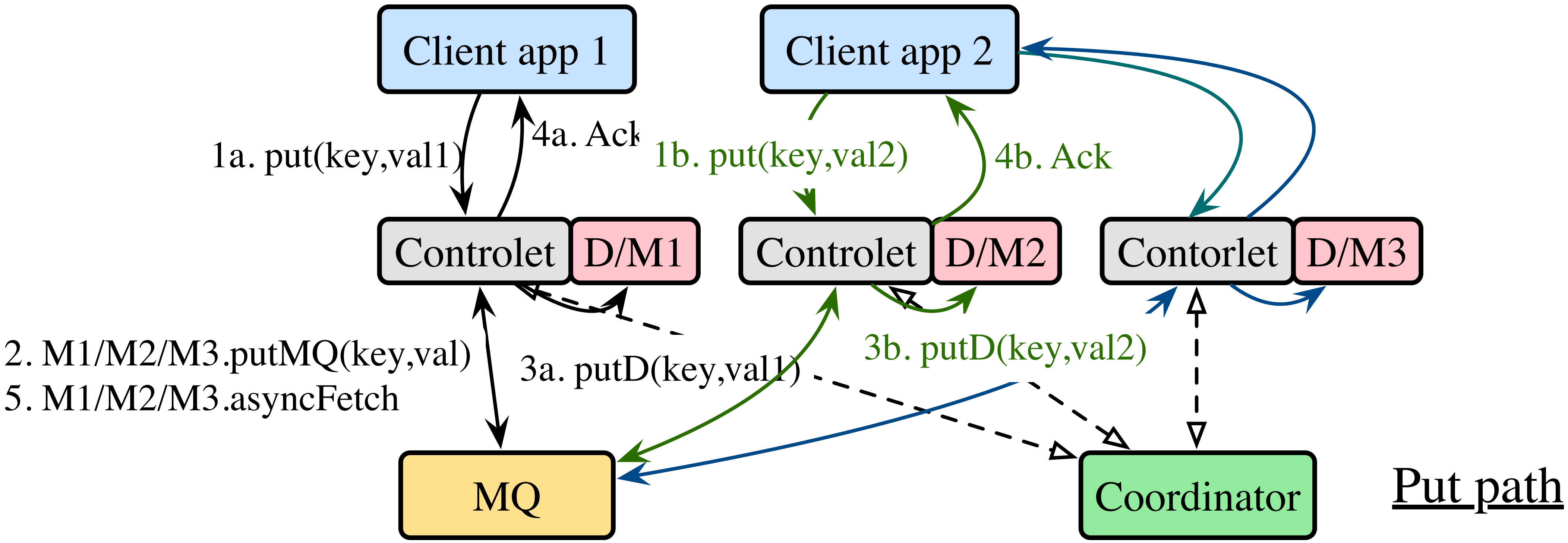
Supporting EC+MS

- Asynchronous writes to slave replicas



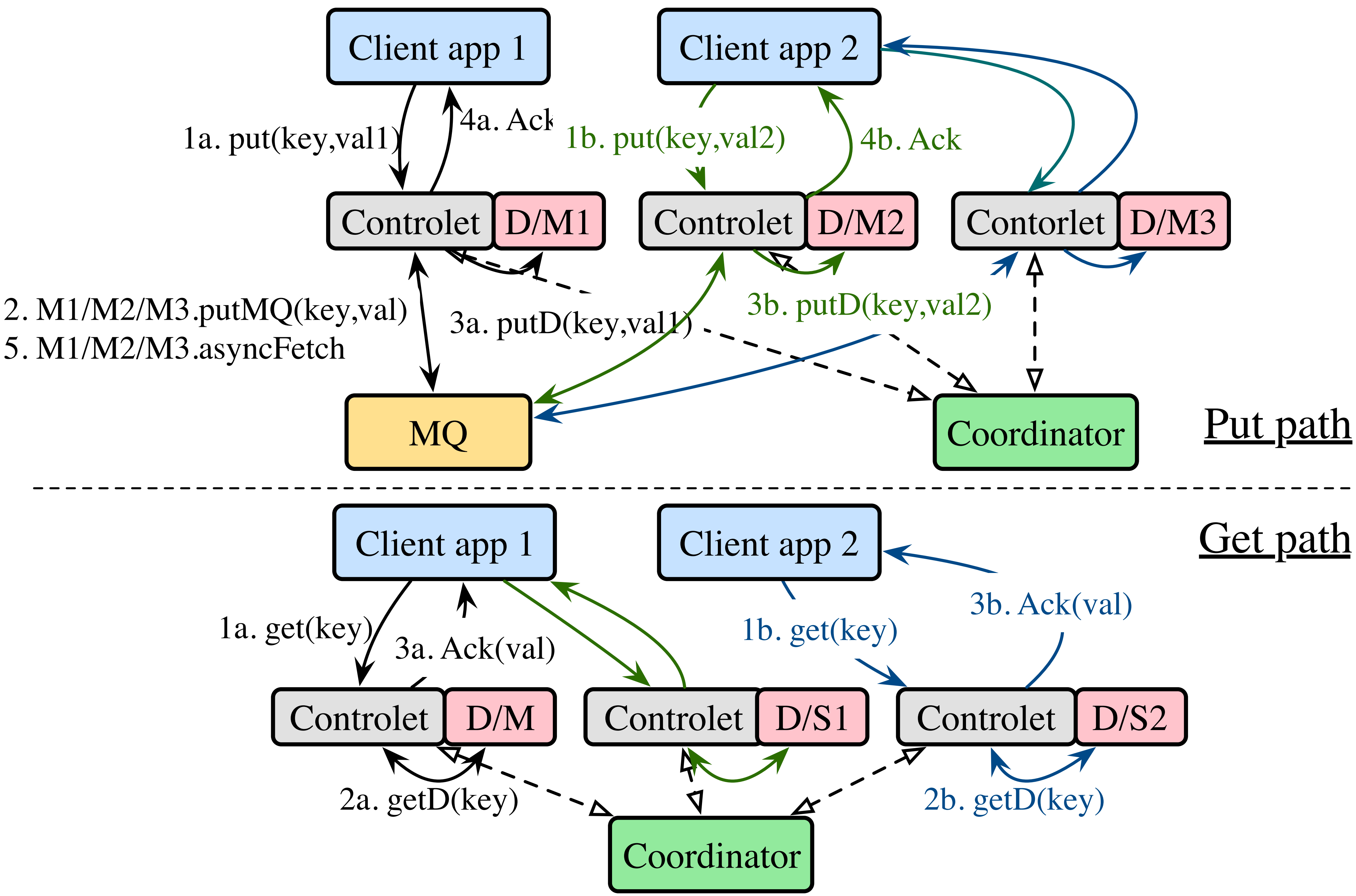
Supporting EC+AA

- Leverage a distributed message queue for multi-master asynchronous writes

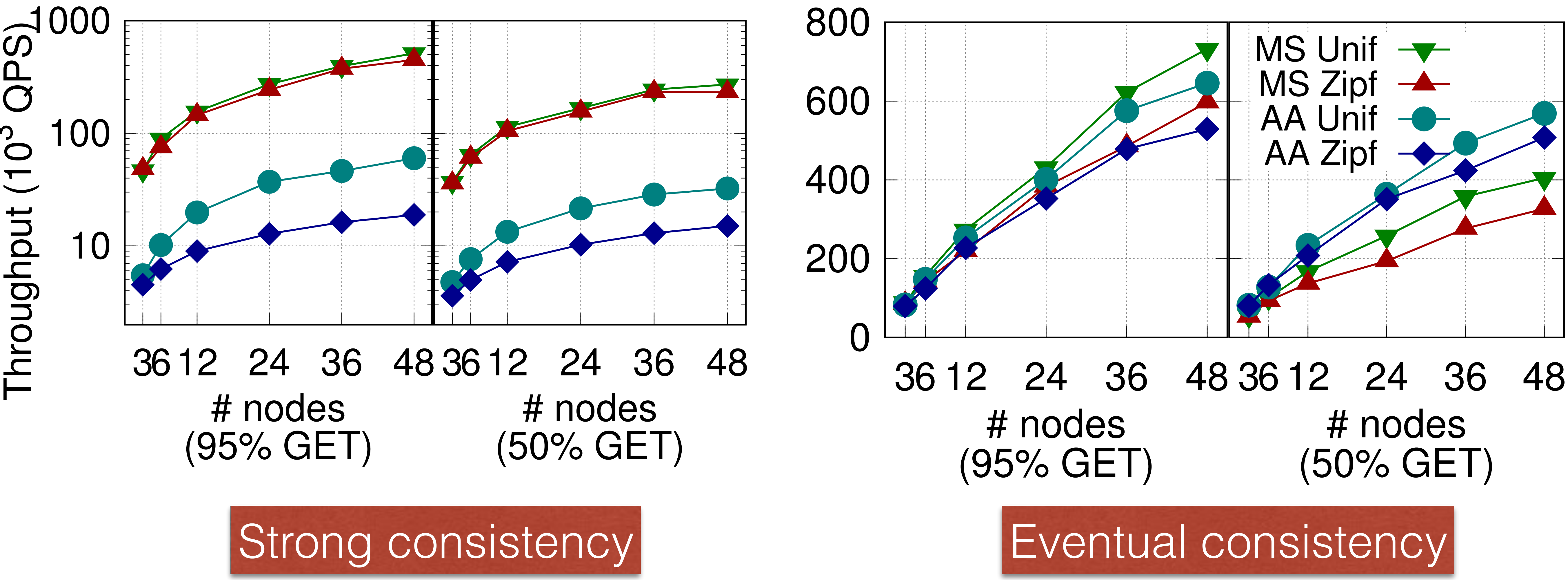


Supporting EC+AA

- Leverage a distributed message queue for multi-master asynchronous writes

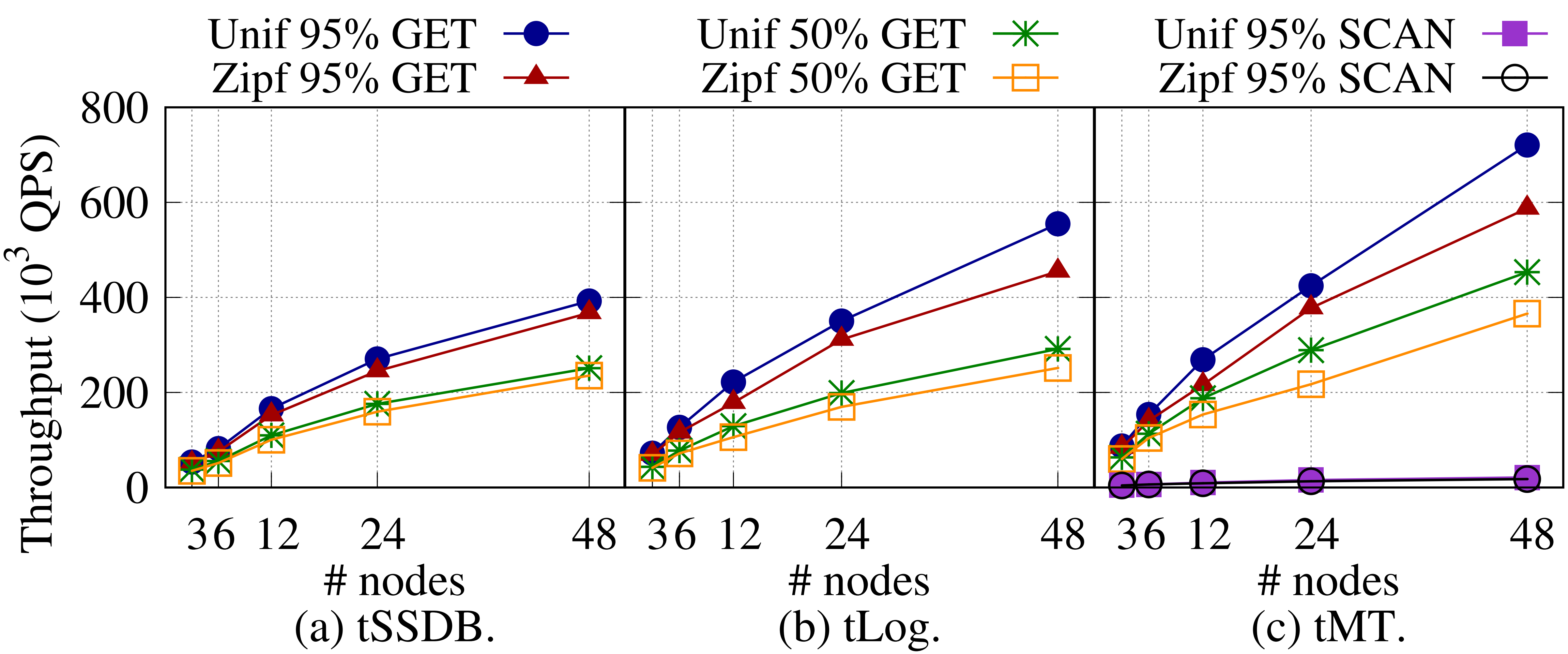


Horizontal scalability on GCP



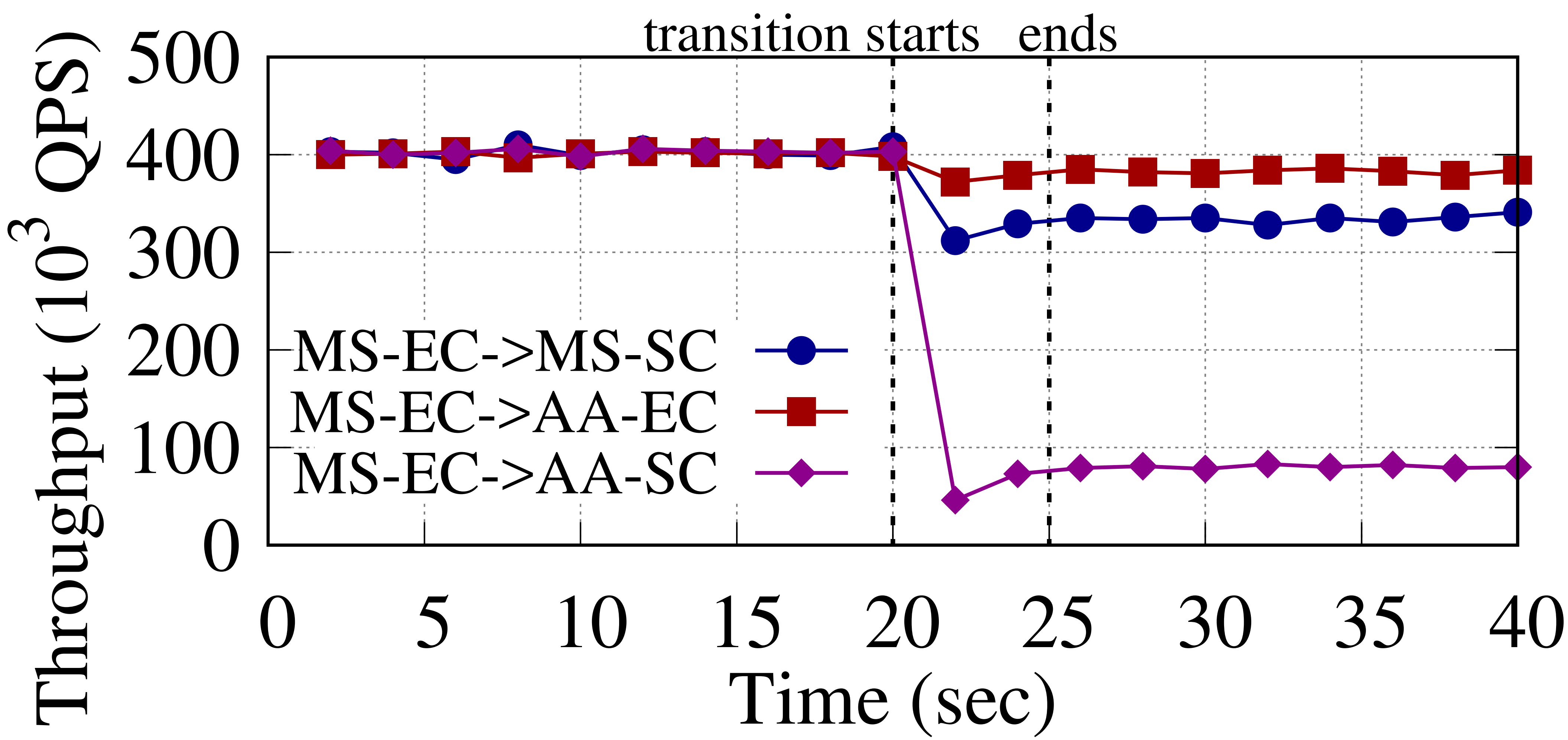
- Workloads: Yahoo! Cloud Service Benchmark
- Each shard has 3 replicas
- Google cloud platform: scaled from 3 VMs to 48 VMs

Data engine flexibility: Varying backend datalets



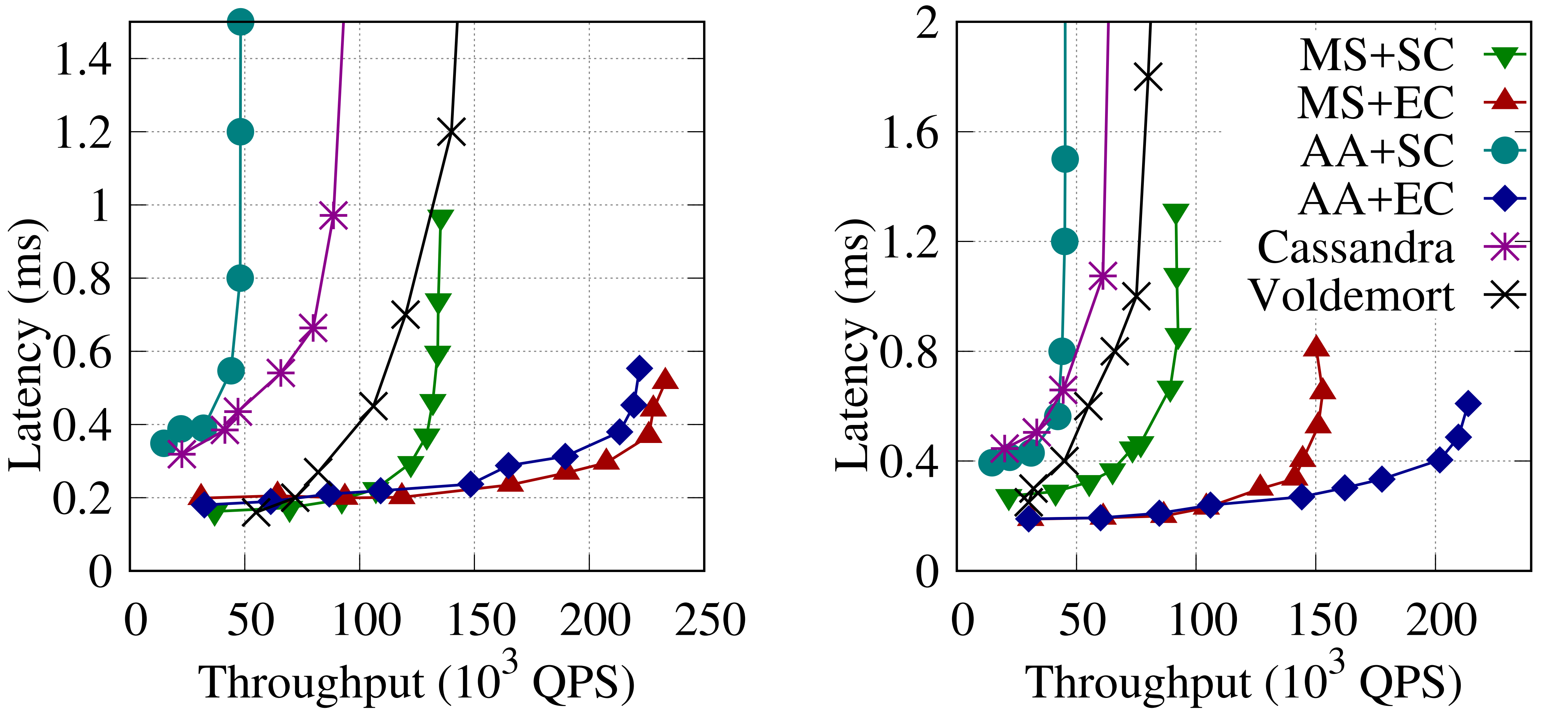
- Workloads: Yahoo! Cloud Service Benchmark
- Each shard has 3 replicas
- Google cloud platform: scaled from 3 VMs to 48 VMs

Service flexibility: Online reconfigurability



- Workloads: Yahoo! Cloud Service Benchmark
- Each shard has 3 replicas
- BespoKV seamlessly adapts service from MS-EC to MS-SC, AA-EC, and AA-SC

Performance comparison



(a) 95% Get.

(b) 50% Get.

- Workloads: Yahoo! Cloud Service Benchmark
- Each shard has 3 replicas
- Comparing against Cassandra and Voldemort

Consensus

- Definition
  - A general agreement about something
  - An idea or opinion that is shared by all the people in a group

Distributed consensus algorithms

- Consensus of a set of processes (i.e., a distributed system)
    - **Termination:** All non-faulty processes eventually decide on a value
    - **Agreement:** All processes that decide to do so on the same value
    - **Validity:** The value that has been decided must have proposed by some process

Assumptions

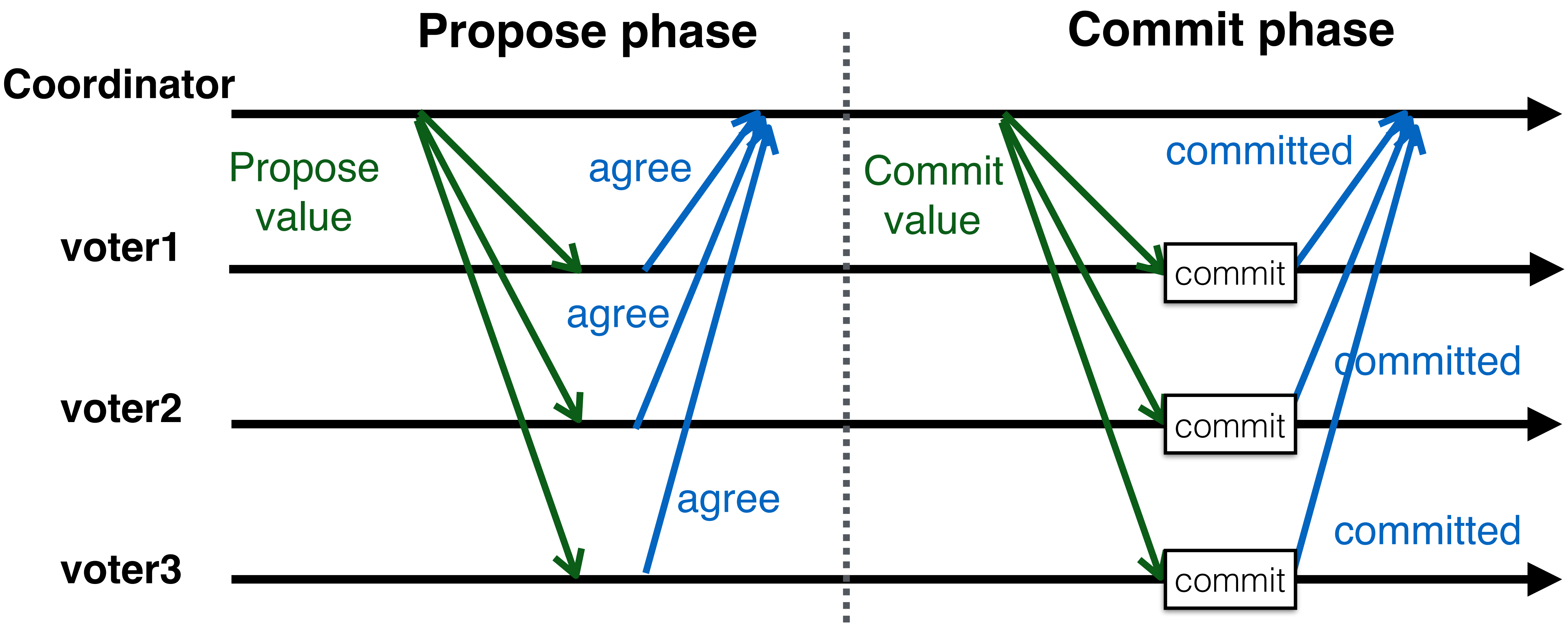
- Failure models
  - Fail-stop
  - Fail-recover

- Asynchronous distributed systems
  - Delayed/dropped messages
  - No upper bound on clock drift rate
  - No synchronization among processes

Consensus used in systems

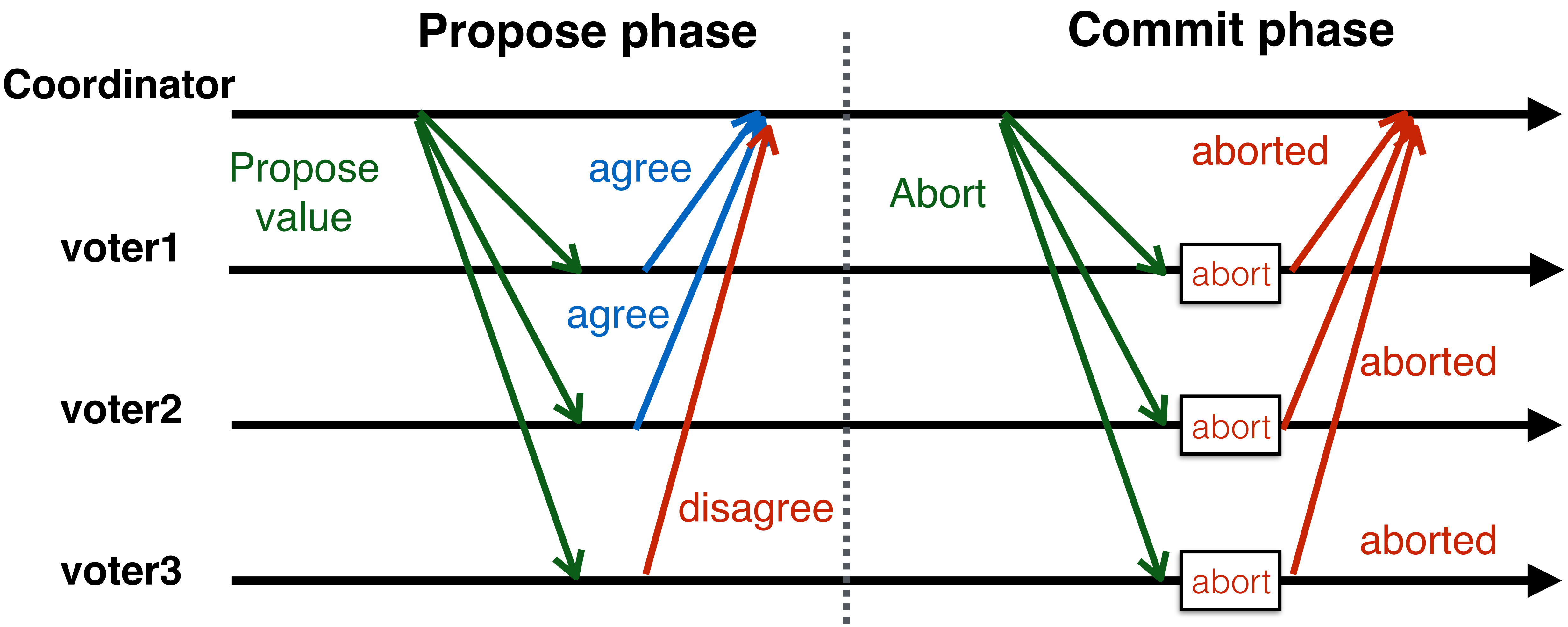
- Deciding whether or not to commit a transaction to a database
- Synchronizing clocks by agreeing on the current time
- Making sure all servers in group receive the same commands (or data) in the same order as each other
  - The famous replicated state machine approach
- Electing a leader node to coordinate some higher-level protocol

Two-phase commit (2PC)



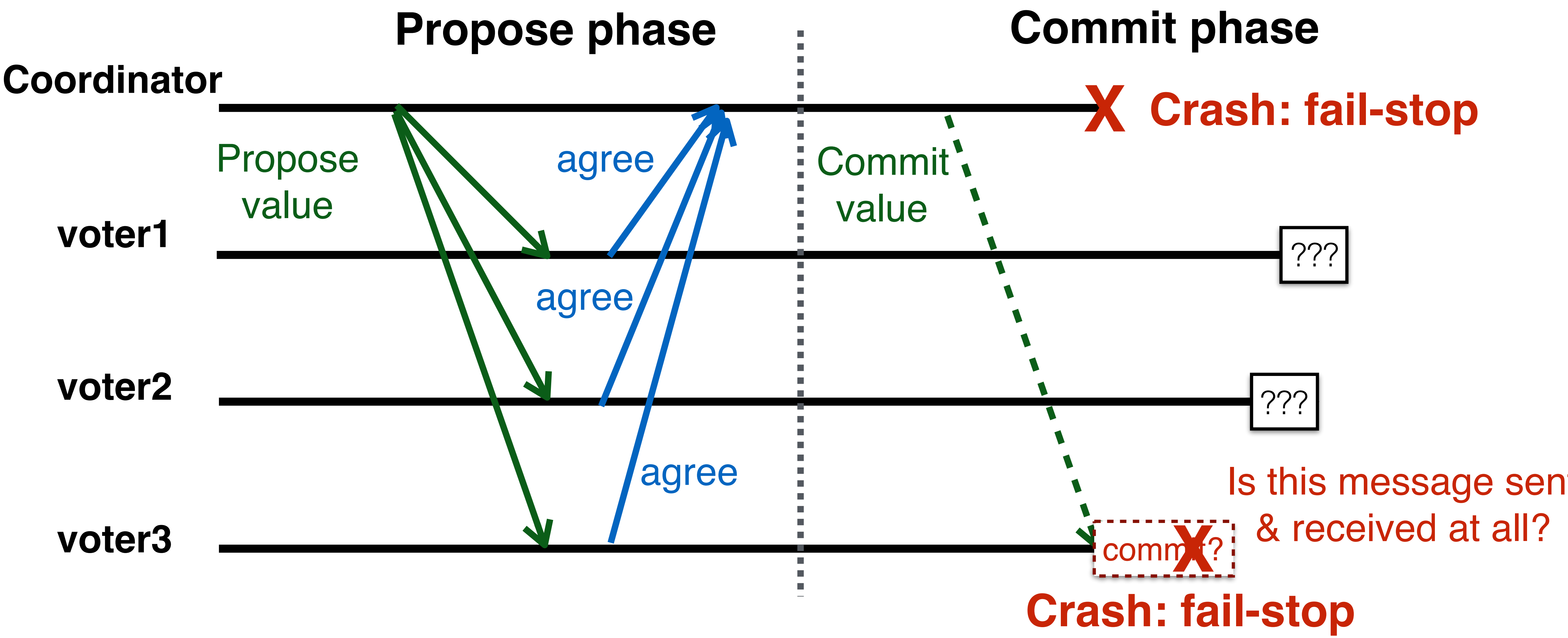
- Simple and natural: **two** phases
    1. **Propose:** Contact every participant, suggest a value and gather their responses
    2. **Commit:** If everyone agrees, contact every participant again to let them know. Otherwise, contact every participant to *abort* the consensus

Two-phase commit (2PC)



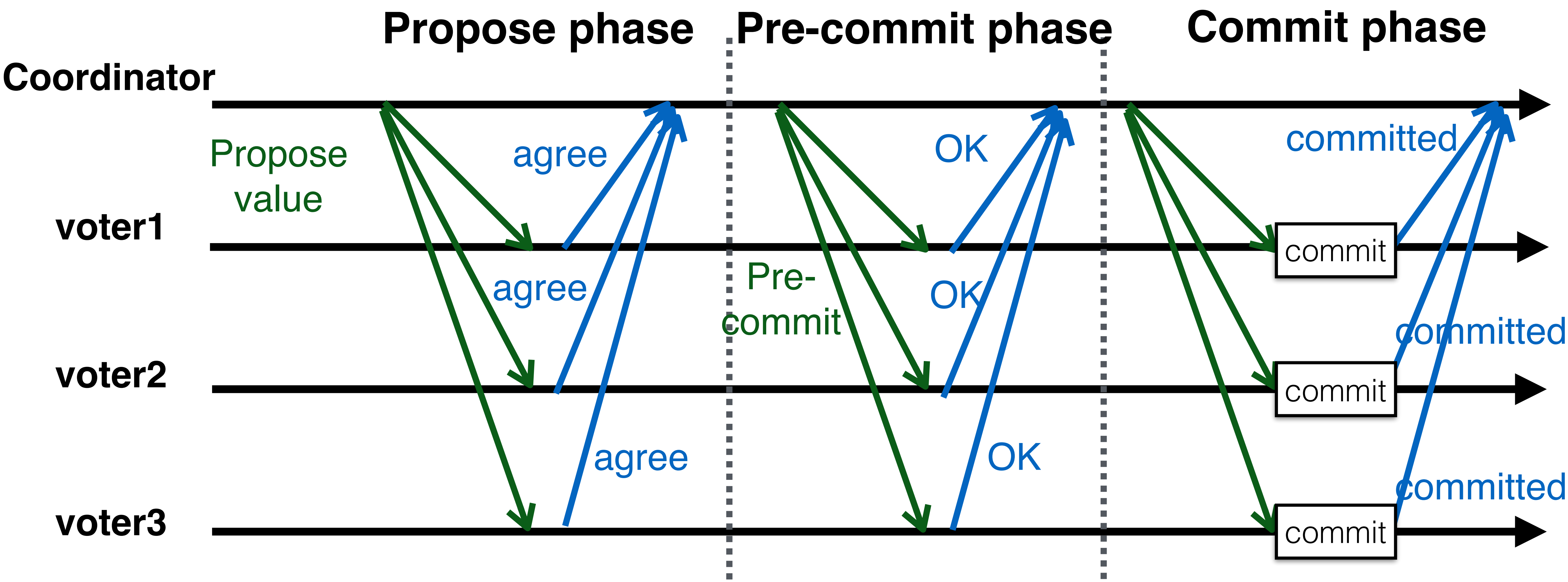
- If the proposal was not accepted by any one of the voters, the proposal will not be committed (aborted)
- Voters{1,2,3} are still in consensus

Crashes and failures in 2PC



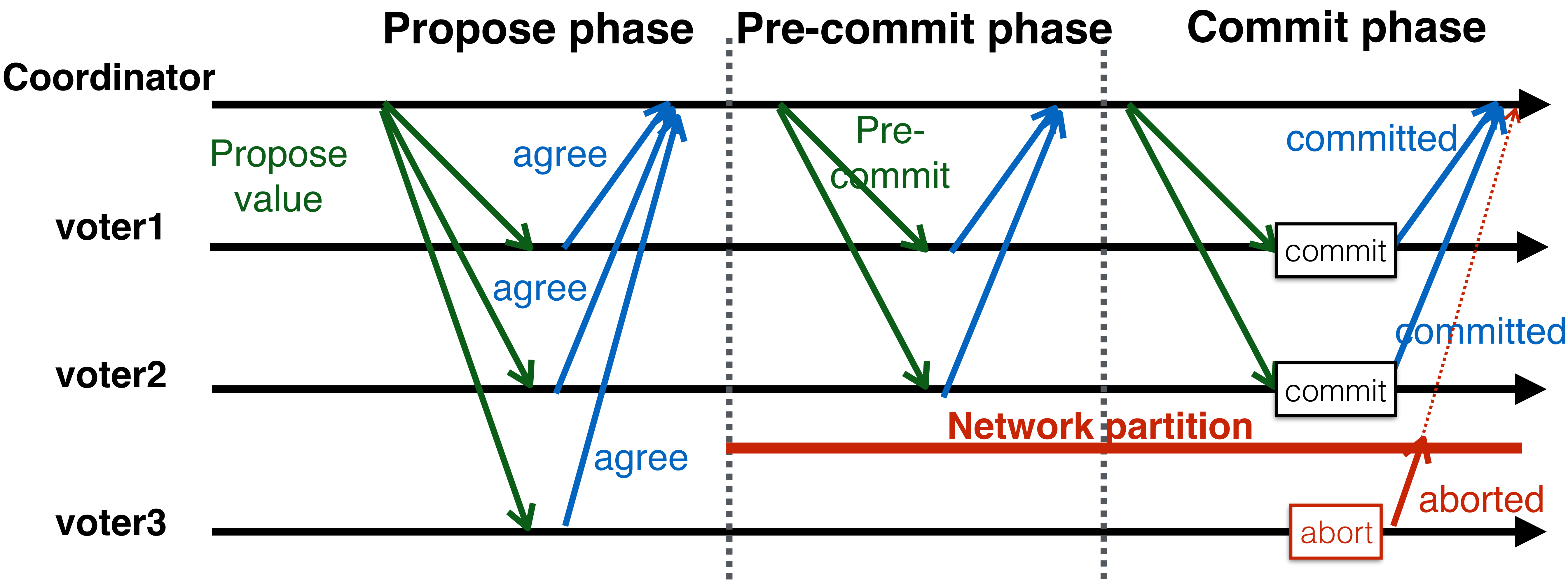
- 2PC is not able to handle **fail-stop** failure
  - Coordinator and voter3 both crash during Commit phase
  - Voter1 and voter2 fall in a dilemma where they cannot decide whether:
    - **Voter3 has agreed (in Propose phase) and committed**
    - **-OR- disagreed (in Propose phase)**

Three-phase commit (3PC)



- 3PC further breaks the "***Commit phase***" of 2PC into ***two*** sub-phases
  1. **Prepare-to-commit:** Every participant gets to know the voting result without commitment (<u>so that they can get prepared to commit</u>…)
  2. **Commit:** If everyone agrees & is willing to commit, contact every participant again to let them know. Otherwise, contact every participant to *abort* the consensus

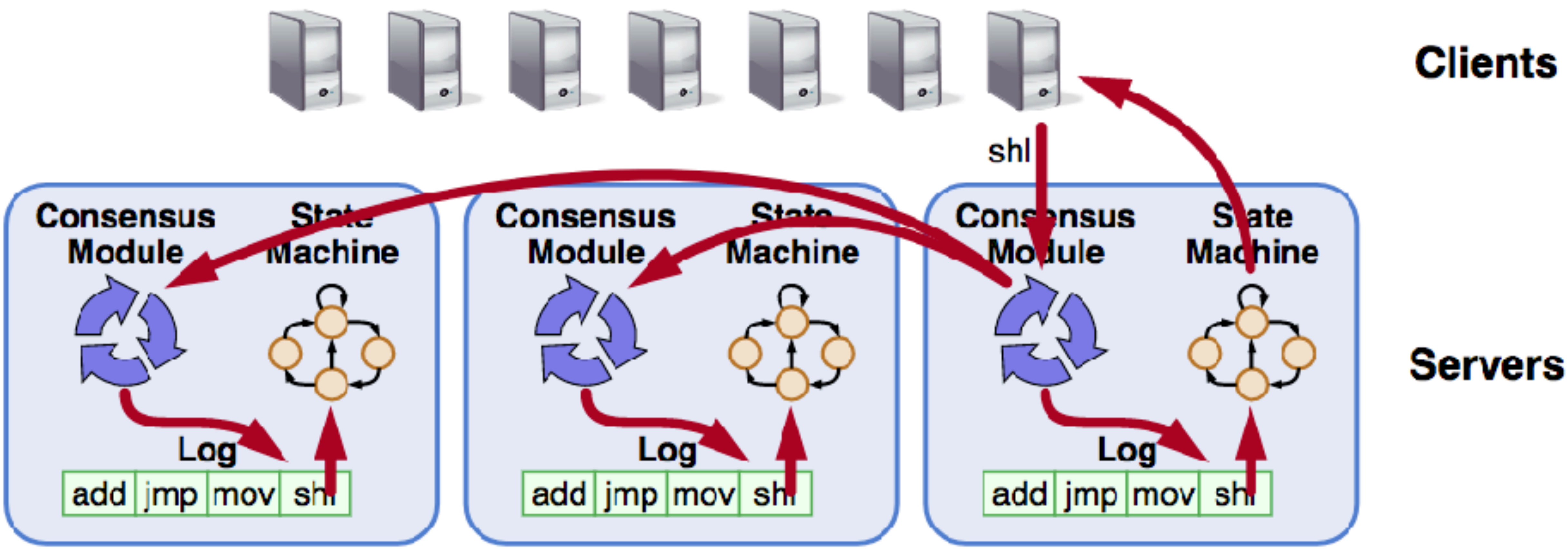
Crashes and failures in 3PC



- 3PC is not able to handle **network partition**
  - At Prepare-to-commit phase, network partition occurs, voter1 and voter2 will do something opposite than voter3 (<u>who is on the other side of the partitioned network</u>)

One step further: Paxos

Goal of Paxos in practice



- Replicated log —> replicated state machine
  - All servers execute the same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
- Failure model: fail-stop (not Byzantine), delayed/lost messages

Picture credit: Ousterhout and Ongaro, Implementing Replicated Logs with Paxos

Requirements for basic Paxos

- **Safety**
  - Only one value that has been proposed may be chosen
  - If a value is chosen by a process, then the same value must be chosen by any other process that has chosen a value
- **Liveness (as long as majority of servers are up and communicating with reasonable timeliness**
  - Some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value

> "… it is among the simplest and most obvious of distributed algorithms…" — Leslie Lamport

The Paxos algorithm

- **Contribution:** Separately consider safety and liveness issues
  - Safety can be guaranteed (**consensus is not violated**)
  - Liveness is ensured during period of synchrony: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached (**eventually**)

Paxos components

- **Proposers**
  - Active: put forth particular values to be chosen
  - Handle client requests
- **Acceptors**
  - Passive: response to messages from proposers
  - Responses represent votes that form consensus
  - Store chosen value, state of the decision process
  - Want to know which value was chosen
- **Assumption**
  - Each Paxos server contains both components

Proposal numbers

- Each proposal has a unique number
  - Higher numbers take **priority** over lower numbers
  - It must be possible for a proposer to choose a new proposal number higher than anything it has seen/used before
- One simple approach

Proposal Number

Round number	Server ID

  - Each server stores maxRound: the largest Round Number it has seen so far
  - To generate a new proposal number:
    - Increment maxRound
    - Concatenate with Server ID
  - Proposers must **persist** maxRound on disk: must not reuse proposal numbers after crash/restart

Basic Paxos

- **Two-phase approach**
  - **Phase 1: Broadcast Prepare RPCs**
    - Find out about any chosen values
    - Reject older proposals that have not yet completed
  - **Phase 2: Broadcast Accept RPCs**
    - Ask acceptors to accept a specific value

Basic Paxos

Proposers | **Acceptors**

Proposers

1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers

Acceptors

3) Respond to Prepare(n):
   - If n > minProposal then minProposal = n
   - Return(acceptedProposal, acceptedValue)

Proposers

4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers

Acceptors

6) Respond to Accept(n, value):
   - If n ≥ minProposal then
     acceptedProposal = minProposal = n
     acceptedValue = value
   - Return(minProposal)

Proposers

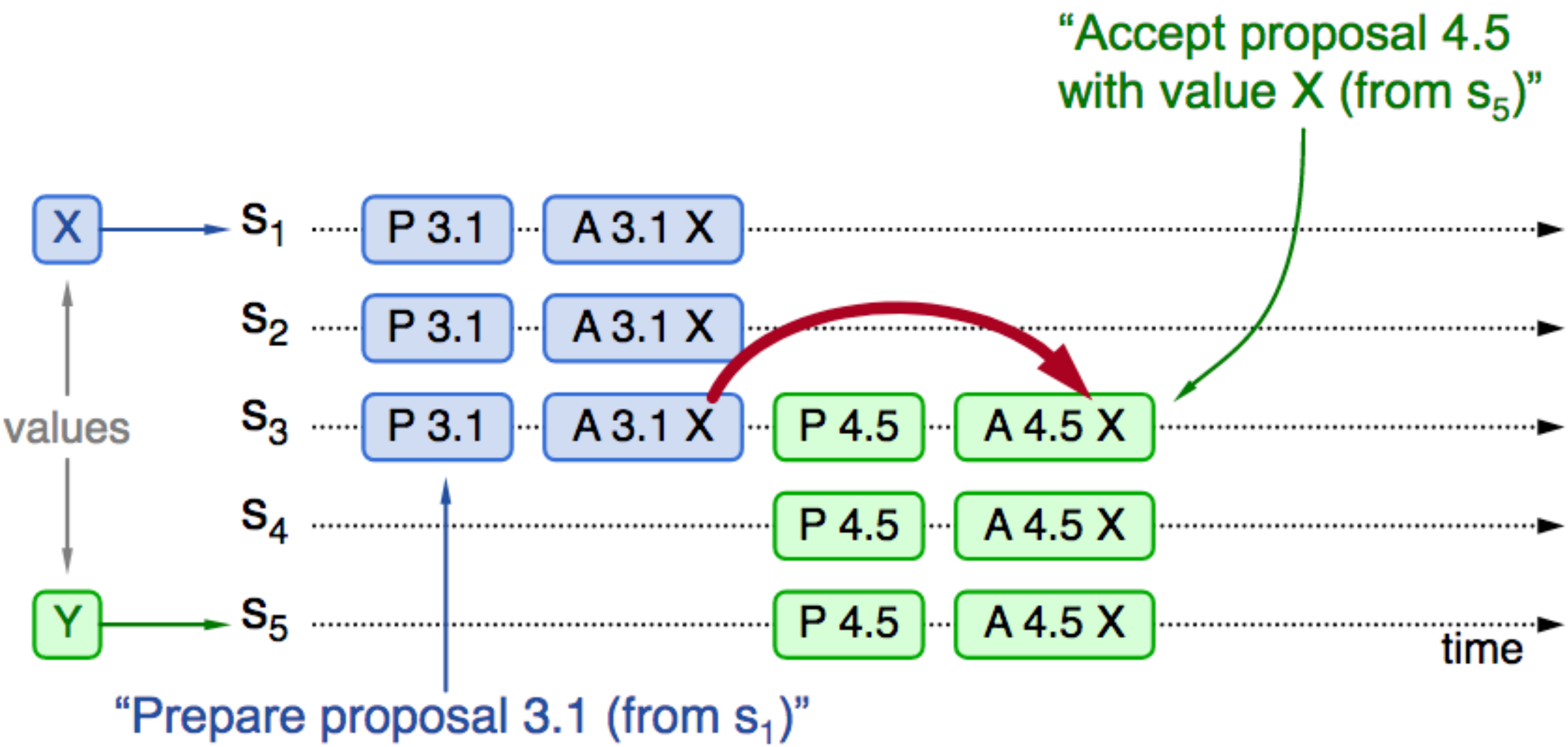
6) When responses received from majority:
   - Any rejections (result > n)? goto (1)
   - Otherwise, **value is chosen**

Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)

Picture credit: Ousterhout and Ongaro, Implementing Replicated Logs with Paxos

Basic Paxos examples

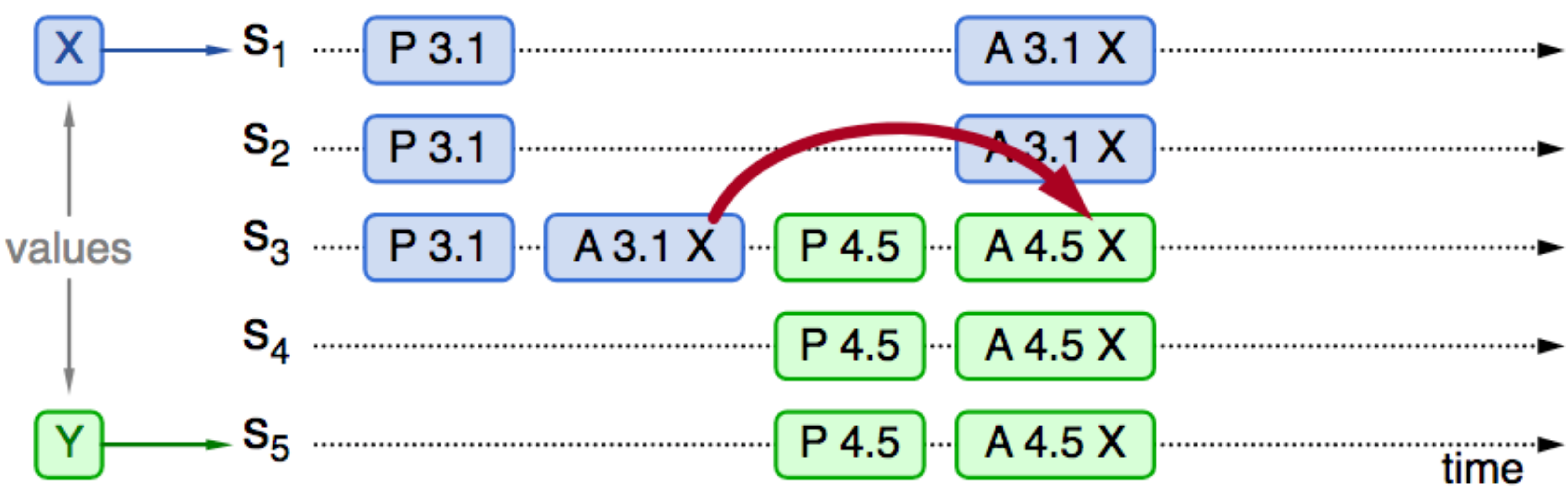
- What if previous value is already chosen
  - New proposer will find it and use it



Picture credit: Ousterhout and Ongaro, Implementing Replicated Logs with Paxos

Basic Paxos examples

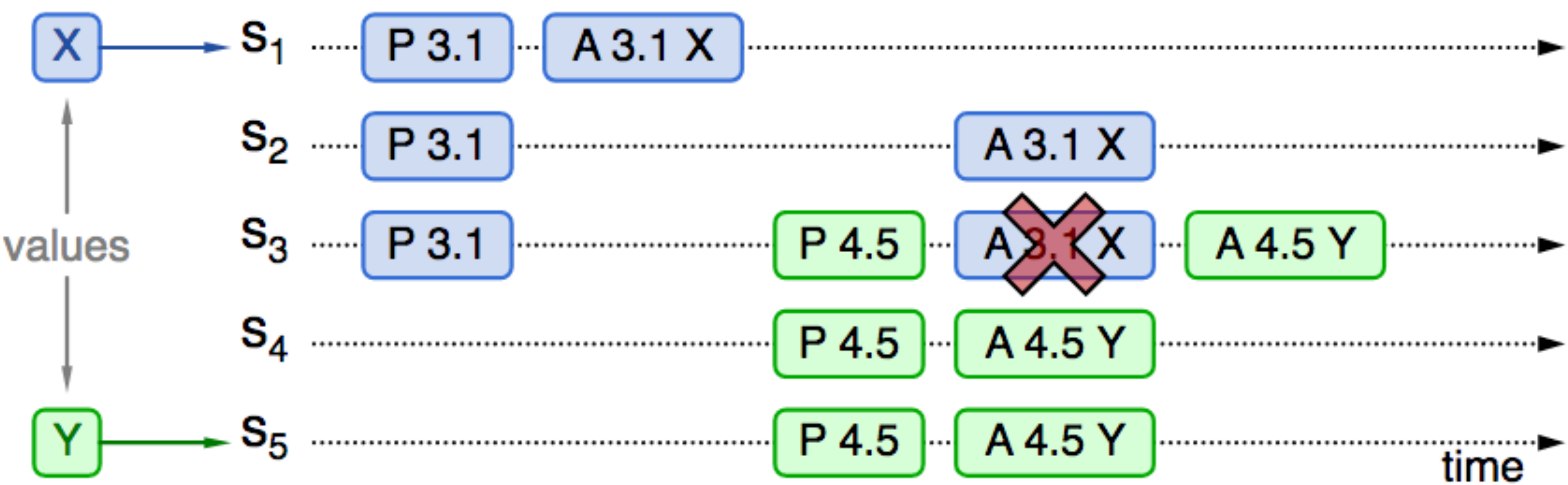
- What if previous value has not been chosen but new proposer sees it
  - New proposer will use existing value
  - Both proposers can succeed



Picture credit: Ousterhout and Ongaro, Implementing Replicated Logs with Paxos

Basic Paxos examples

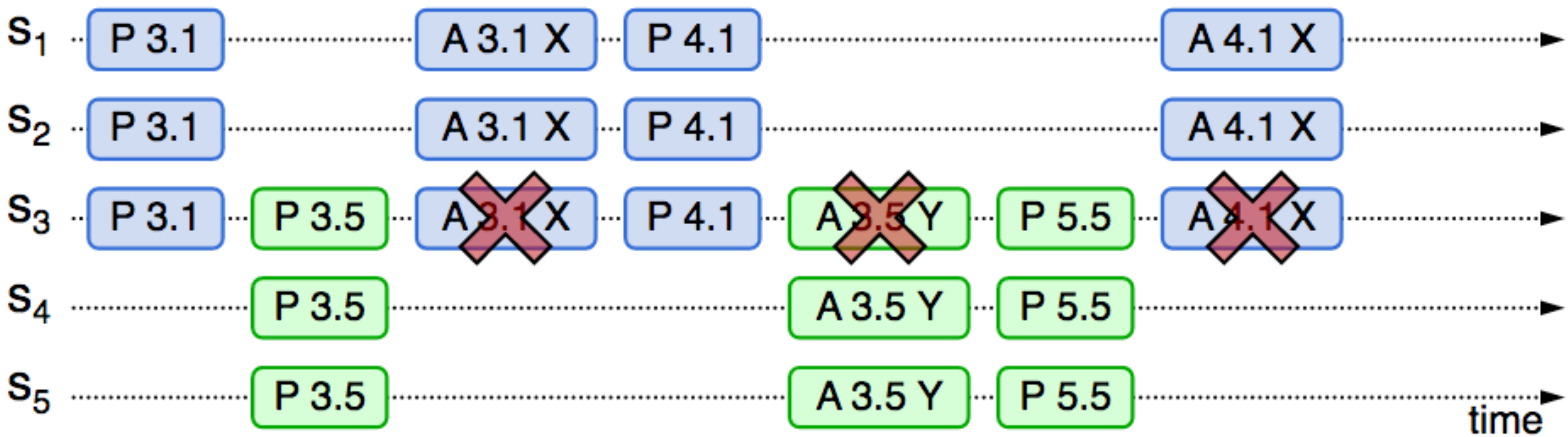
- What if previous value has not been chosen but new proposer doesn't see it
  - New proposer chooses its own value
  - Older proposal rejected



Picture credit: Ousterhout and Ongaro, Implementing Replicated Logs with Paxos

Liveness

- Competing proposers can **livelock**



- **One solution: randomized delay before restarting**
  - Give other proposers a chance to finish choosing

Picture credit: Ousterhout and Ongaro, Implementing Replicated Logs with Paxos

Announcements

- Next class: paper presentations and discussions
  - **Raft + Zookeeper**

Apache ZooKeeper™

- Make sure to fill out the paper evaluation form (Google form closes 10 min before class)
- Scribe report on Piazza due by end of next day (Thursday)