How to apply and flesh out Paxos

Diego Ongaro

June 2012

Intro

- Can I explain how to use Paxos in a practical and complete implementation?
- These ideas form the basis of LogCabin
  - A new configuration service for distributed systems (like Chubby, ZooKeeper)

Intro

- Can I explain how to use Paxos in a practical and complete implementation?
- These ideas form the basis of LogCabin
  - A new configuration service for distributed systems (like Chubby, ZooKeeper)
- Quick survey
  1. Have heard of Paxos?
  2. Know when to apply Paxos?
  3. Understand Paxos?
  4. Fear Paxos?

Brief History

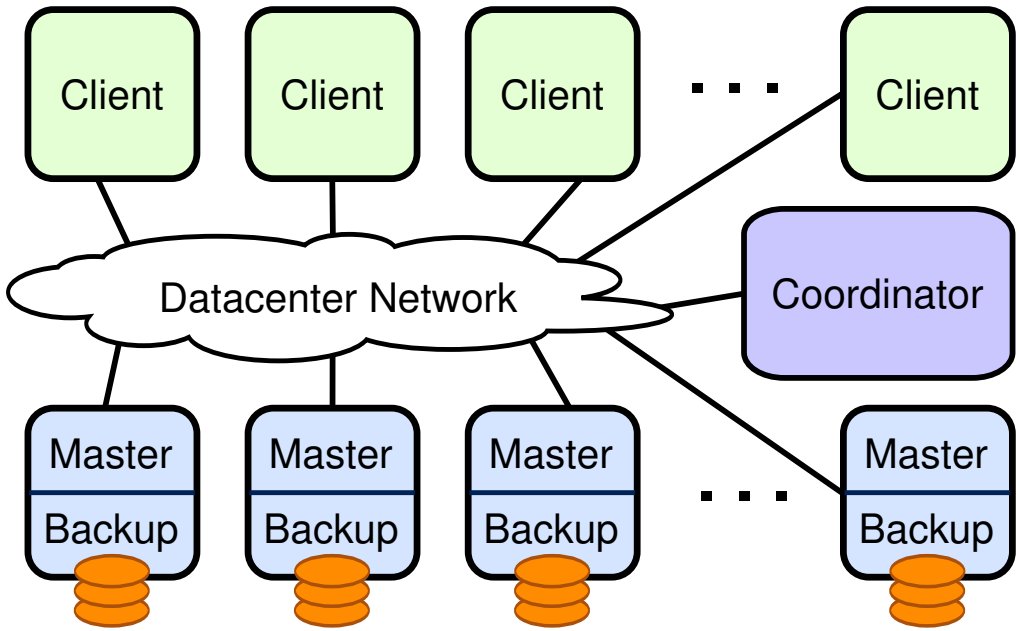
- Viewstamped Replication – 1988 – Oki and Liskov
- Paxos – 1989 through 1998 – Lamport
- This presentation explains a variant of Multi-Paxos

Goals and assumptions

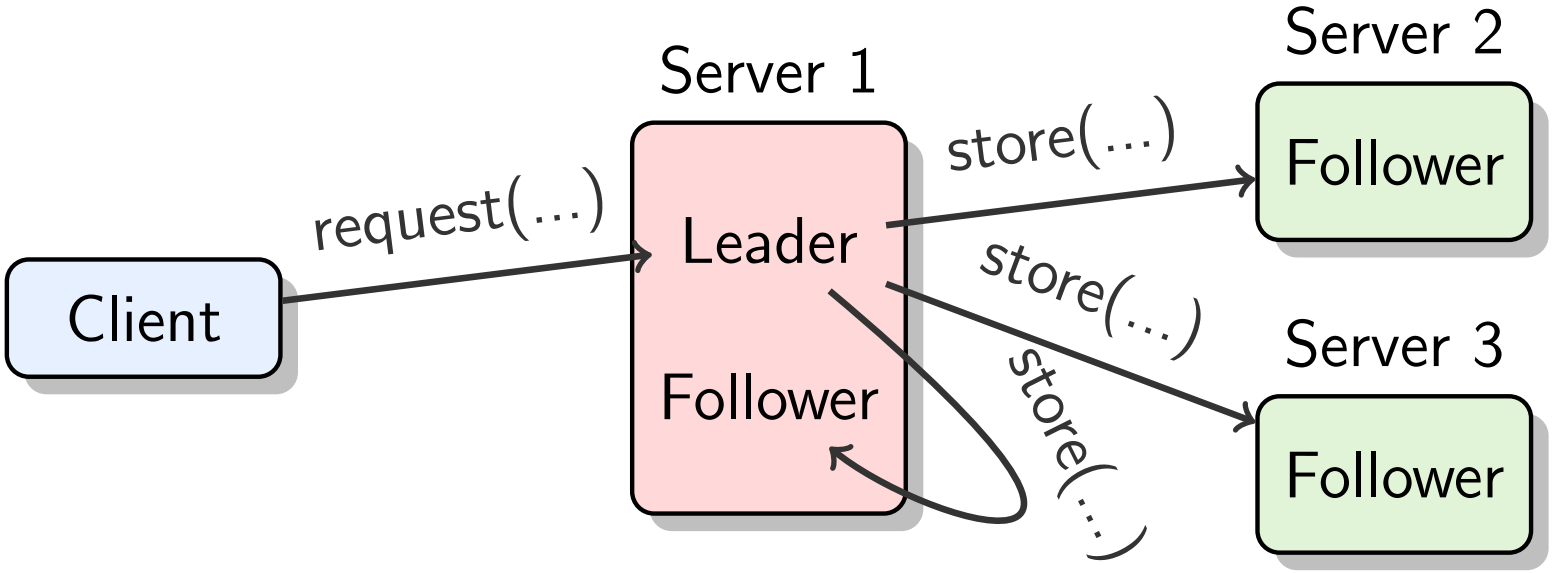
- Goal: framework to build a small, fault-tolerant state machine
- Servers can crash at any time, can later restart
  - Assume non-byzantine failures
- No single point of failure
  - Service should be up if any majority of the cluster is up
- Small cluster sizes, such as 5 servers

When is it appropriate to use Paxos?

- Want fault-tolerant service and can't tolerate split-brain problem
- In RAMCloud, the cluster coordinator must be fault-tolerant, but there must be at most one at a time.

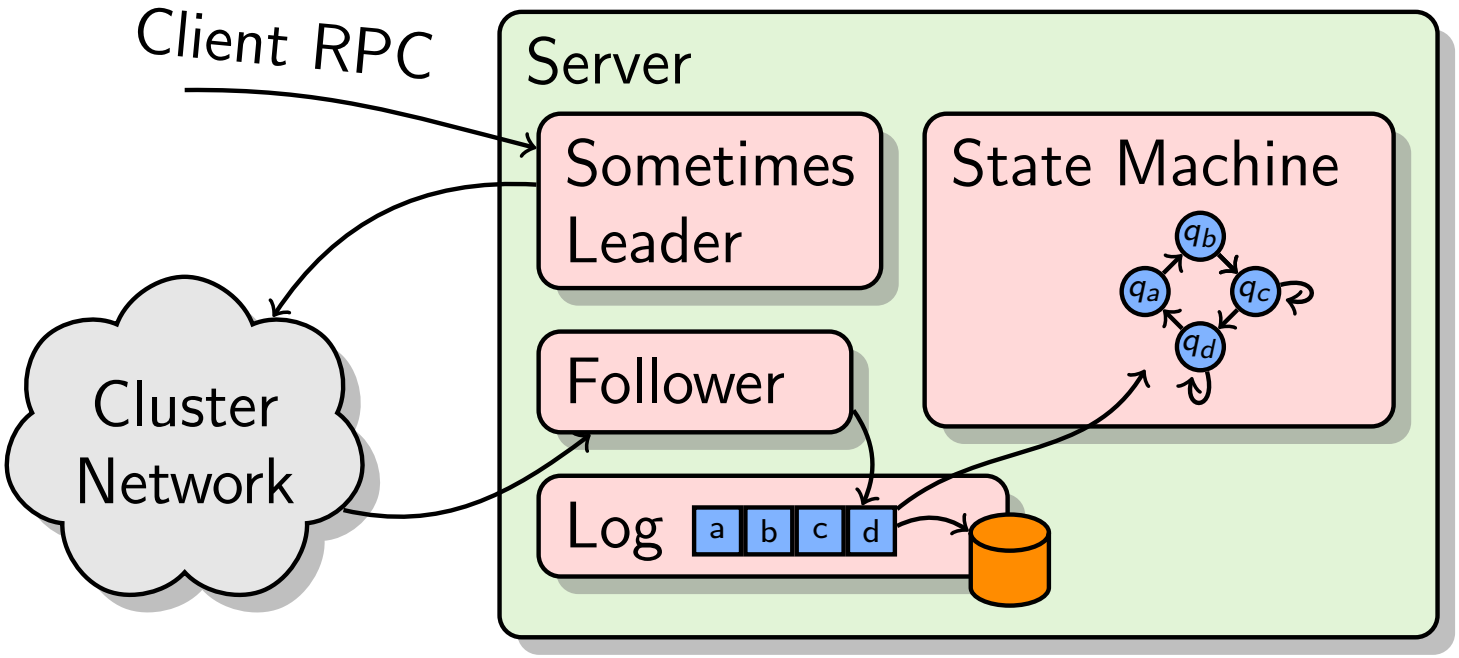


Desired operation



- Leader: a server willing to act on client requests
- Can't guarantee a single leader at a time
  - Will guarantee safety with multiple leaders
  - Will favor a single leader using timeouts

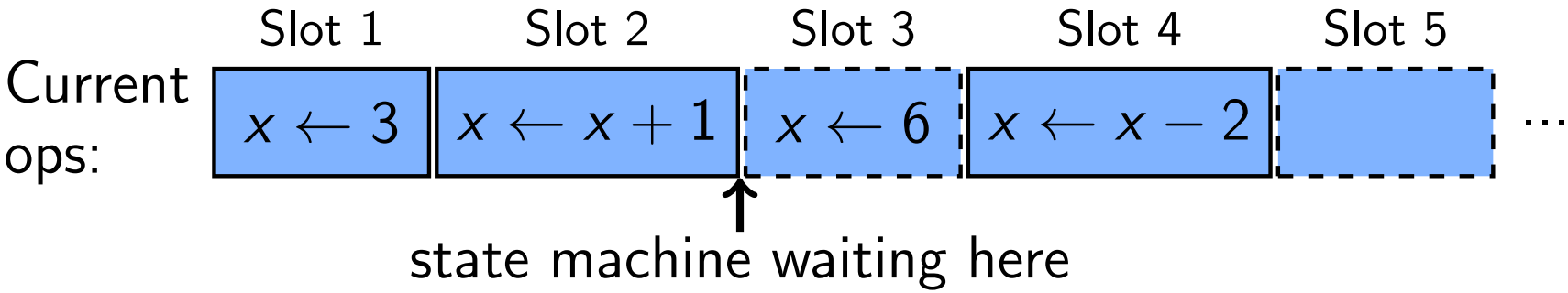
Replicating a log of operations



- This framework provides an ordered log of operations to a state machine
- The state machine can implement a key-value store, a lock server, etc
- If all servers play the same log, their states will be the same

Log contents

- Each server stores a full copy of the log, made up of *slots*:
  - *operation* – a client's request to the state machine
  - *finalized* flag – set after majority of the replicas have stored same operation, guaranteed not to change
  - *epoch* – explained later
- Each state machine advances once the next slot's operation is finalized
- Main idea: take a client's request, commit it to the next available slot, wait for the local state machine to advance there, respond to the client

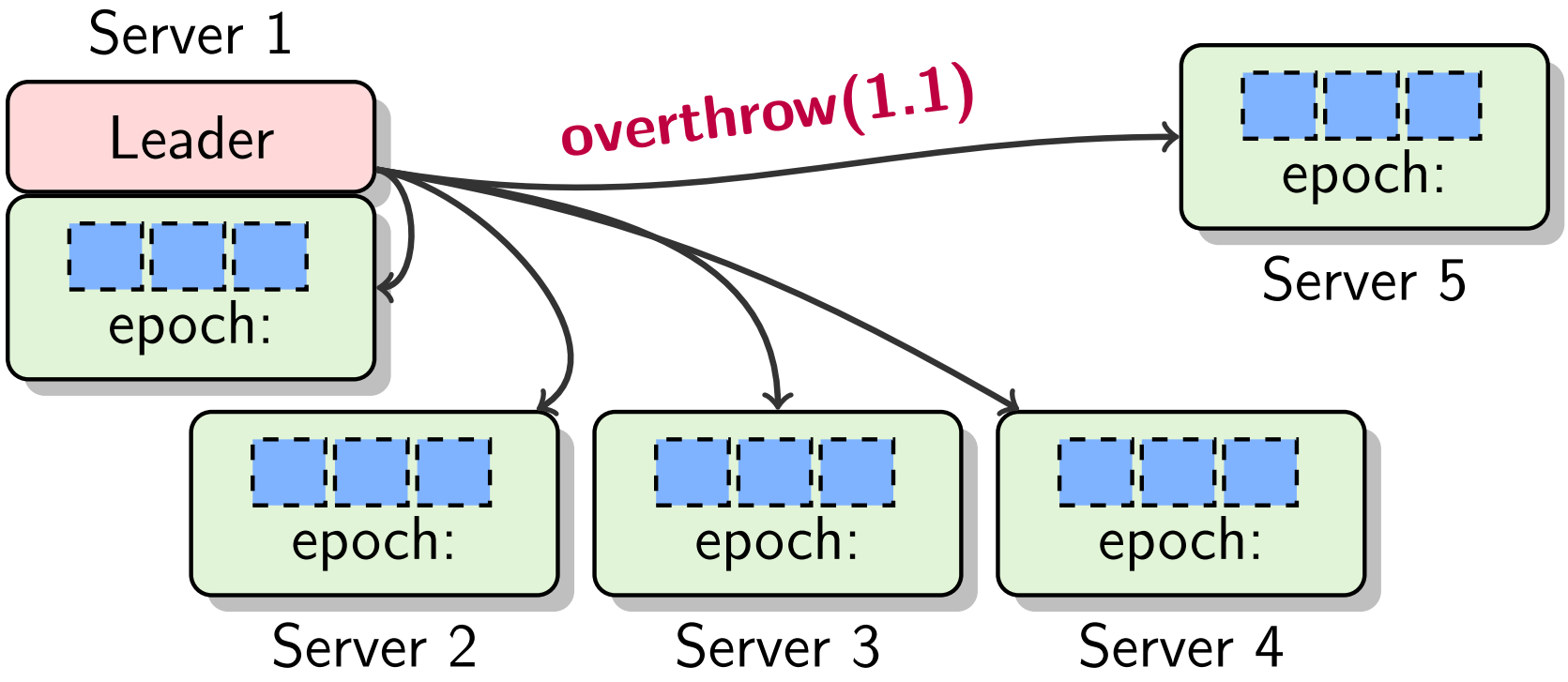


Three-phase algorithm

1. overthrow(new epoch) $\rightarrow$
   last used slot | current epoch
   - Used by new leader to kill off old leader
2. store(epoch, slot, operation) $\rightarrow$ ok | current epoch
   - Used to replicate operations
3. finalize(epoch, slot) $\rightarrow$ ok | missing
   - Used to flag slots as immutable

Always need a majority of responses

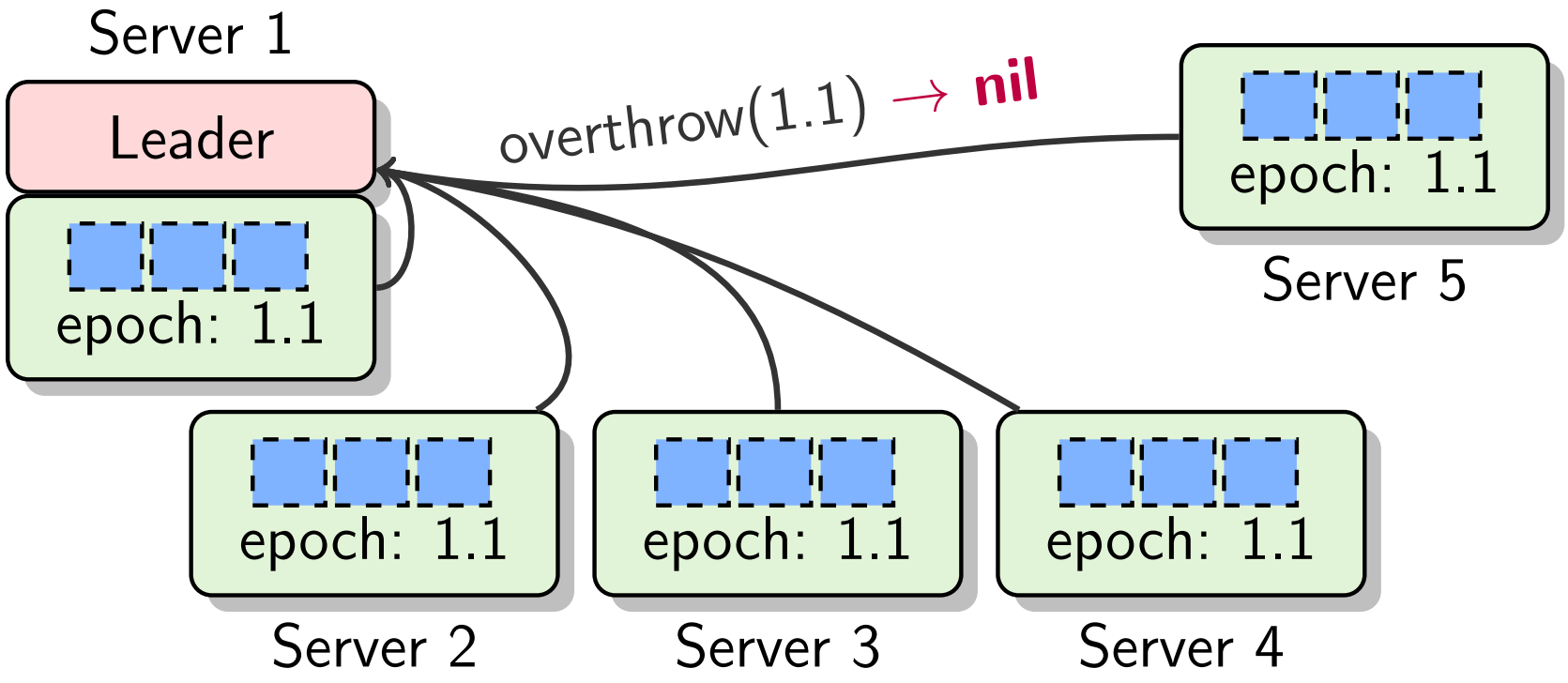
Beginning of time



overthrow(new epoch) $\rightarrow$ last used slot | current epoch

- Won't make sense yet – just need to bootstrap
- Epoch is made up of monotonically increasing number and server ID

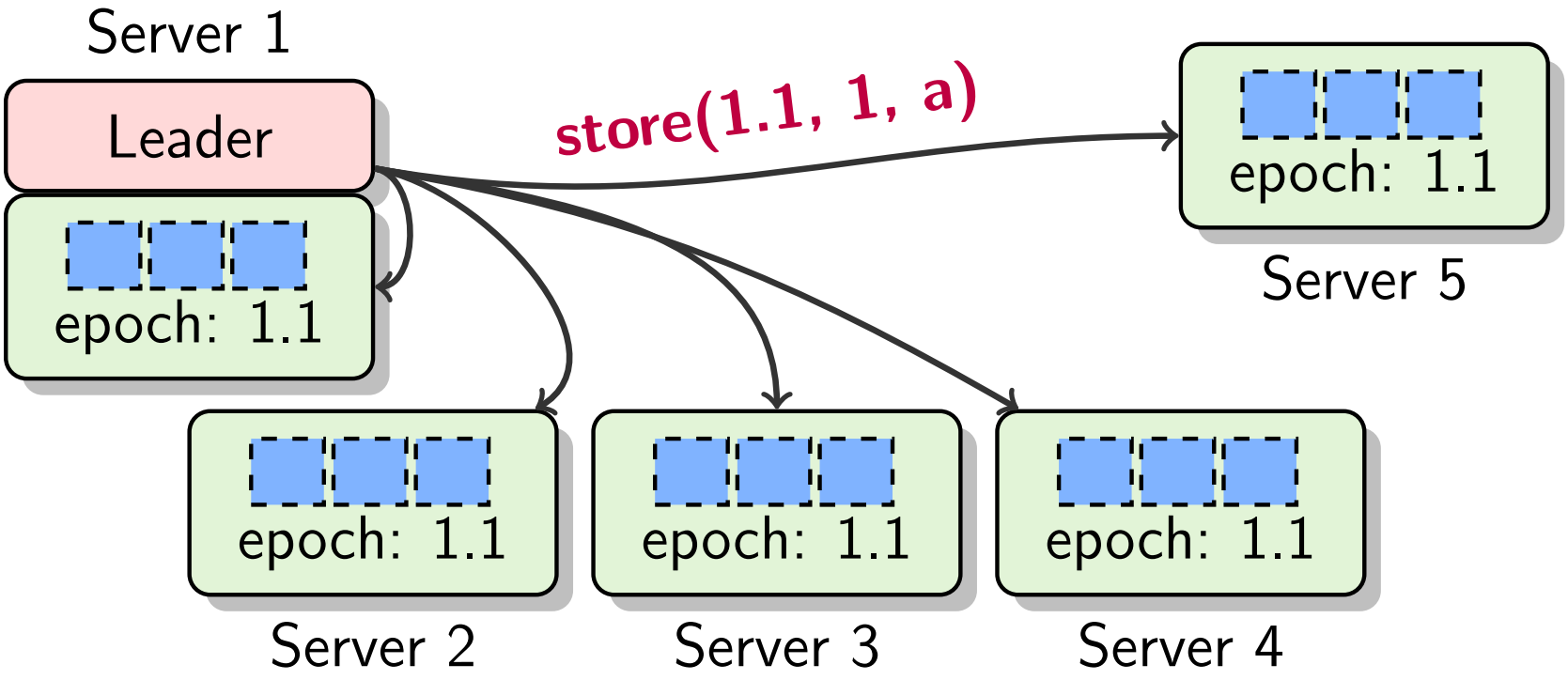
Beginning of time



overthrow(new epoch) → last used slot | current epoch

- Won't make sense yet – just need to bootstrap
- Epoch is made up of monotonically increasing number and server ID

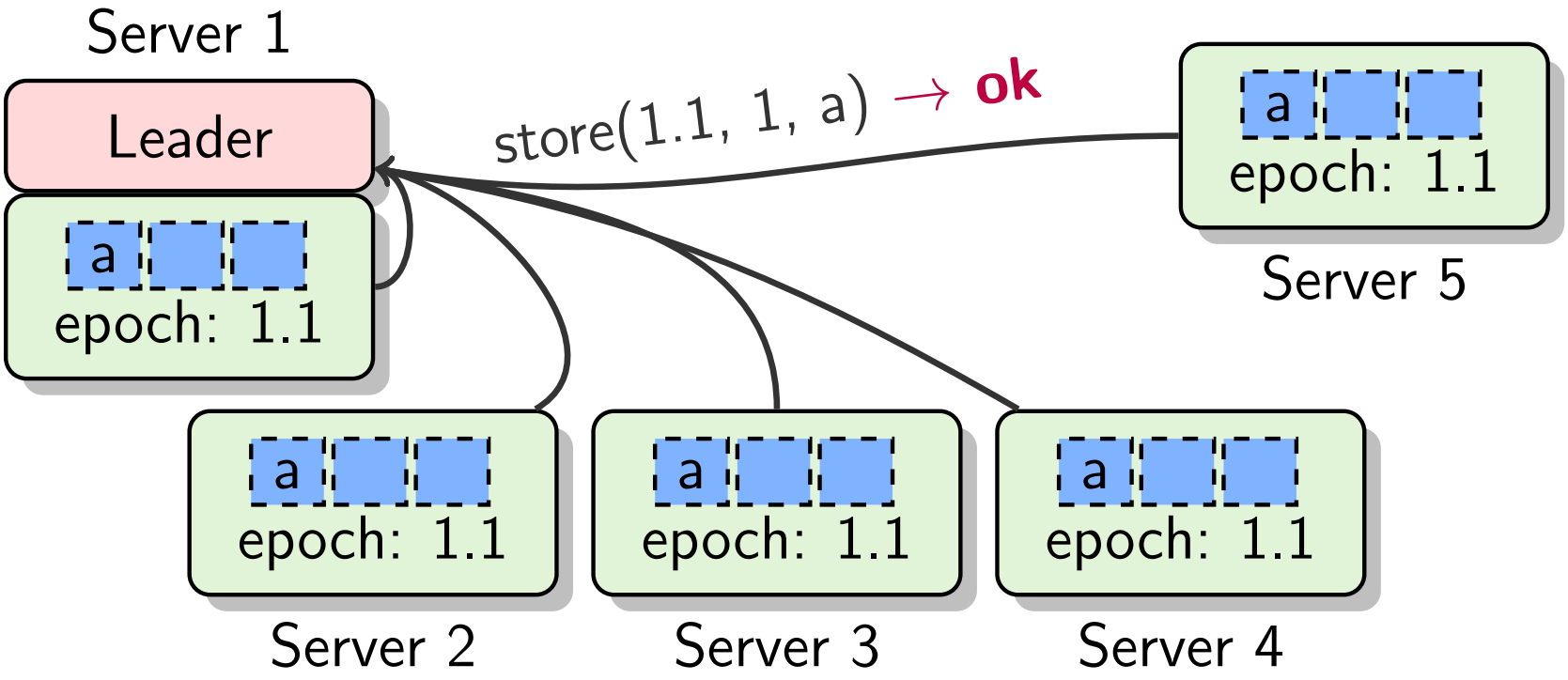
Store



store(epoch, slot, operation) $\rightarrow$ ok | current epoch

- If epoch = current, return ok, else return current
- Used to replicate operations
- Later stores may overwrite earlier stores

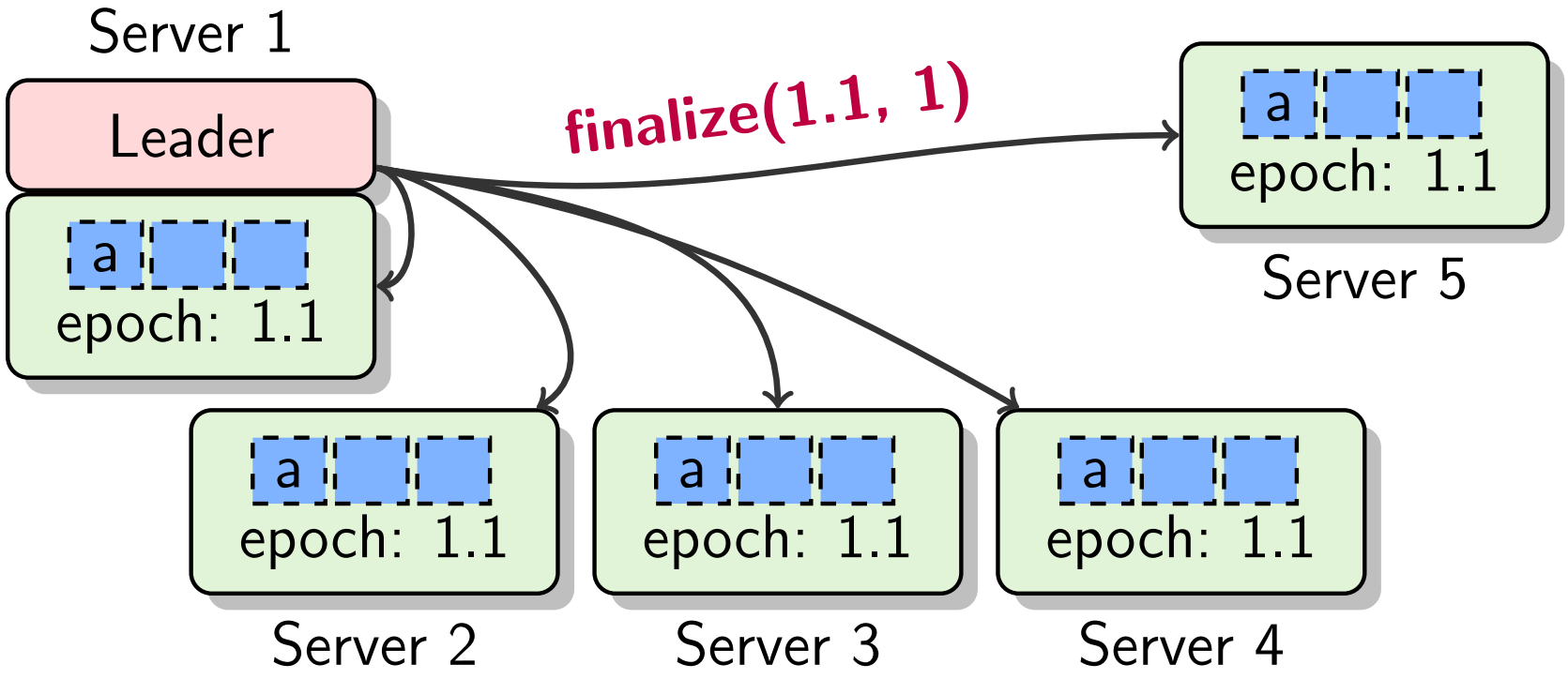
Store



store(epoch, slot, operation) → ok | current epoch

- If epoch = current, return ok, else return current
- Used to replicate operations
- Later stores may overwrite earlier stores

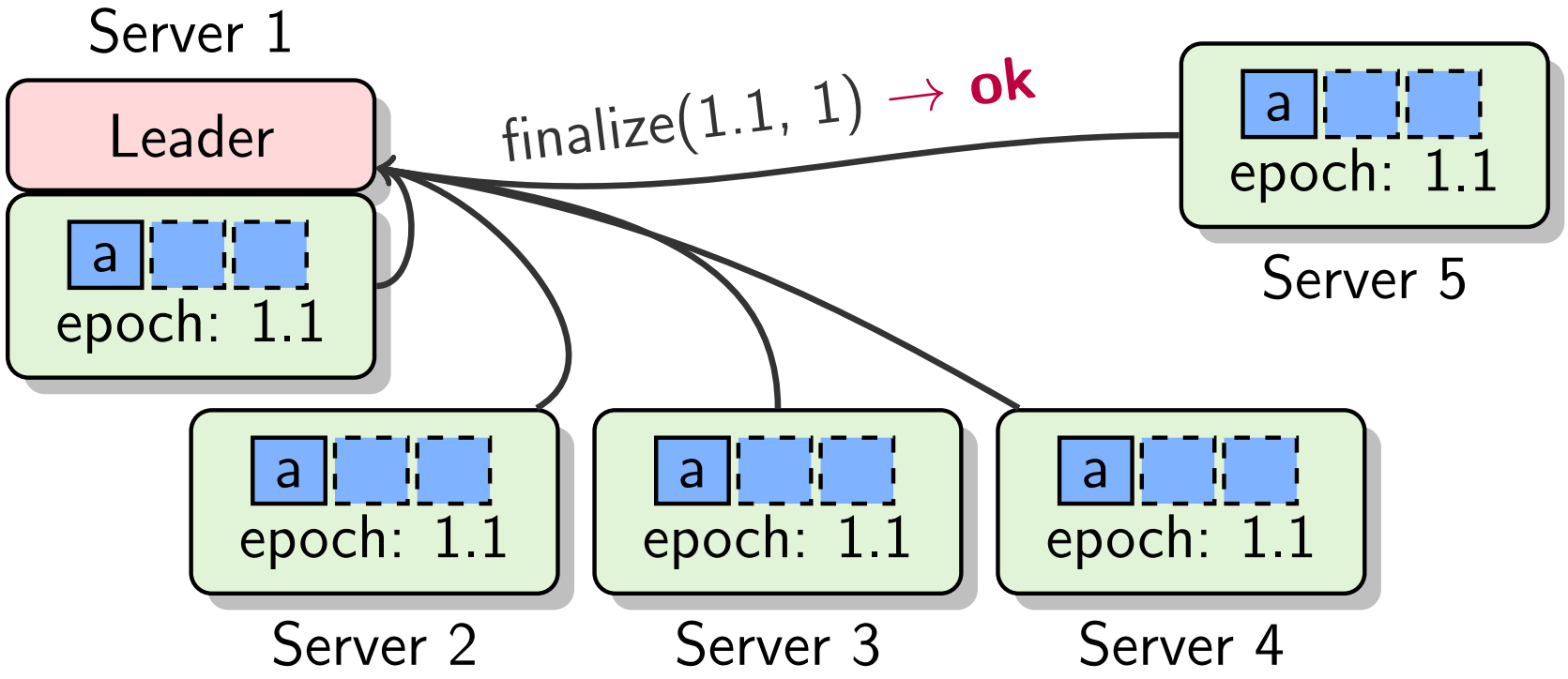
Finalize



finalize(epoch, slot) $\rightarrow$ ok | missing

- If slot has (epoch, op), mark as finalized
- Allows state machines to advance

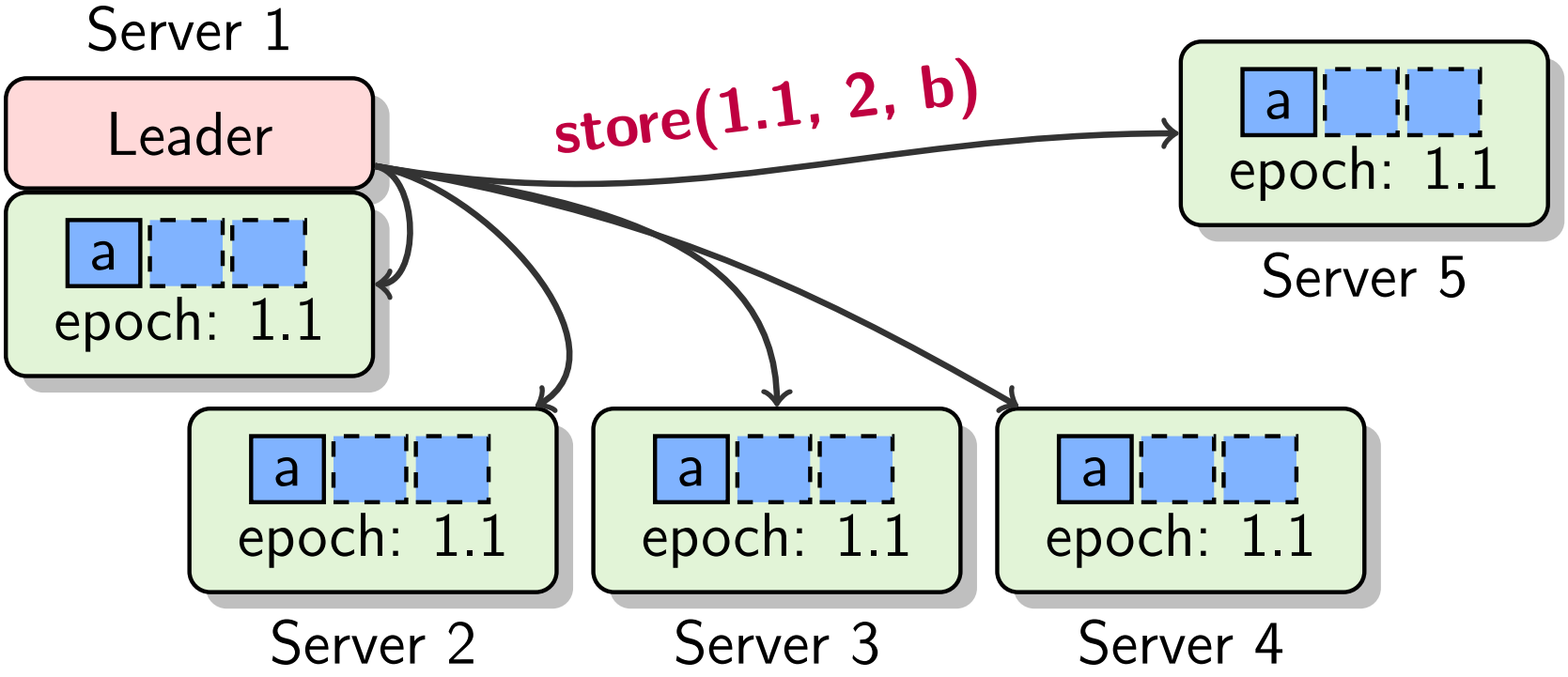
Finalize



finalize(epoch, slot) → ok | missing

- If slot has (epoch, op), mark as finalized
- Allows state machines to advance

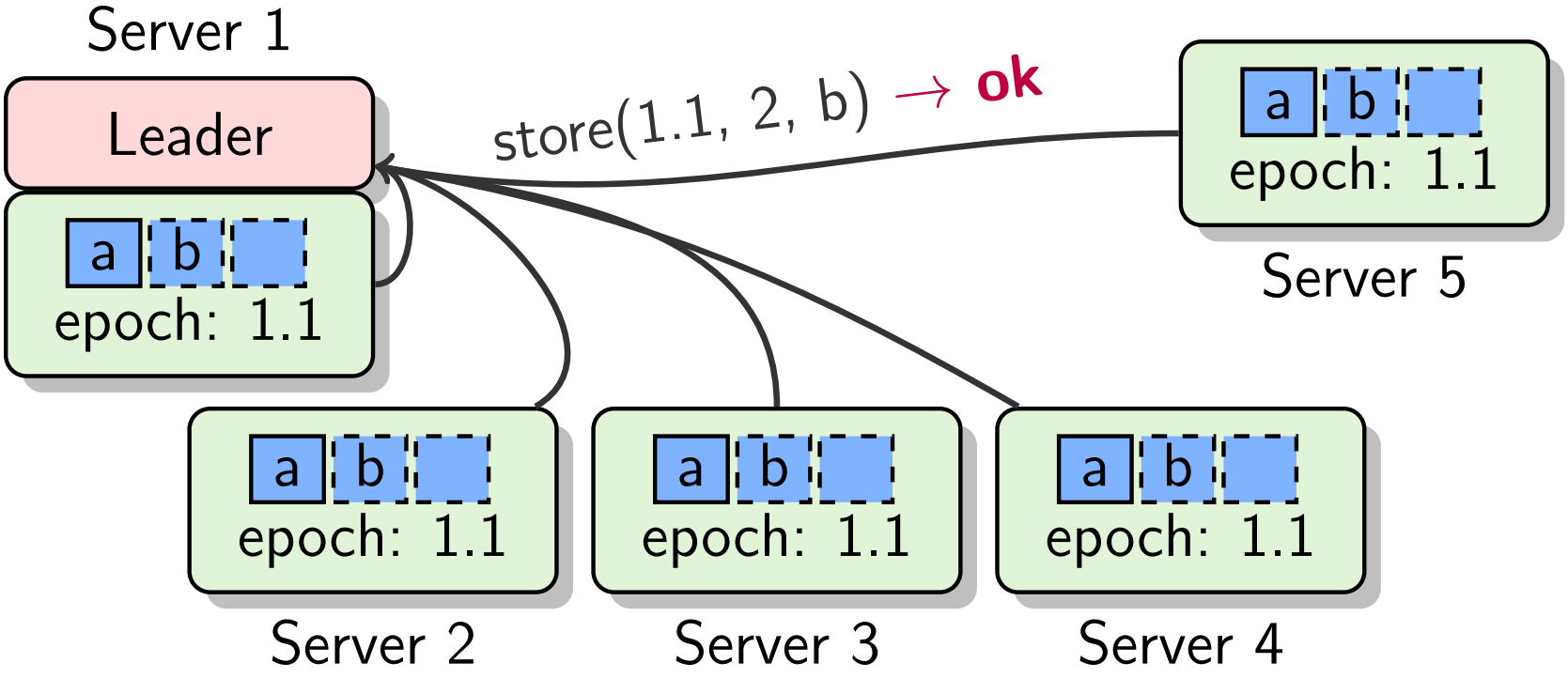
Store again



store(epoch, slot, operation) $\rightarrow$ ok | current epoch

- If epoch = current, return ok, else return current

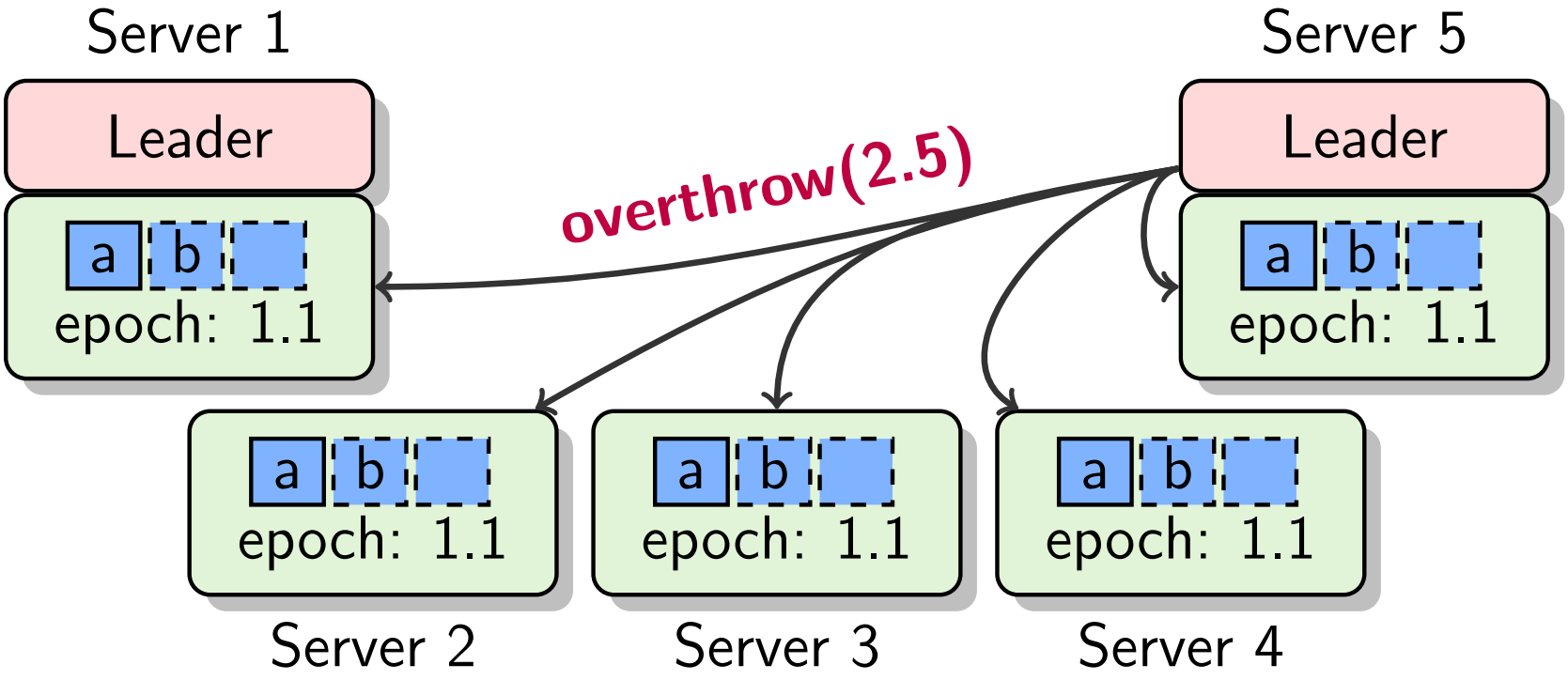
Store again



store(epoch, slot, operation) → ok | current epoch

- If epoch = current, return ok, else return current

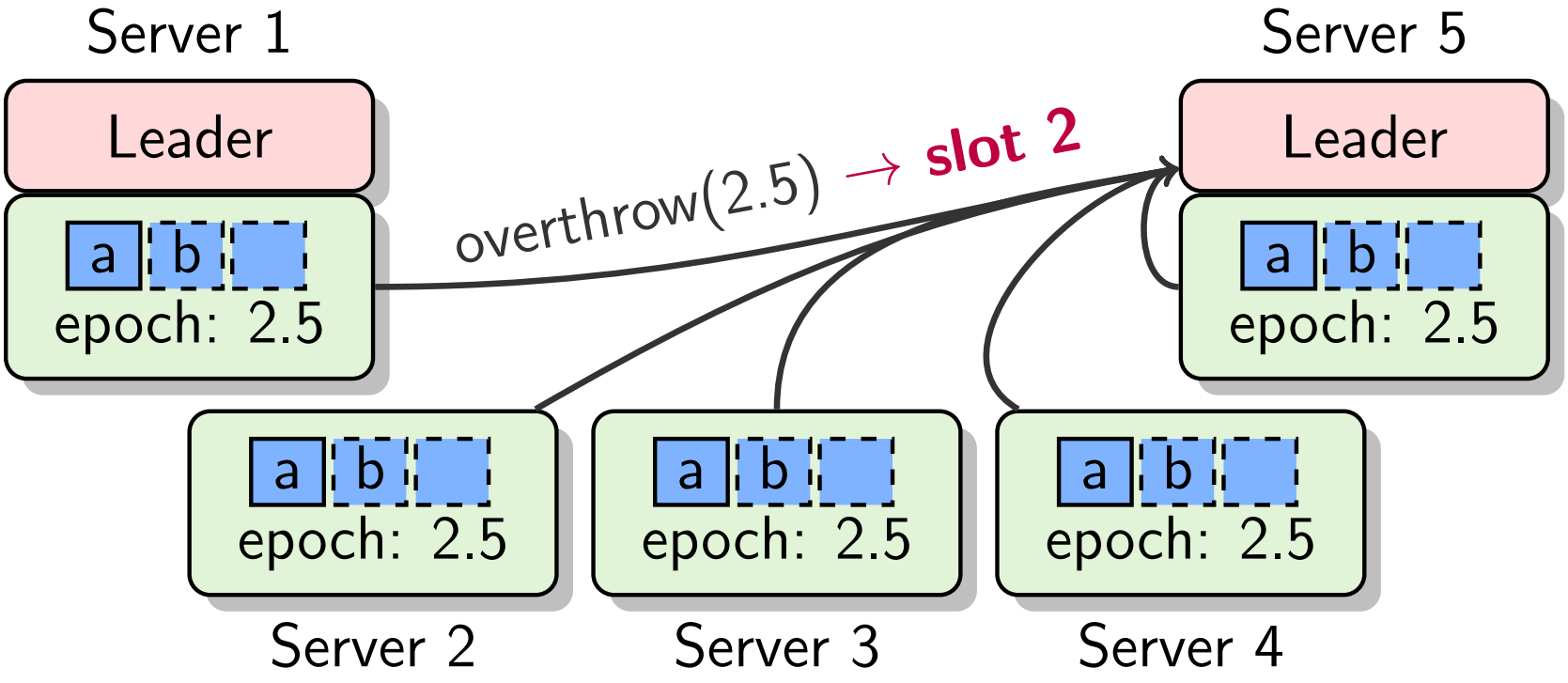
New leader overthrows old leader



overthrow(new epoch) $\rightarrow$ last used slot | current epoch

- If epoch $\geq$ current, return last used slot, else return current
- Used by new leader to kill off old leader

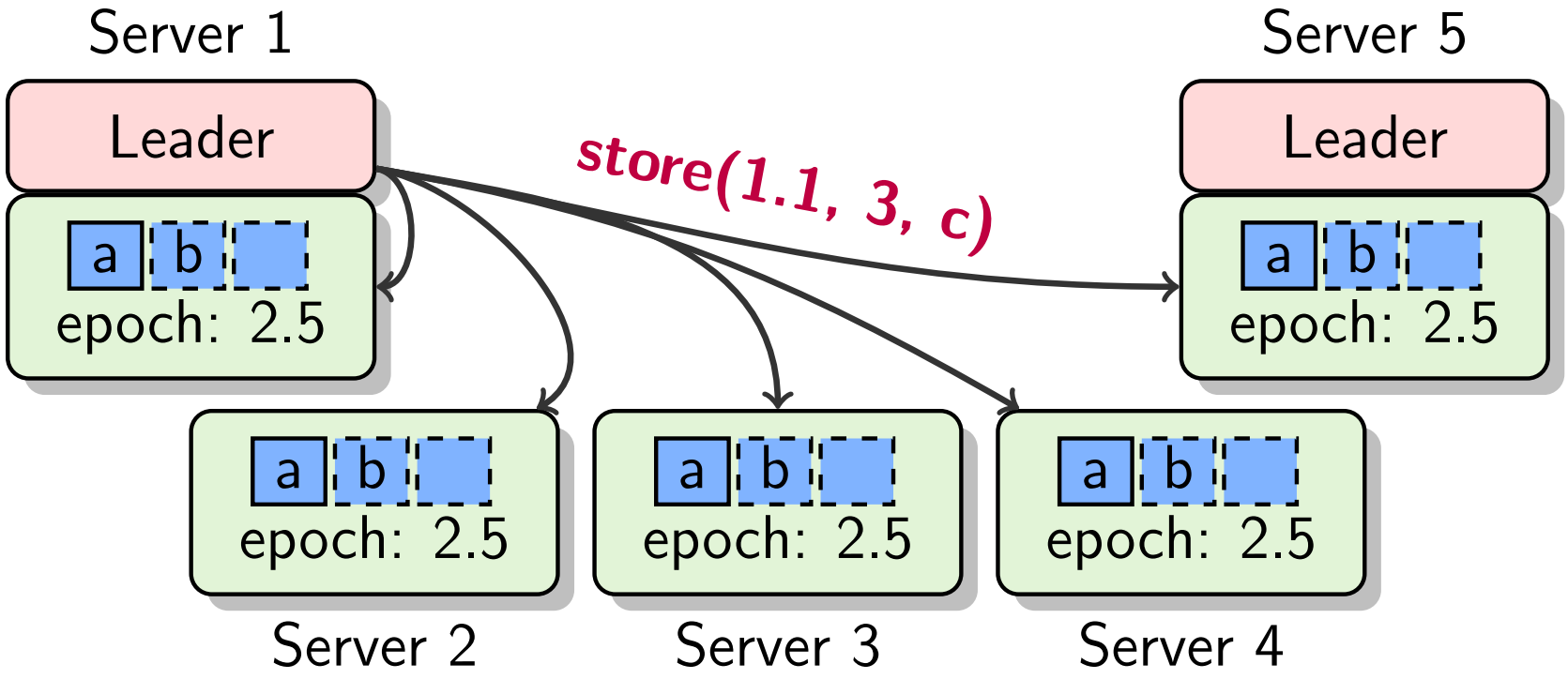
New leader overthrows old leader



overthrow(new epoch) → last used slot | current epoch

- If epoch ≥ current, return last used slot, else return current
- Used by new leader to kill off old leader

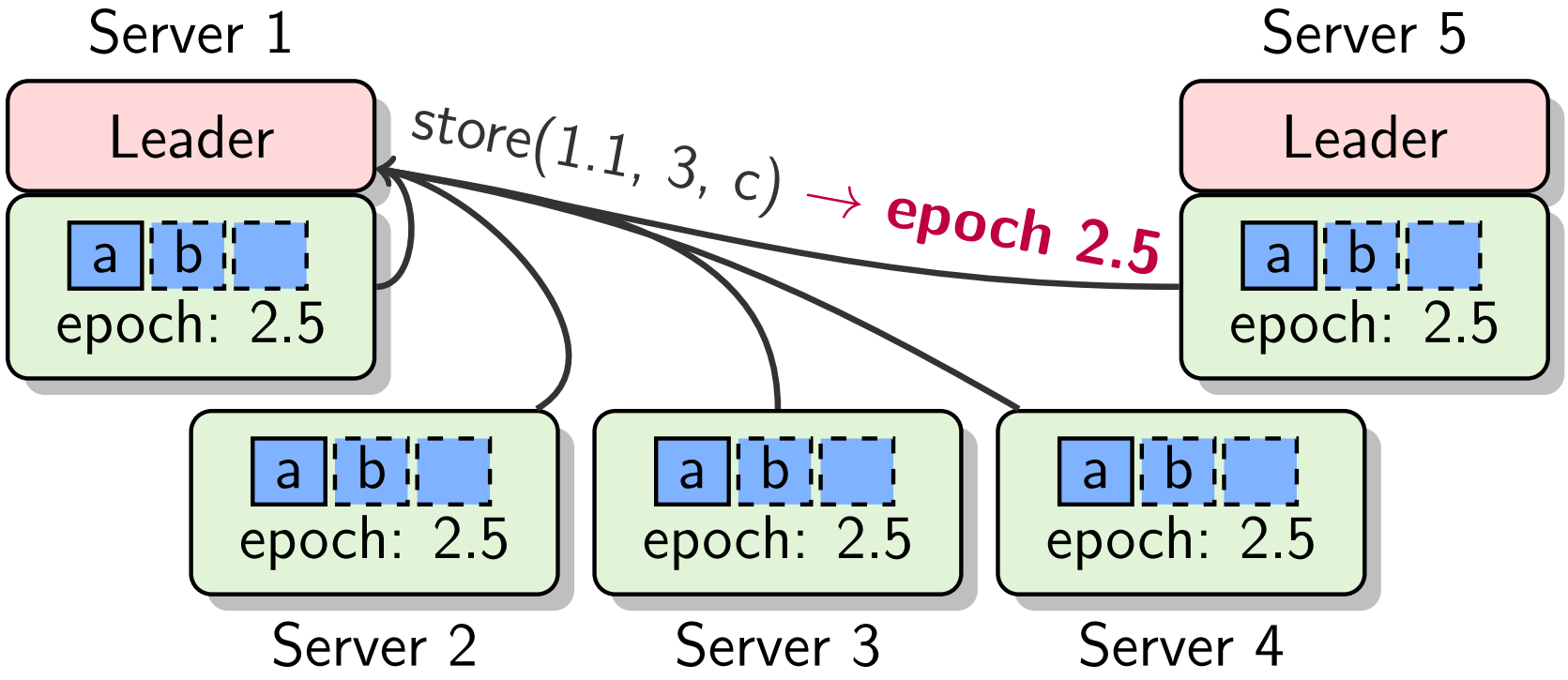
Old leader can no longer store



store(epoch, slot, operation) $\rightarrow$ ok | current epoch

- If epoch $=$ current, return ok, else return current

Old leader can no longer store

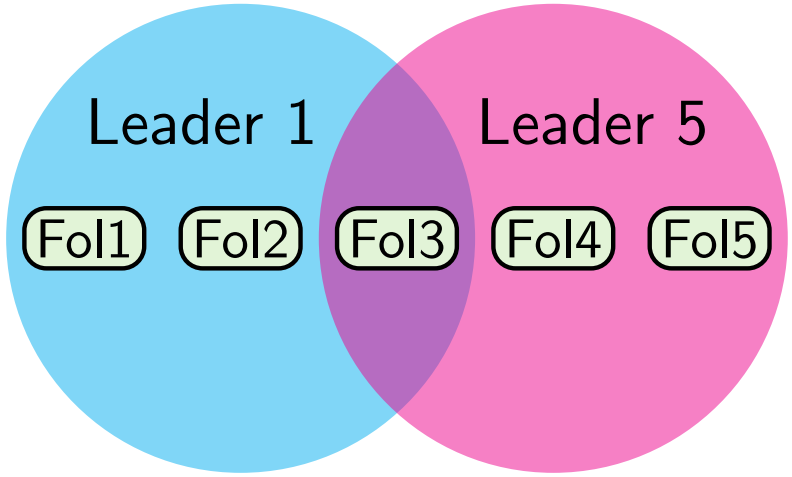


store(epoch, slot, operation) → ok | current epoch

- If epoch = current, return ok, else return current

Concurrency still safe

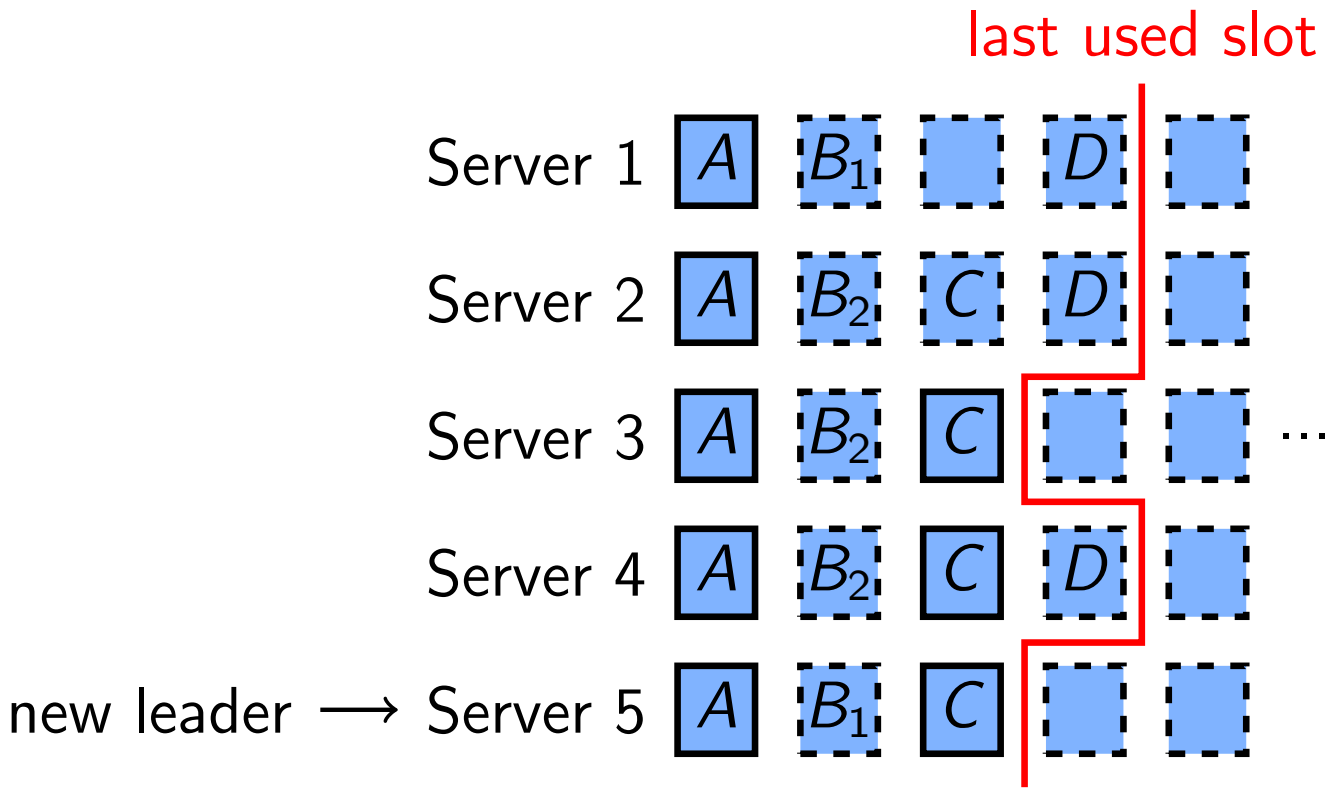
- Previous example: leaders operated in lockstep
- Still safe with concurrent operation
- If both leaders each call a majority of followers, at least one server will hear from both



- After overthrow, old leader's store calls will never succeed on a majority of followers

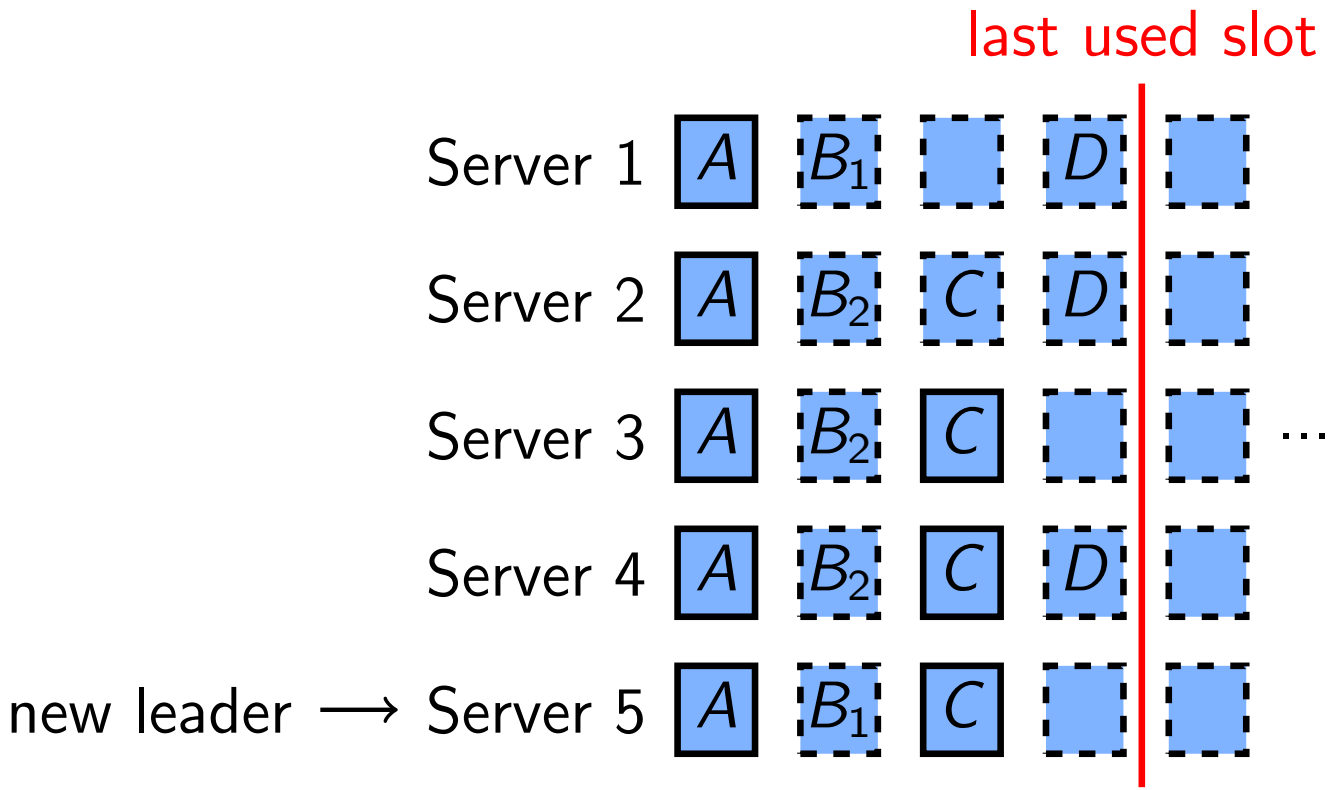
Responsibilities of leadership

1. To advance the local state machine, finalize all slots locally up to last used slot
2. To speed up future recoveries, replicate operations and finalized flags widely



Responsibilities of leadership

1. To advance the local state machine, finalize all slots locally up to last used slot
2. To speed up future recoveries, replicate operations and finalized flags widely



Read from remote logs

read(slot) $\rightarrow$ epoch, operation

- Safe to finalize an operation if a majority of servers have stored it.

Read from remote logs

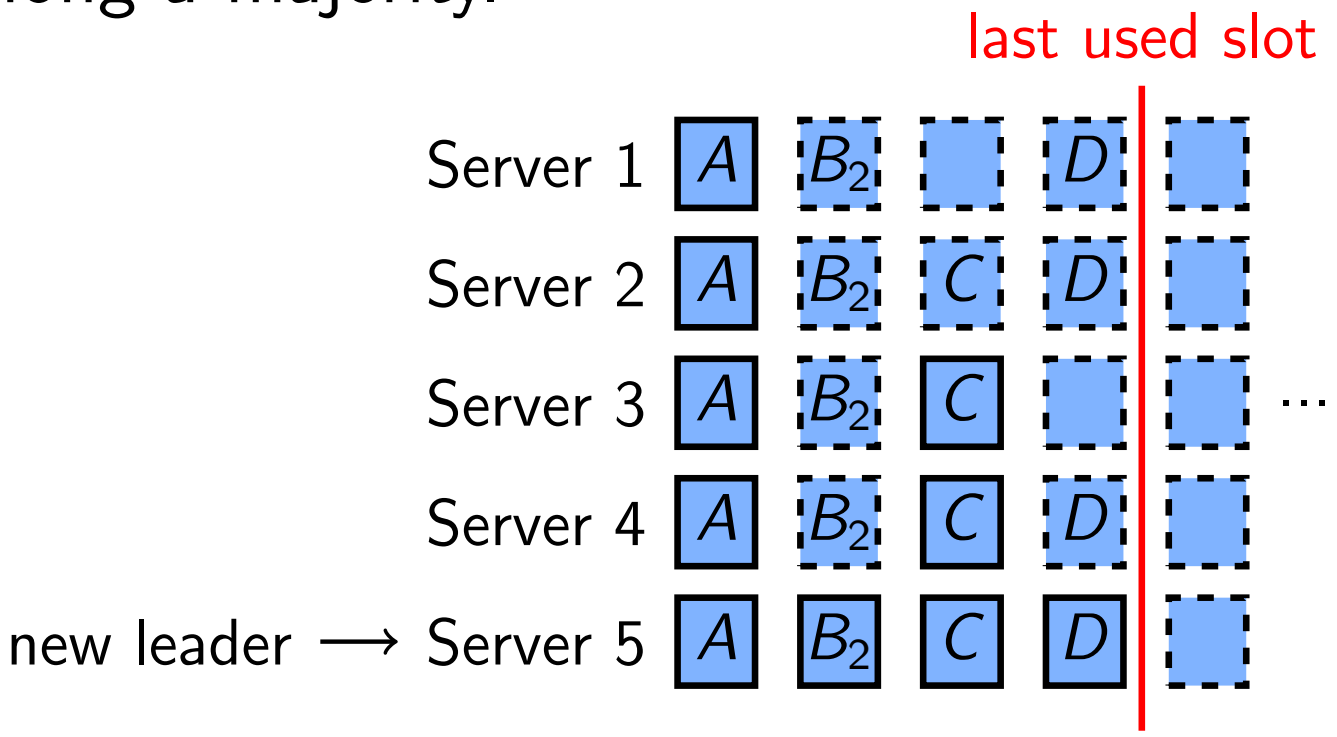
read(slot) $\rightarrow$ epoch, operation

- Safe to finalize an operation if a majority of servers have stored it.
- Safe to store the operation with the latest epoch among a majority.

Read from remote logs

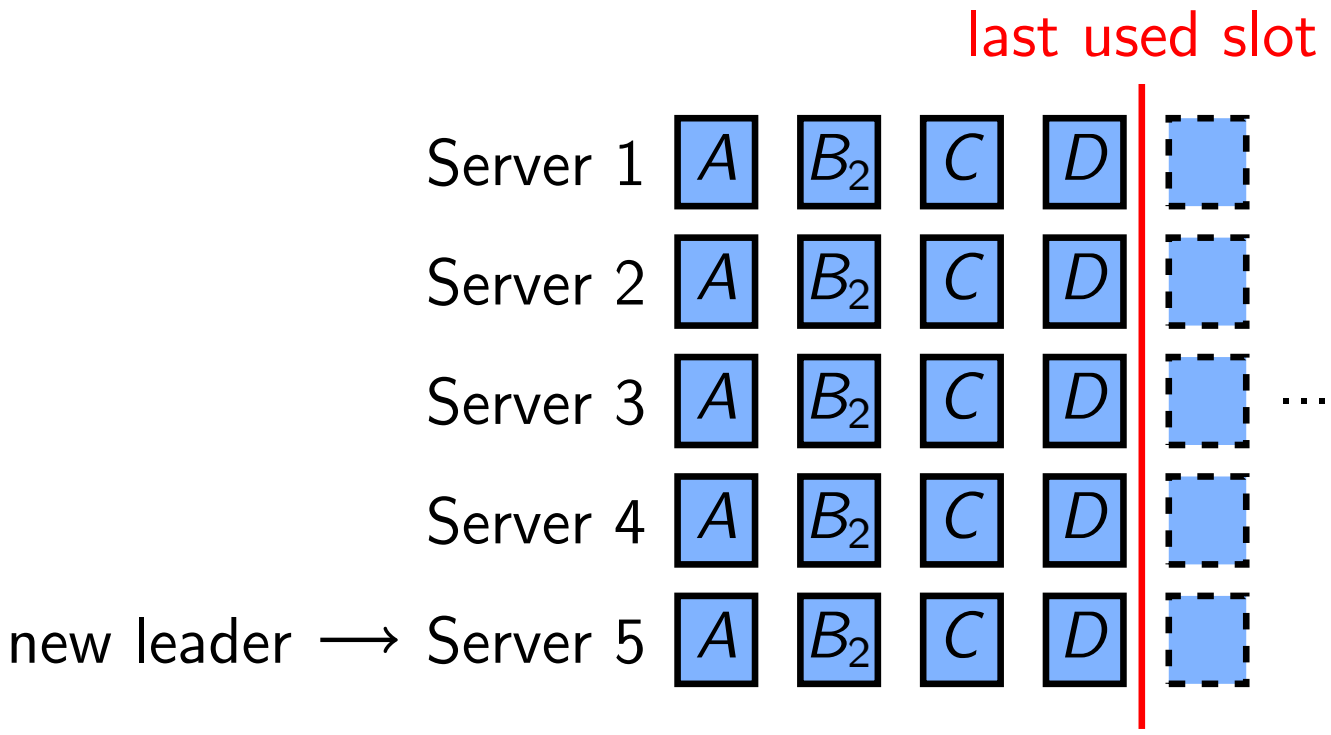
read(slot) $\rightarrow$ epoch, operation

- Safe to finalize an operation if a majority of servers have stored it.
- Safe to store the operation with the latest epoch among a majority.



Replicate more widely

- To replicate operations and finalize slots from previous leaders on followers, need to know what they're missing.
- query() $\rightarrow$ first unfinalized slot number
- Leader uses store and finalize to fill in the gap.



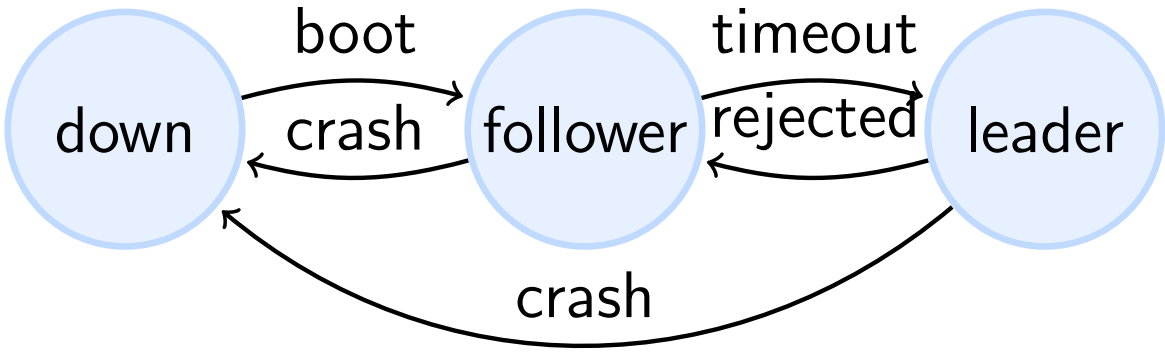
Performance

It's a three-phase protocol, but:

- overthrow is used rarely
- finalize only speeds up recovery, so it can be deferred

Common case: one round of RPCs

Encouraging one leader at a time



- Timeouts make passive servers become leaders
  - Leader issues heartbeats in case of inactivity
  - Timeout period chosen randomly so not all servers wake up at once
- Epoch numbers select arbitrarily between the available leaders
  - If a leader's store is rejected, it becomes passive.

Issues not covered

Who do clients talk to?

- Guess a leader, redirected if wrong

Issues not covered

Who do clients talk to?

- Guess a leader, redirected if wrong

Read-only operations

- Leases or RPC, don't need to go to disk

Issues not covered

Who do clients talk to?

- Guess a leader, redirected if wrong

Read-only operations

- Leases or RPC, don't need to go to disk

What's persisted?

- Followers persist everything, leaders don't

Issues not covered

Who do clients talk to?

- Guess a leader, redirected if wrong

Read-only operations

- Leases or RPC, don't need to go to disk

What's persisted?

- Followers persist everything, leaders don't

How is space reclaimed?

- Snapshots (typically) or cleaning (John's students)

Issues not covered

Who do clients talk to?

- Guess a leader, redirected if wrong

Read-only operations

- Leases or RPC, don't need to go to disk

What's persisted?

- Followers persist everything, leaders don't

How is space reclaimed?

- Snapshots (typically) or cleaning (John's students)

How is cluster membership managed?

- Submit request to state machine

Issues not covered

Who do clients talk to?

- Guess a leader, redirected if wrong

Read-only operations

- Leases or RPC, don't need to go to disk

What's persisted?

- Followers persist everything, leaders don't

How is space reclaimed?

- Snapshots (typically) or cleaning (John's students)

How is cluster membership managed?

- Submit request to state machine

How can we get linearizable semantics?

- This gets you at-least-once semantics;
  use sequence numbers for exactly-once

Questions and feedback

1. Have heard of Paxos?
2. Know when to apply Paxos?
3. Understand Paxos?
4. Fear Paxos?

RPCs

- overthrow(new epoch) $\rightarrow$ last used slot | current epoch
- store(epoch, slot, operation) $\rightarrow$ ok | current epoch
- finalize(epoch, slot) $\rightarrow$ ok | missing
- read(slot) $\rightarrow$ epoch, operation
- query() $\rightarrow$ first unfinalized slot number