▼ **Consistency under partial failure: 2PC & replicated state machine**

1 ✦ Consistency under partial failure: 2PC & replicated state machine
2 ✦ Outline of today's content
▼ Multi-site transaction and two-phase commit
3 ✦ Motivating case: Multi-site transaction
4 ✦ Multi-site transactions under two-phase commit
▼ Single-site commit and recovery
5 ✦ Recall: the commit point in a single-site transaction
6 ✦ Why is it the commit point? Because we can recover afterward
▼ 2PC under failures
7 ✦ Failure: lost prepare
8 ✦ Failure: worker failure during prepare
9 ✦ Multi-site atomicity
10 ✦ What two-phase commit adds to the log
11 ✦ Missing a vote: abort preserves atomicity
12 ✦ More failure cases: lost commit message
13 ✦ Failure: worker failure during commit
14 ✦ Failure: coordinator failure during prepare
15 ✦ Failure: coordinator failure during commit
16 ✦ Extension to single-cite atomicity methods
17 ✦ Summary of two-phase commit
▼ 2PC and CAP
18 ✦ Case study: an e-commerce site with two zones
19 ✦ Another system challenge: network partition
20 ✦ The CAP theorem
21 ✦ First, we clarify what we mean by availability
22 ✦ Analysis: why can't we keep both C and A? (An informal argument)
23 ✦ Analysis: what do we give up during a partition?
24 ✦ Our current 2PC is a CP protocol
25 ✦ Replication is everywhere
26 ✦ Challenge: inconsistency due to reordered operations
27 ✦ Question: do replicas see the same operation order?
▼ RSM
28 ✦ Replicated state machine (RSM) for pessimistic replication
29 ✦ Primary-backup model revisited
30 ✦ What if the primary fails?
31 ✦ Primary-backup model with availability: the first try
32 ✦ Multiple monitors under a network partition
▼ View server
33 ✦ View Server: one server to decide who is the primary
34 ✦ Question: can two servers simultaneously become primaries?
35 ✦ Rules for handling two live primaries
36 ✦ Consider S1 partitioned from the VS
▼ What if a backup fails
37 ✦ Question: how many backups must ack before the primary replies?
38 ✦ Question: which backup becomes the primary?
39 ✦ Question: when is a write reliable?
40 ✦ Quorum
41 ✦ Put it all together: the complete data path of primary-backup
42 ✦ Final issue: what if view server fails?
43 ✦ 参考文献

IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Consistency under partial failure: 2PC & replicated state machine

IPADS, Shanghai Jiao Tong University

Outline of today's content

2PC, and how to handle partial failure

The issue of 2PC: does not support availability by default

Supporting availability & consistency (linearizability)

- Approach #1: Primary-backup replication
- Approach #2: Paxos & Multi-Paxos — the next lecture

Motivating case: Multi-site transaction

Coordinator sends multiple operations (e.g., `deposit`) to different servers to execute one transaction

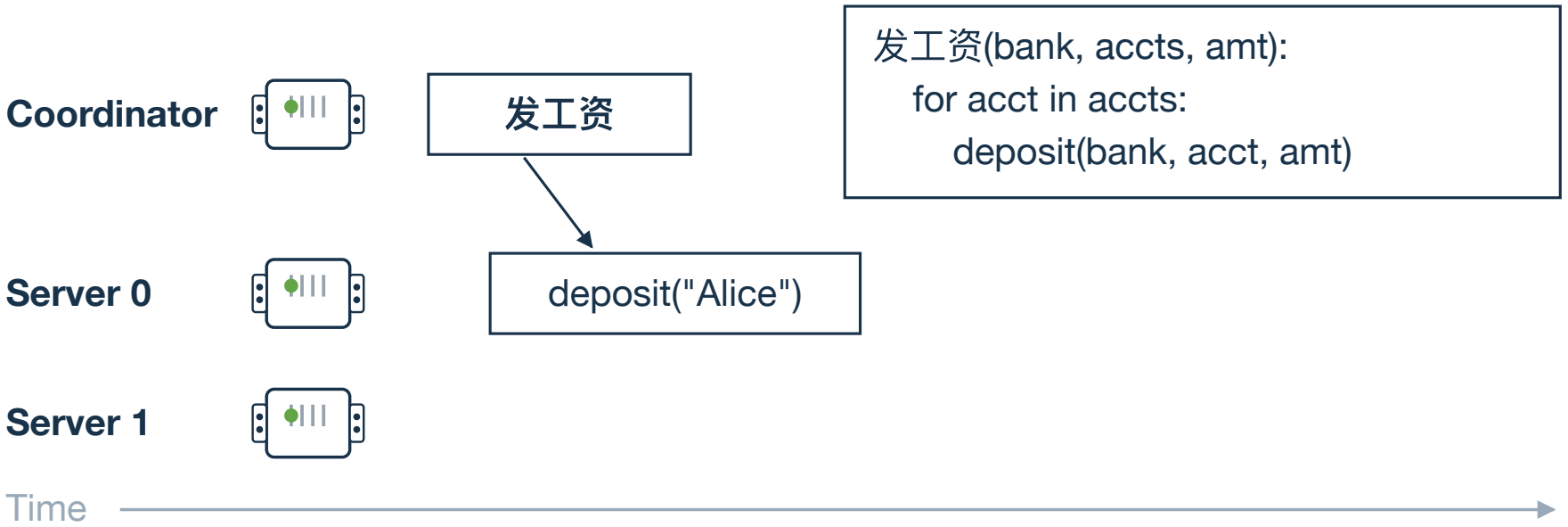
- They use RPCs to send requests to the server to execute transaction



Motivating case: Multi-site transaction

Coordinator sends multiple operations (e.g., `deposit`) to different servers to execute one transaction

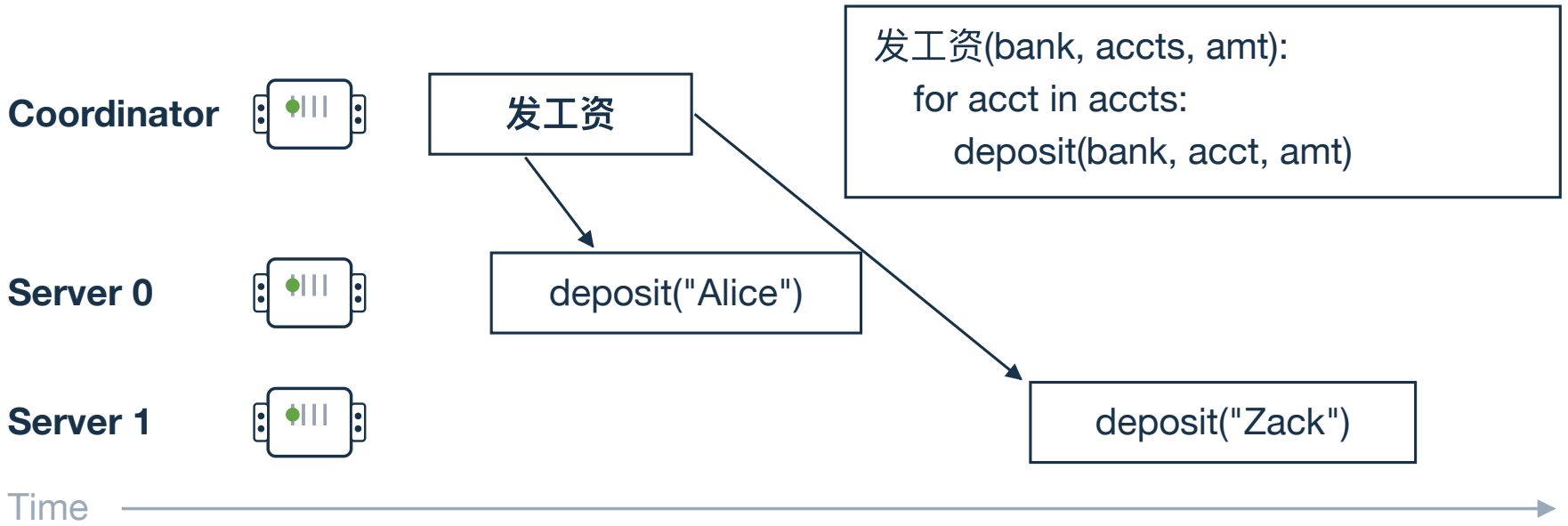
- They use RPCs to send requests to the server to execute transaction



Motivating case: Multi-site transaction

Coordinator sends multiple operations (e.g., `deposit`) to different servers to execute one transaction

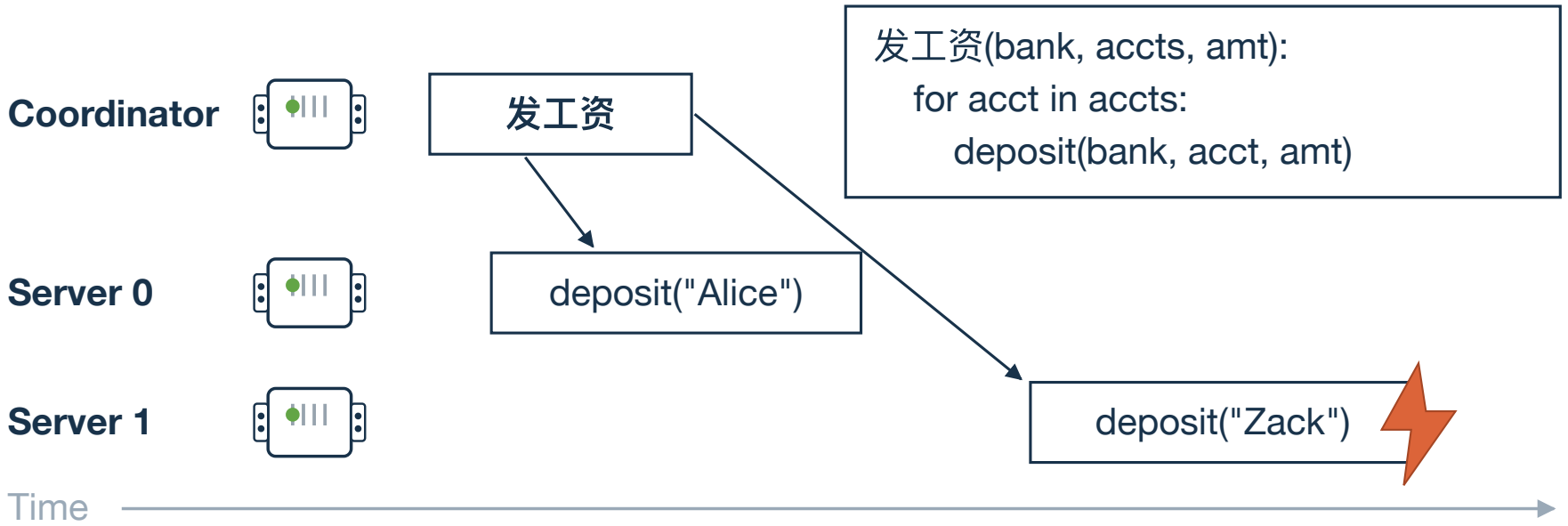
- They use RPCs to send requests to the server to execute transaction



Motivating case: Multi-site transaction

Coordinator sends multiple operations (e.g., `deposit`) to different servers to execute one transaction

- They use RPCs to send requests to the server to execute transaction



What if one server commits and the other aborts?

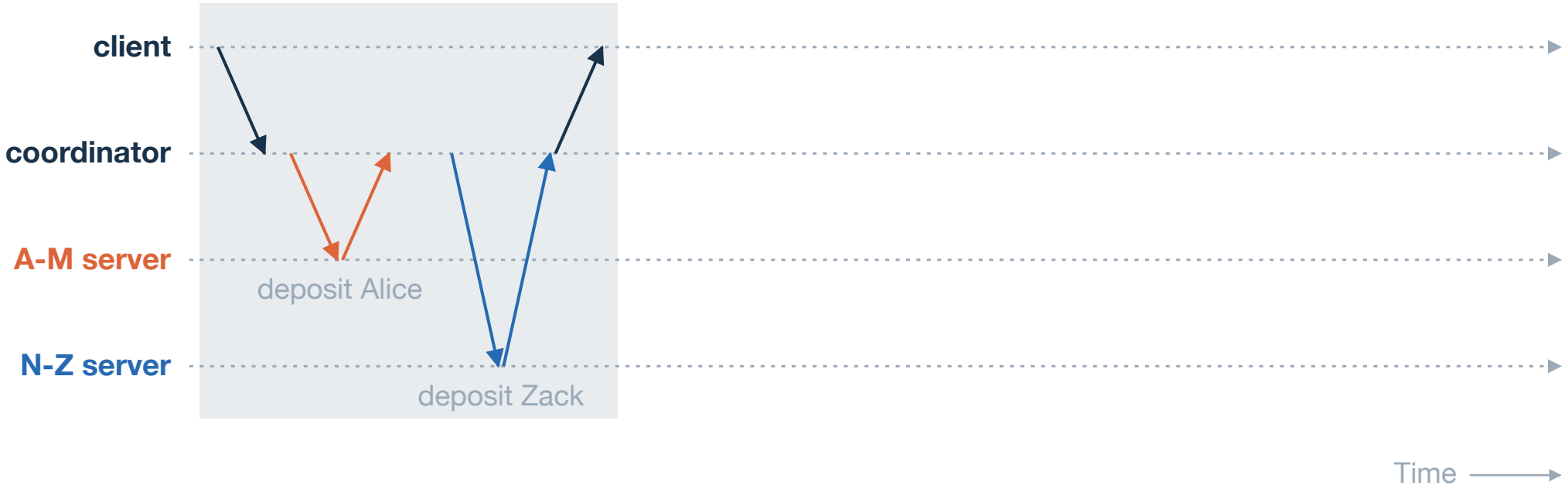
Goal: all sites either all commit or all abort,

under partial failure!

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below

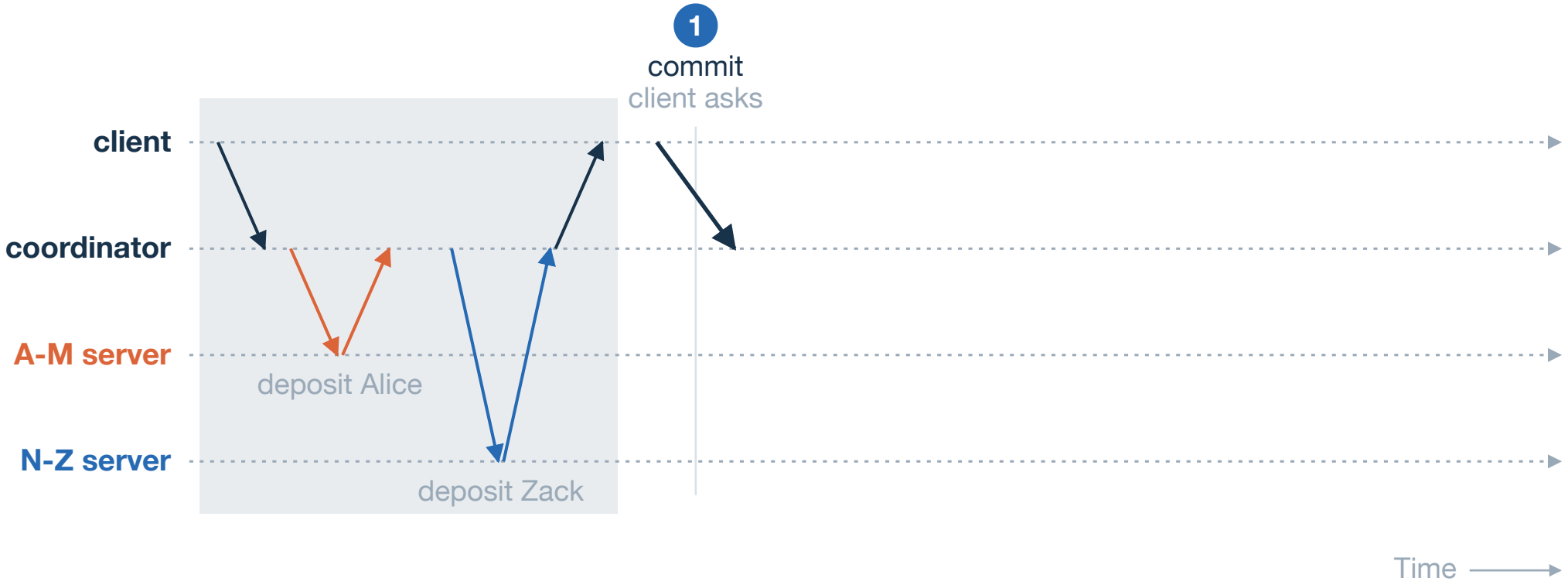


assume all parts of the transactions prior to commit have happened

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below



assume all parts of the transactions prior to commit have happened

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below

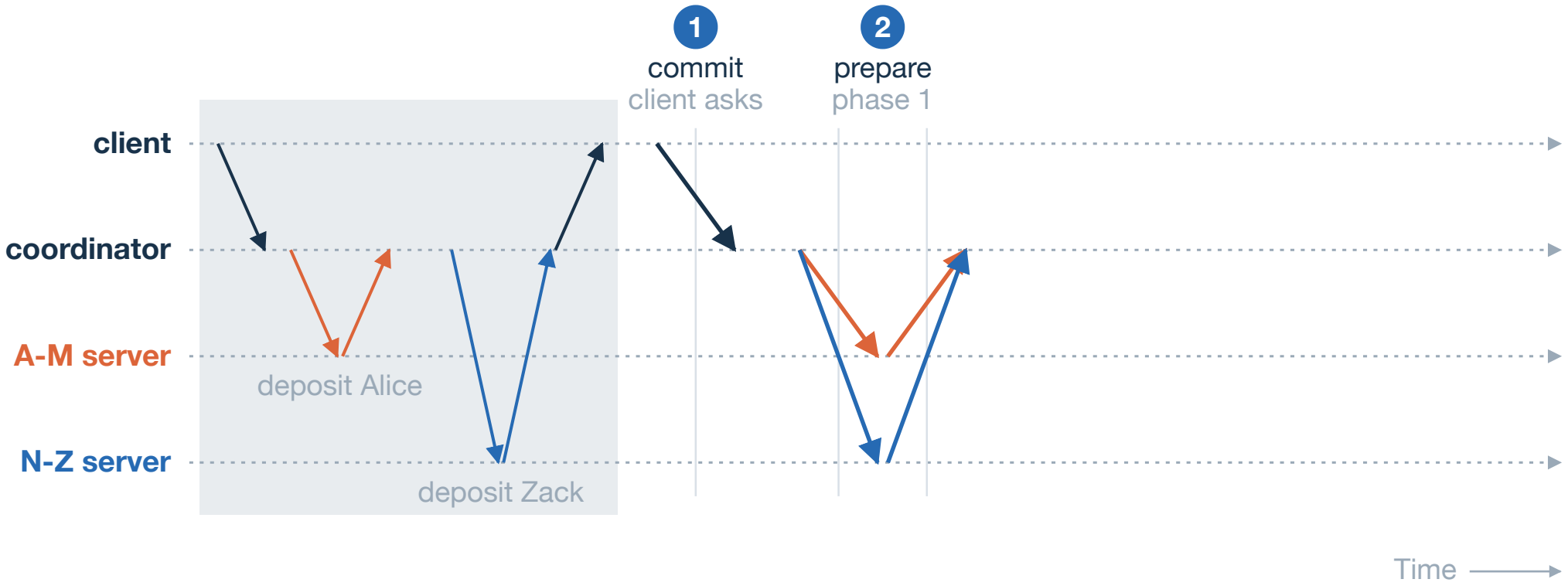


assume all parts of the transactions prior to commit have happened

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below

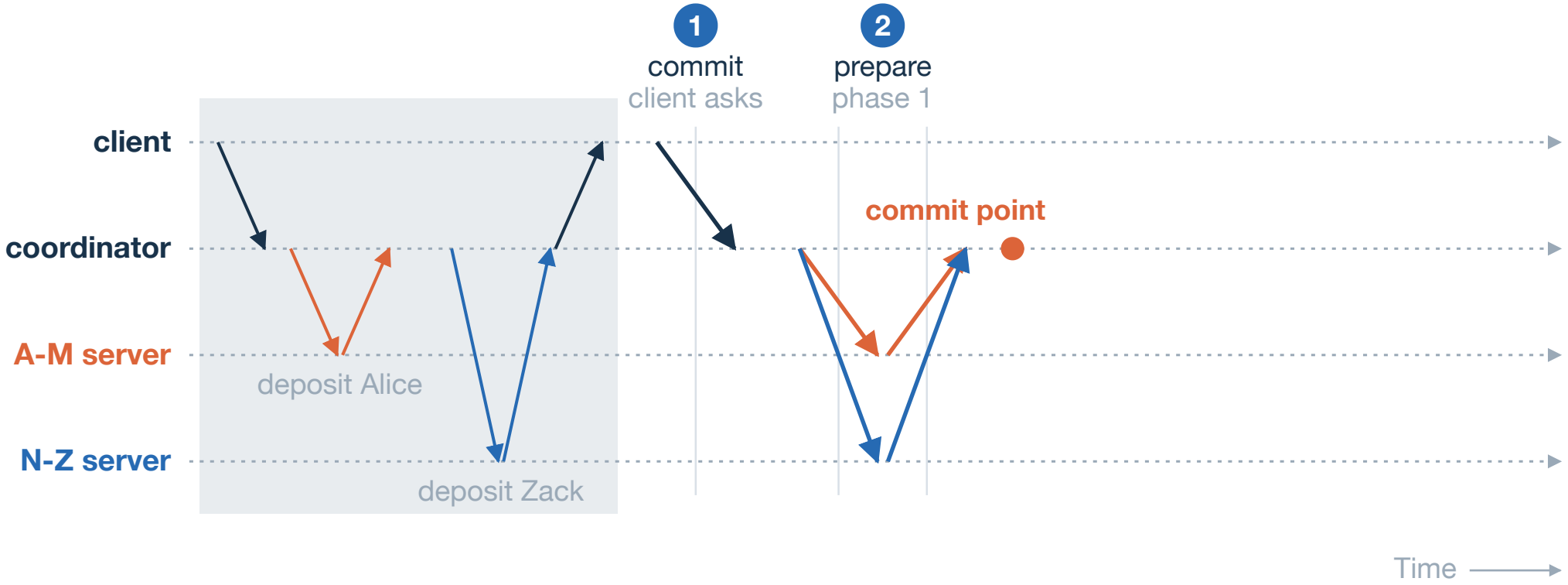


assume all parts of the transactions prior to commit have happened

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below

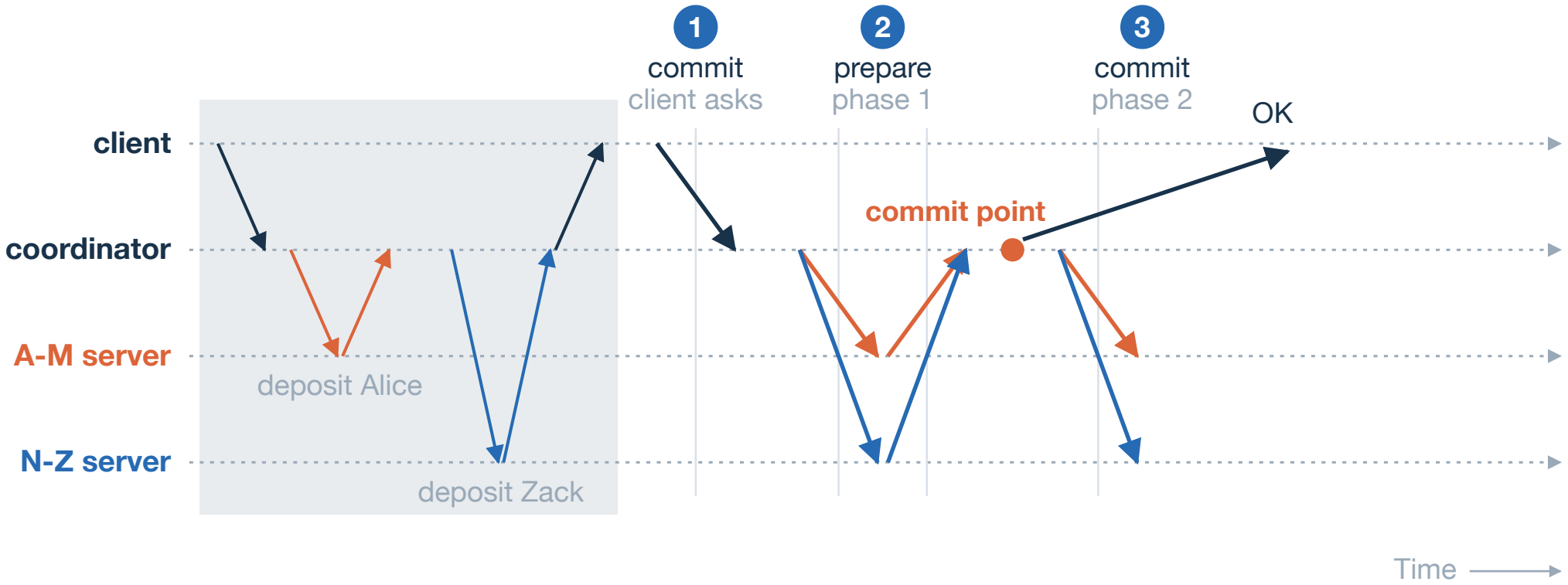


assume all parts of the transactions prior to commit have happened

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below

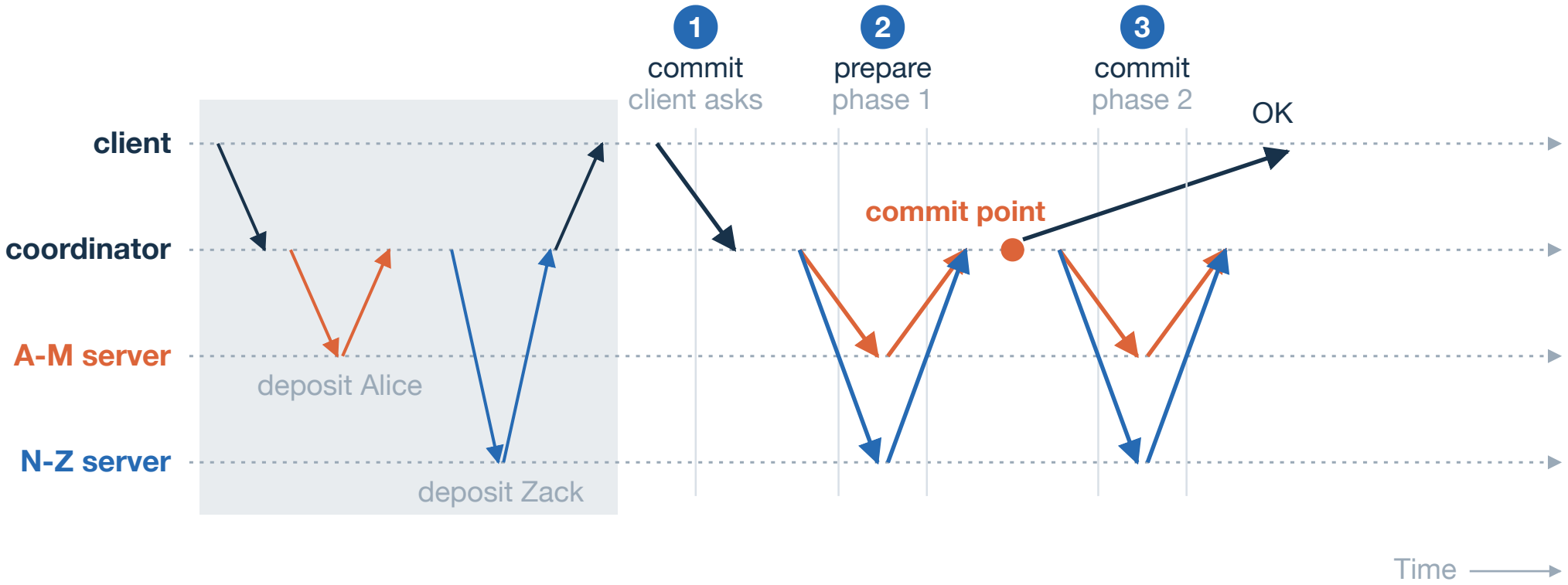


assume all parts of the transactions prior to commit have happened

Multi-site transactions under two-phase commit

Idea: the coordinator is responsible for committing all the TXs, but not others

- The method: By sending RPCs or Messages to all the other servers
- Those servers are the workers: the A-M and N-Z servers below



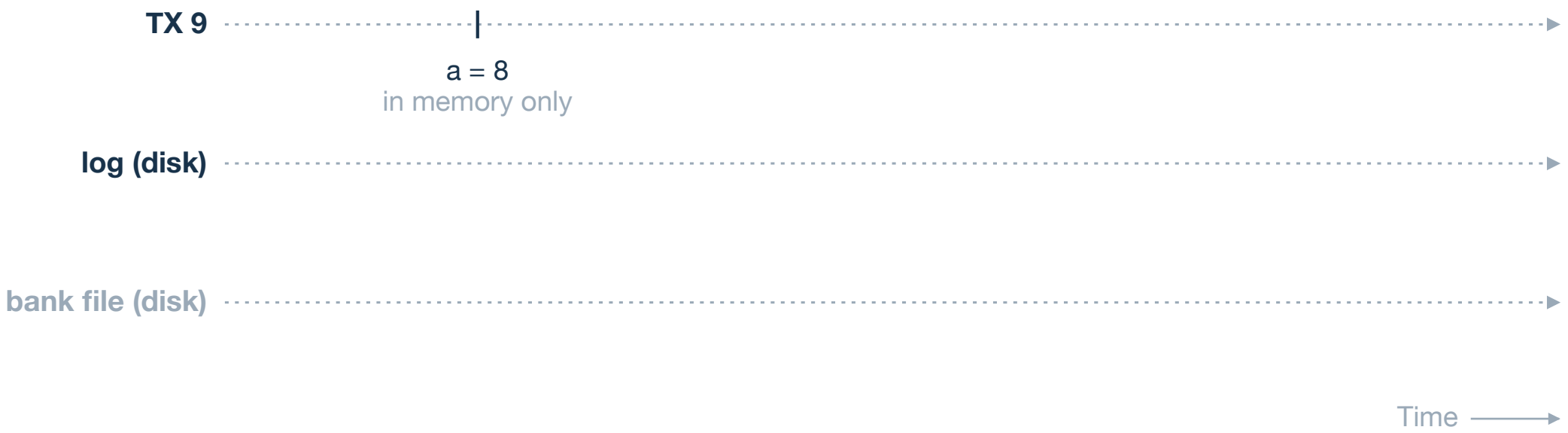
assume all parts of the transactions prior to commit have happened

Recall: the commit point in a single-site transaction

Assuming the redo-only logging for simplicity

Buffer the updates in memory; one log entry makes them durable & atomic

- So commit point is one local disk write: the entry reaching the disk

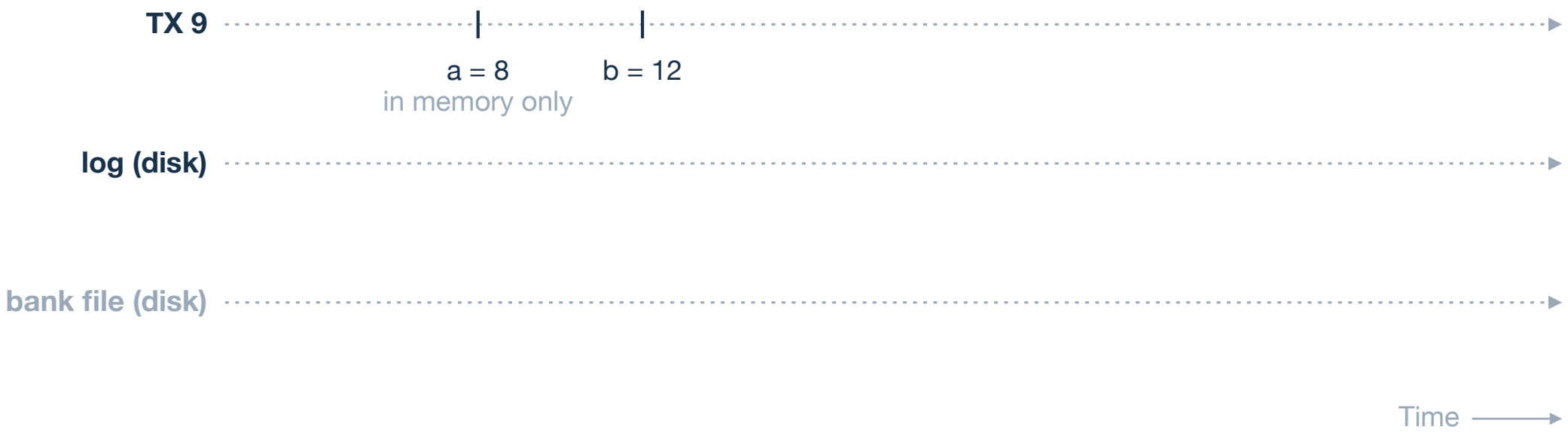


Recall: the commit point in a single-site transaction

Assuming the redo-only logging for simplicity

Buffer the updates in memory; one log entry makes them durable & atomic

- So commit point is one local disk write: the entry reaching the disk

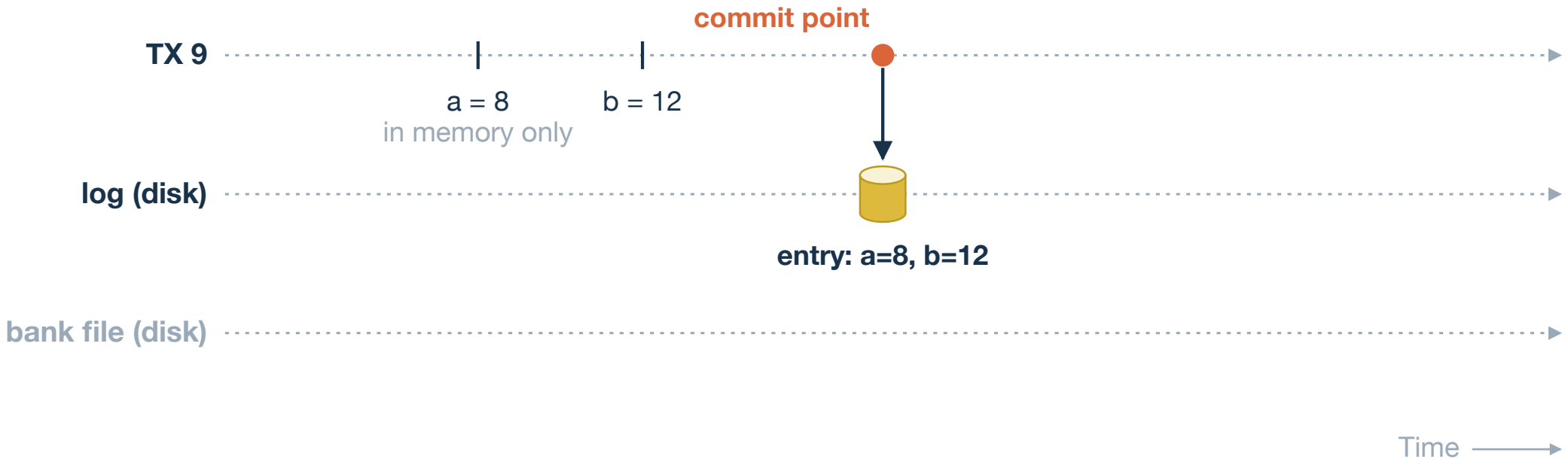


Recall: the commit point in a single-site transaction

Assuming the redo-only logging for simplicity

Buffer the updates in memory; one log entry makes them durable & atomic

- So commit point is one local disk write: the entry reaching the disk

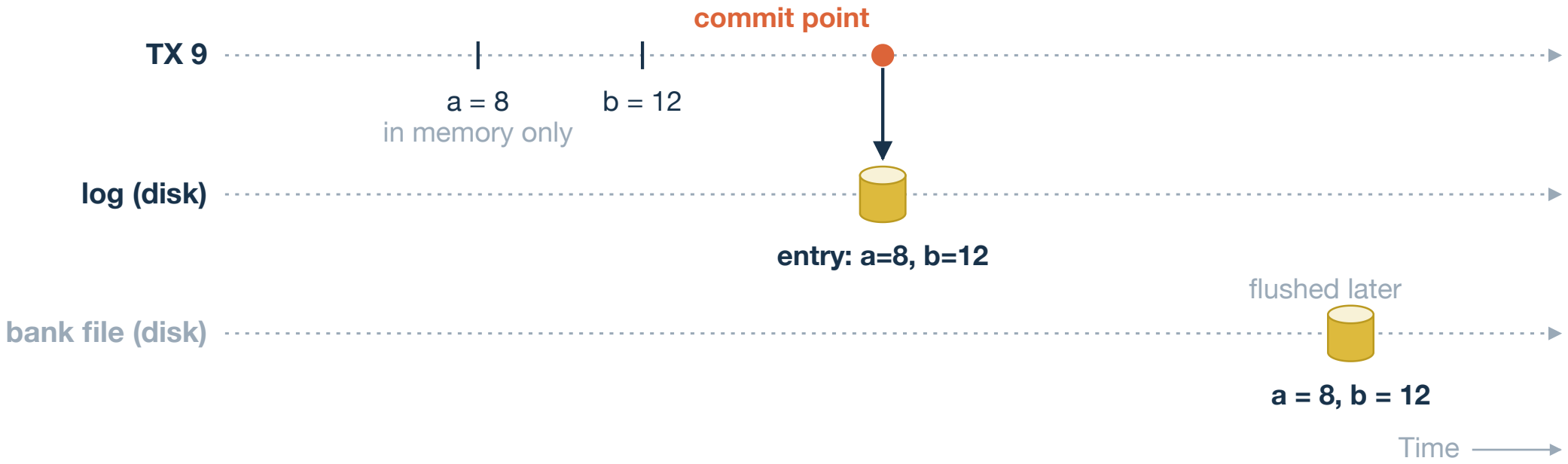


Recall: the commit point in a single-site transaction

Assuming the redo-only logging for simplicity

Buffer the updates in memory; one log entry makes them durable & atomic

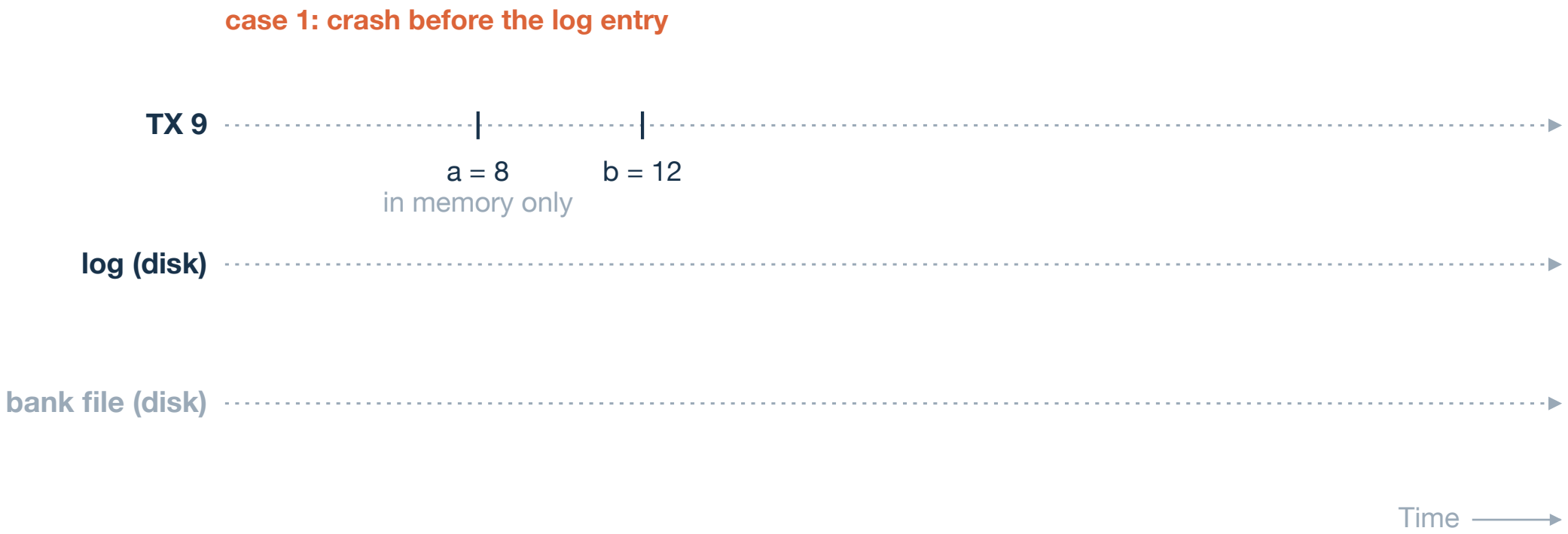
- So commit point is one local disk write: the entry reaching the disk



Why is it the commit point? Because we can recover afterward

After a reboot, the log alone decides

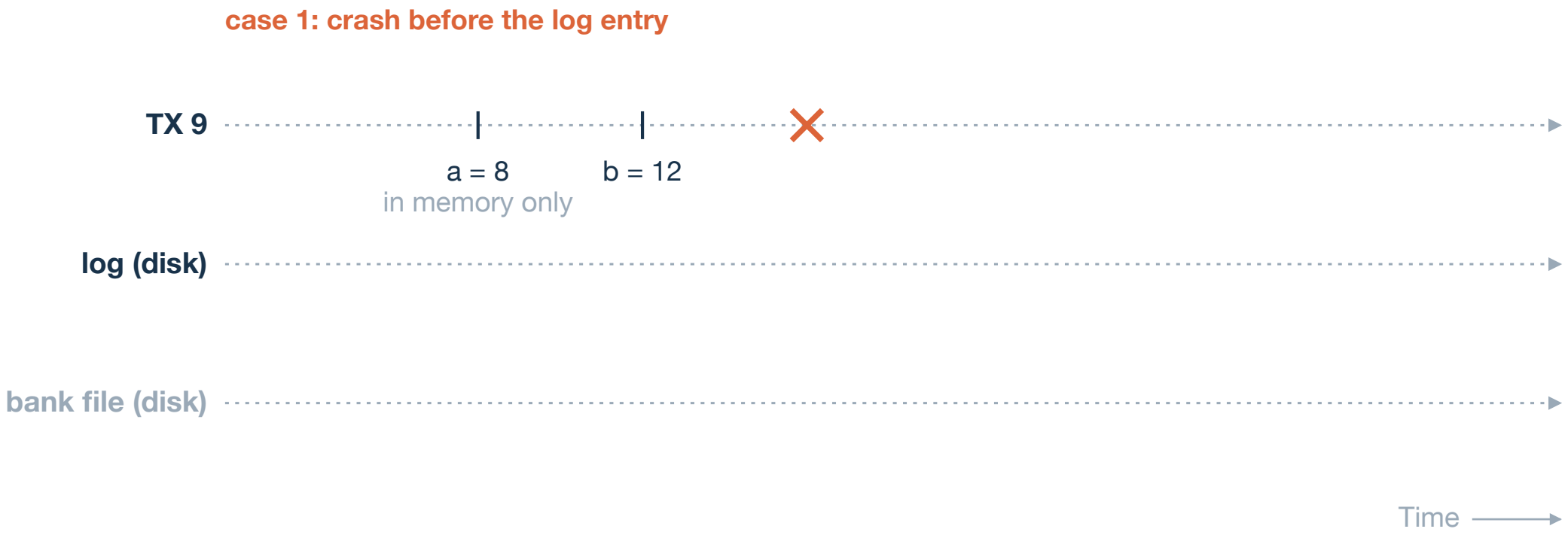
- A complete entry on disk: replay it. No entry: the transaction never happened



Why is it the commit point? Because we can recover afterward

After a reboot, the log alone decides

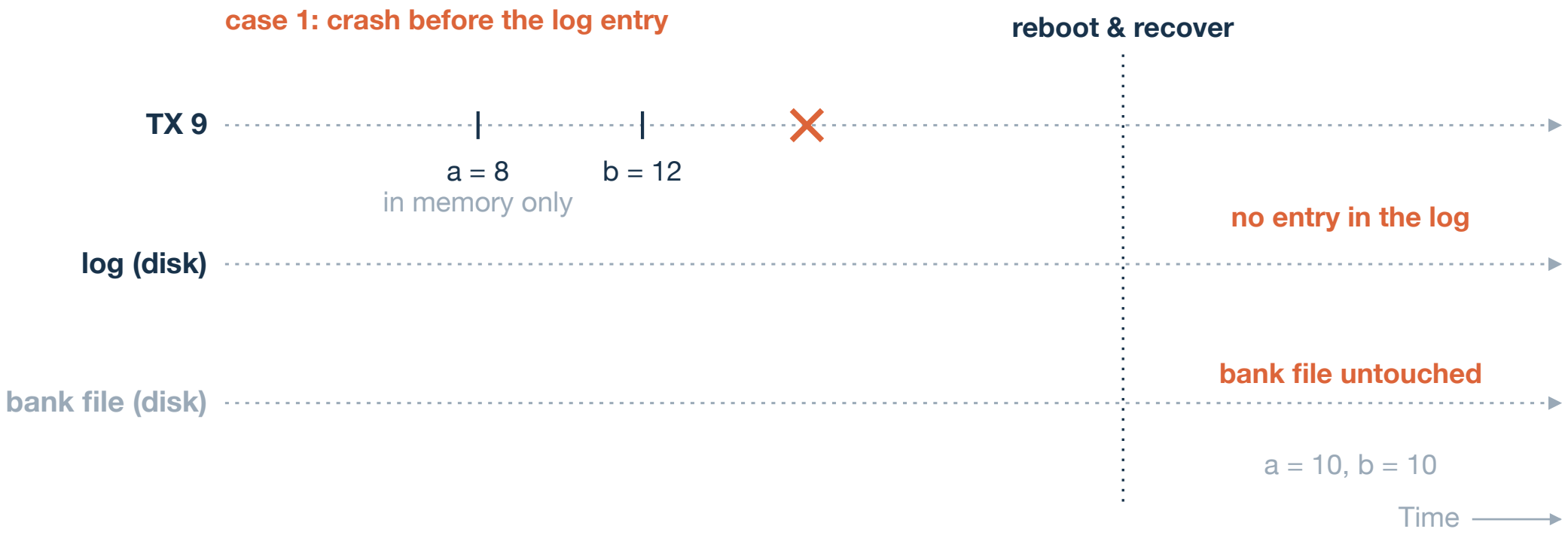
- A complete entry on disk: replay it. No entry: the transaction never happened



Why is it the commit point? Because we can recover afterward

After a reboot, the log alone decides

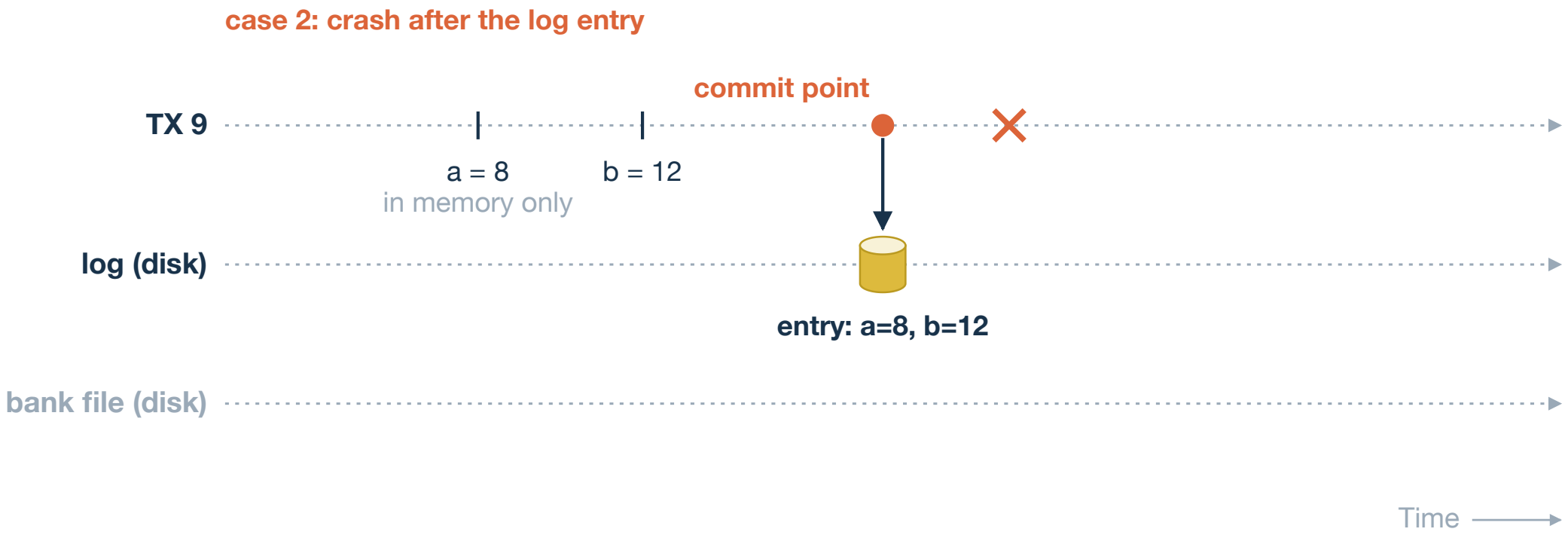
- A complete entry on disk: replay it. No entry: the transaction never happened



Why is it the commit point? Because we can recover afterward

After a reboot, the log alone decides

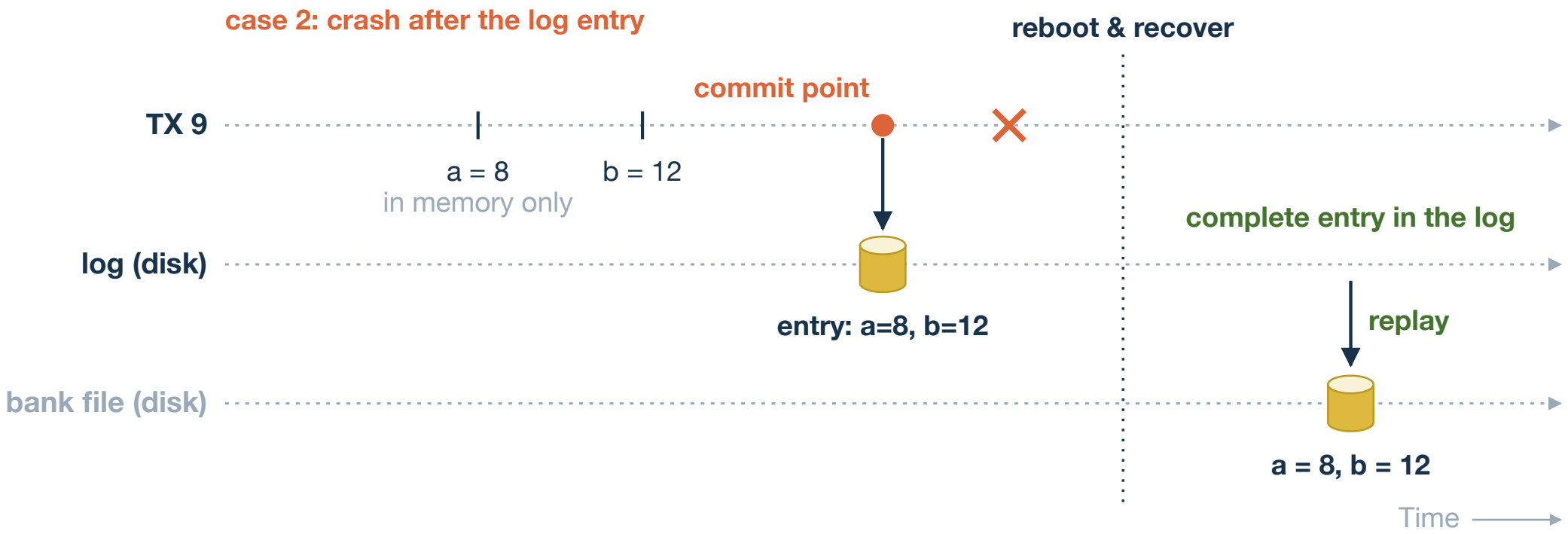
- A complete entry on disk: replay it. No entry: the transaction never happened



Why is it the commit point? Because we can recover afterward

After a reboot, the log alone decides

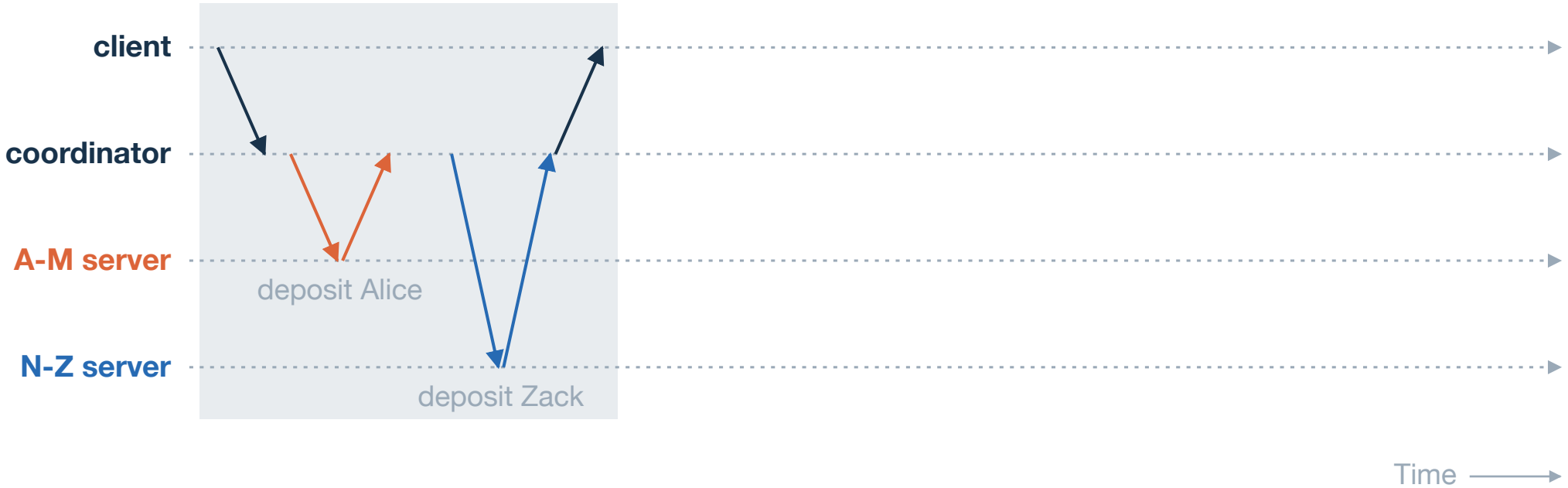
- A complete entry on disk: replay it. No entry: the transaction never happened



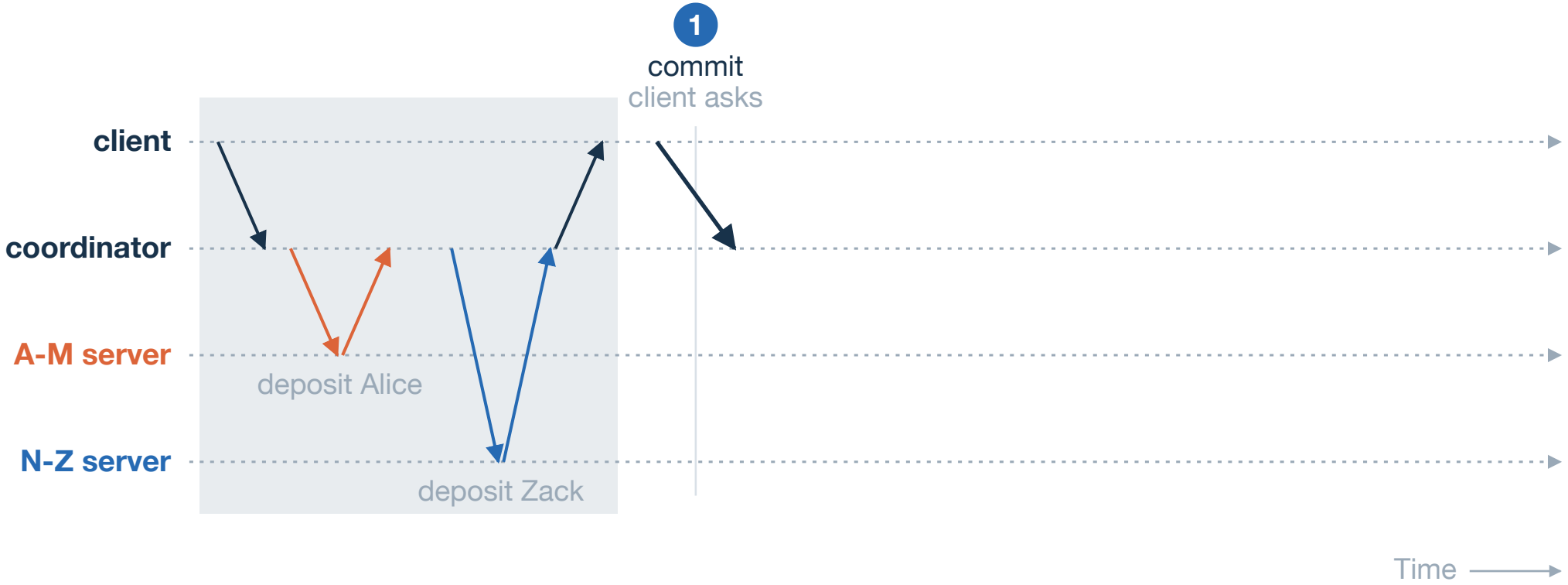
Question: is single-site logging enough for 2PC under partial failure?

Under 2PC, the system can partially fail in multiple ways

Failure: lost prepare



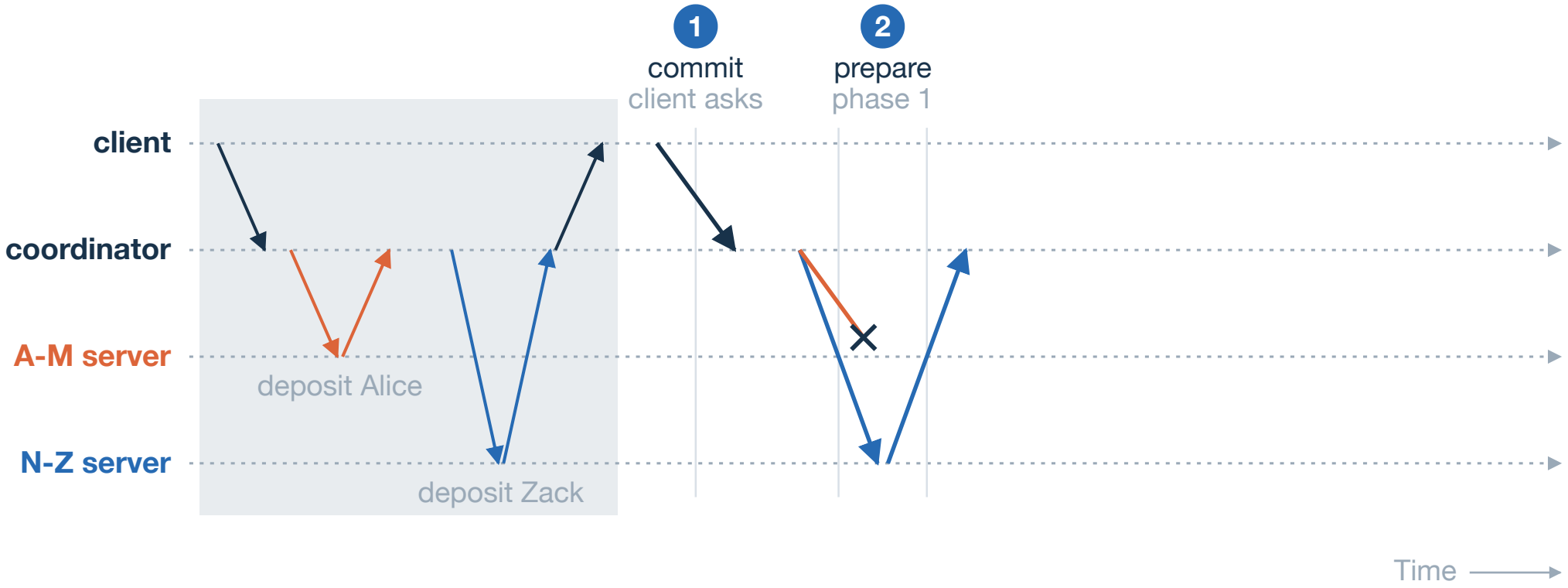
Failure: lost prepare



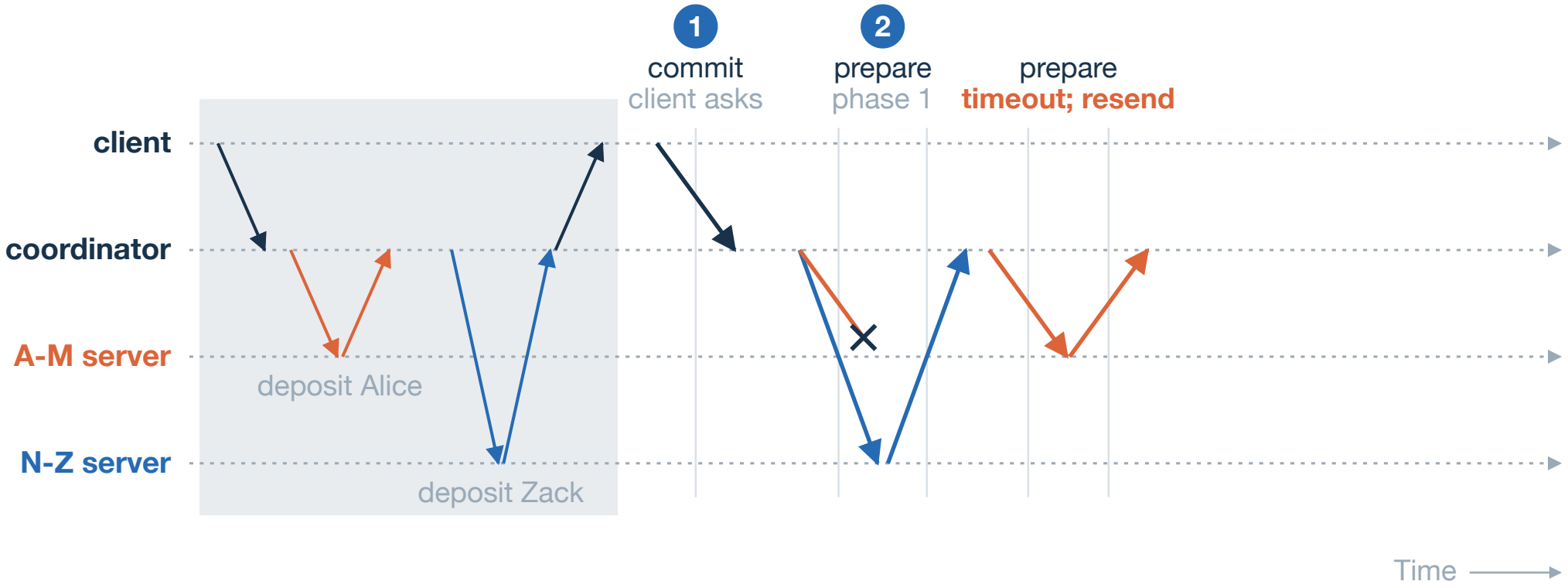
Failure: lost prepare



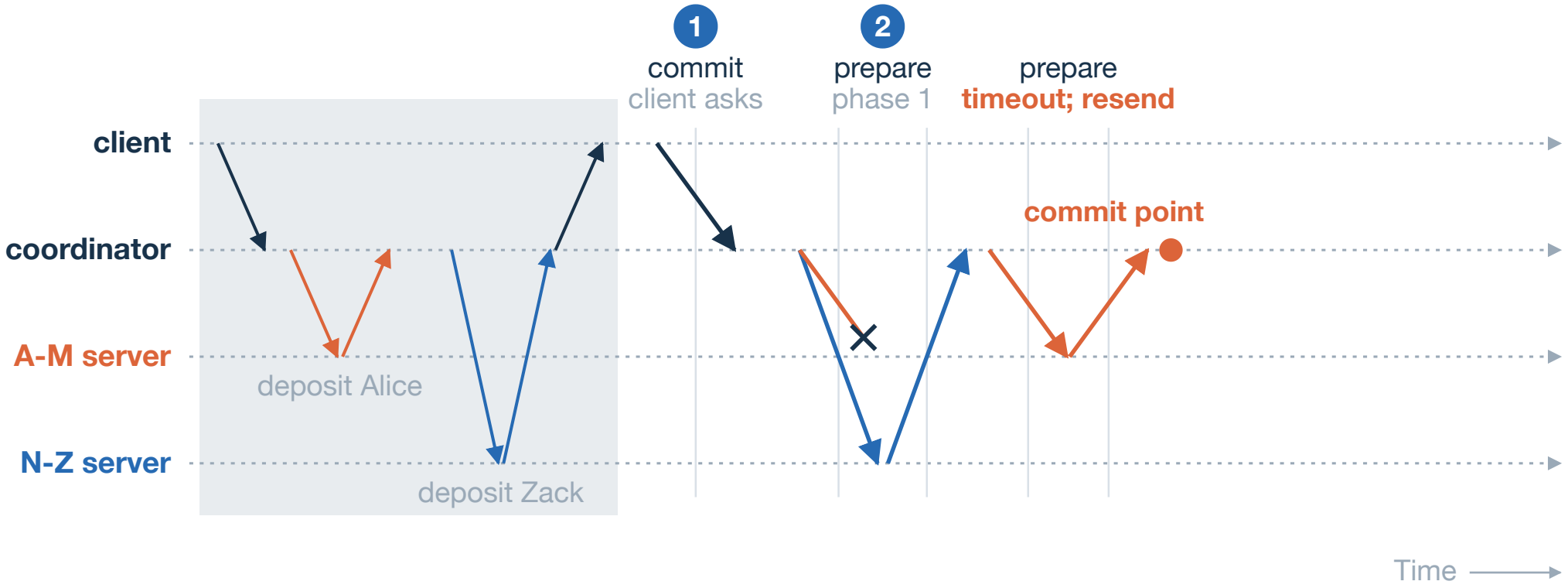
Failure: lost prepare



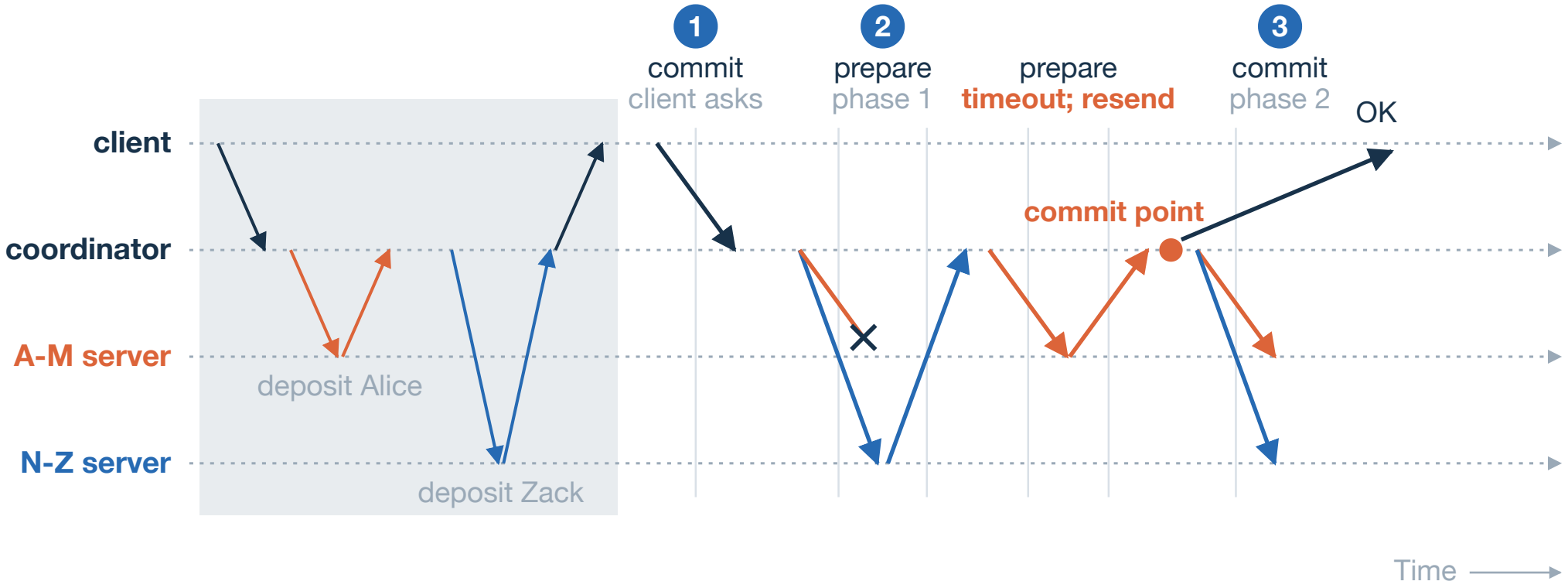
Failure: lost prepare



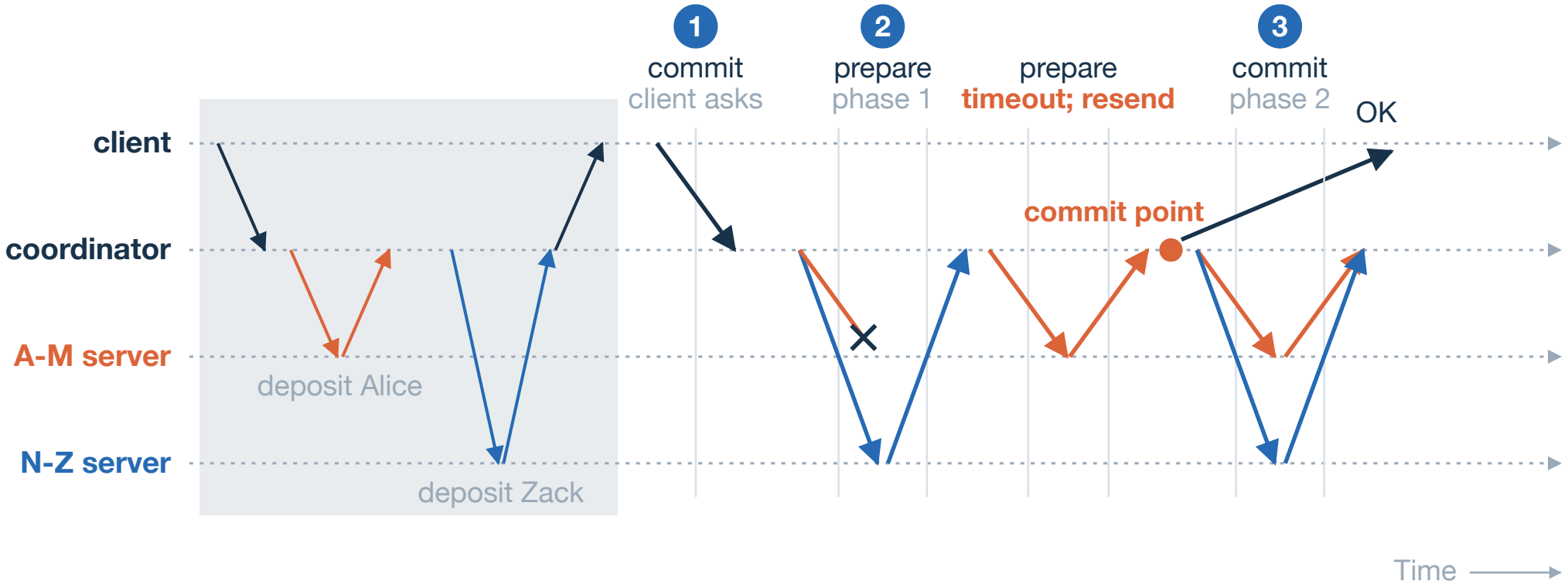
Failure: lost prepare



Failure: lost prepare



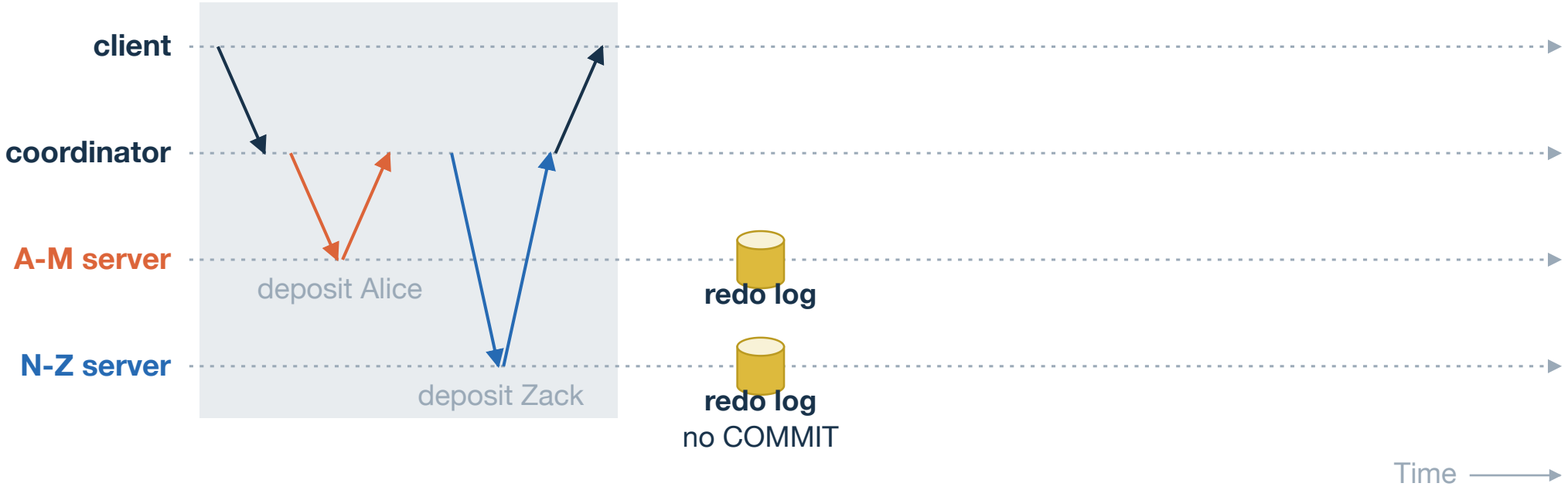
Failure: lost prepare



Handled: timeout & resend

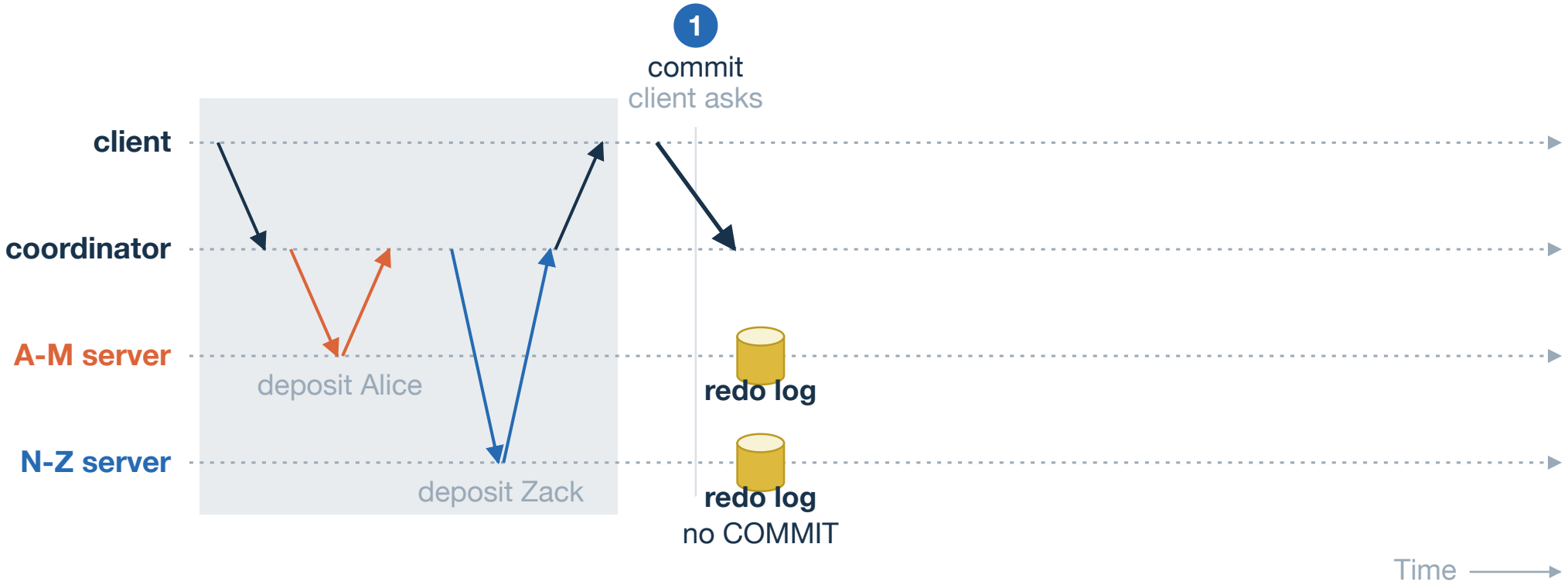
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



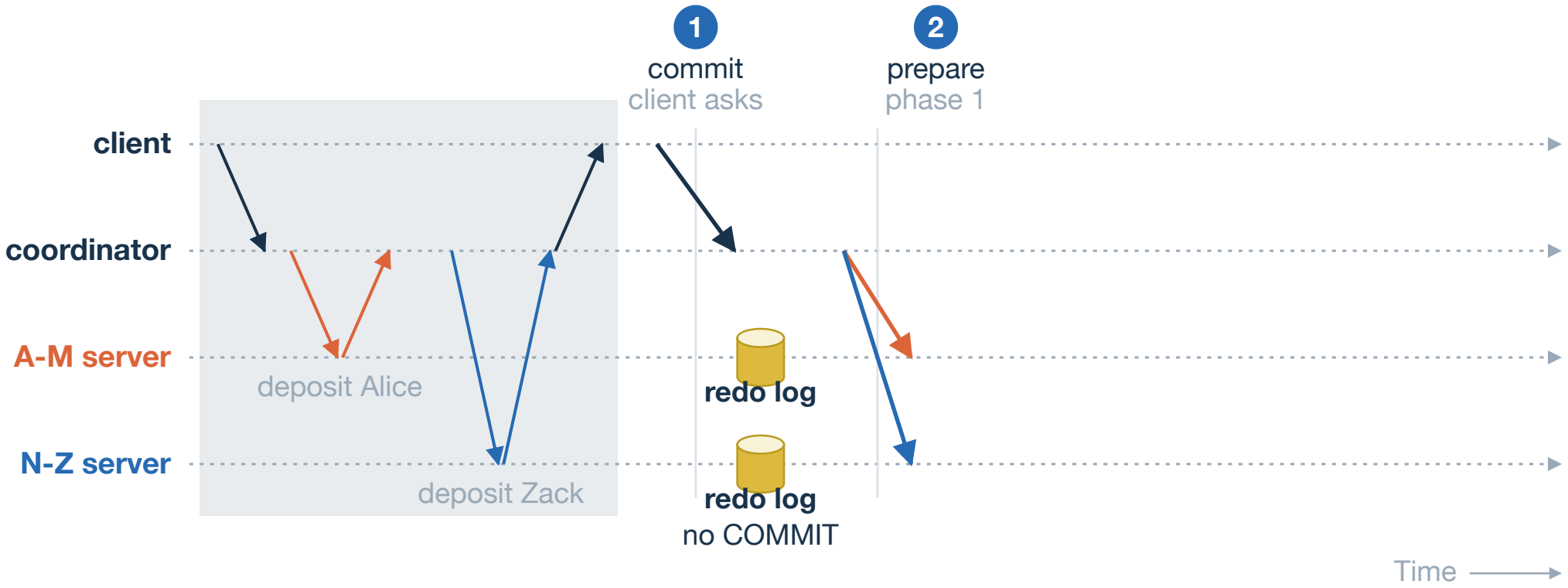
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



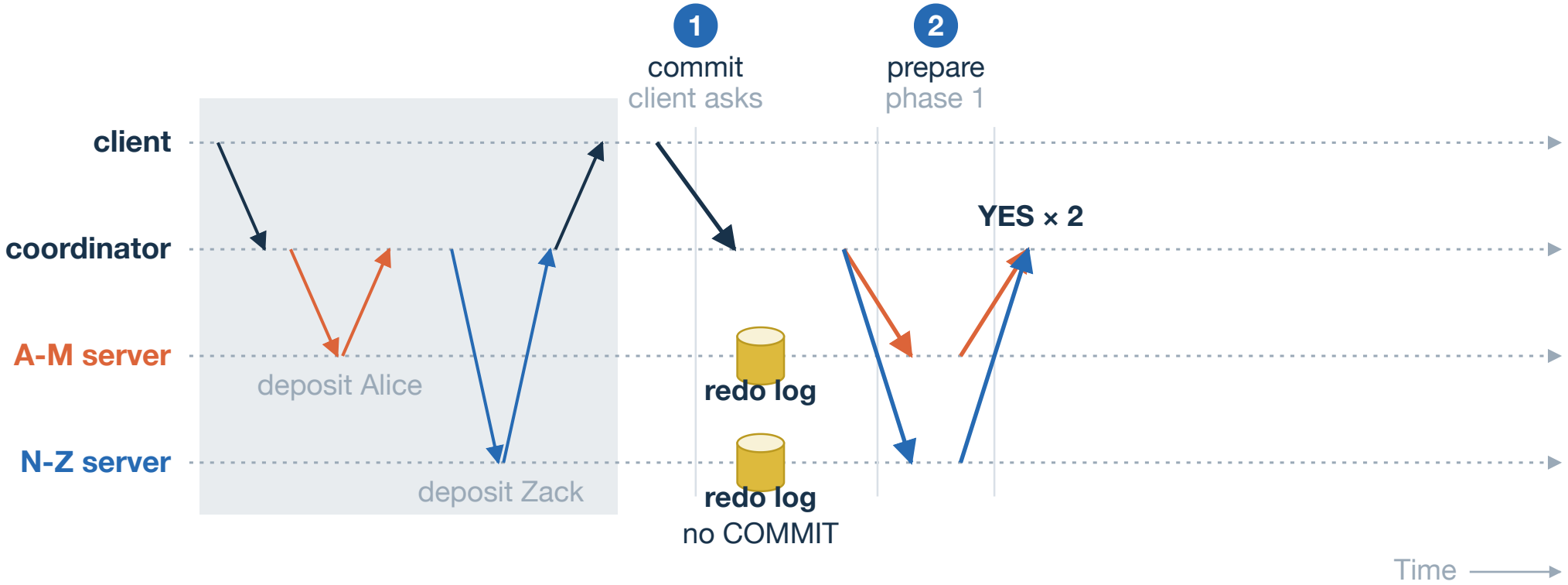
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



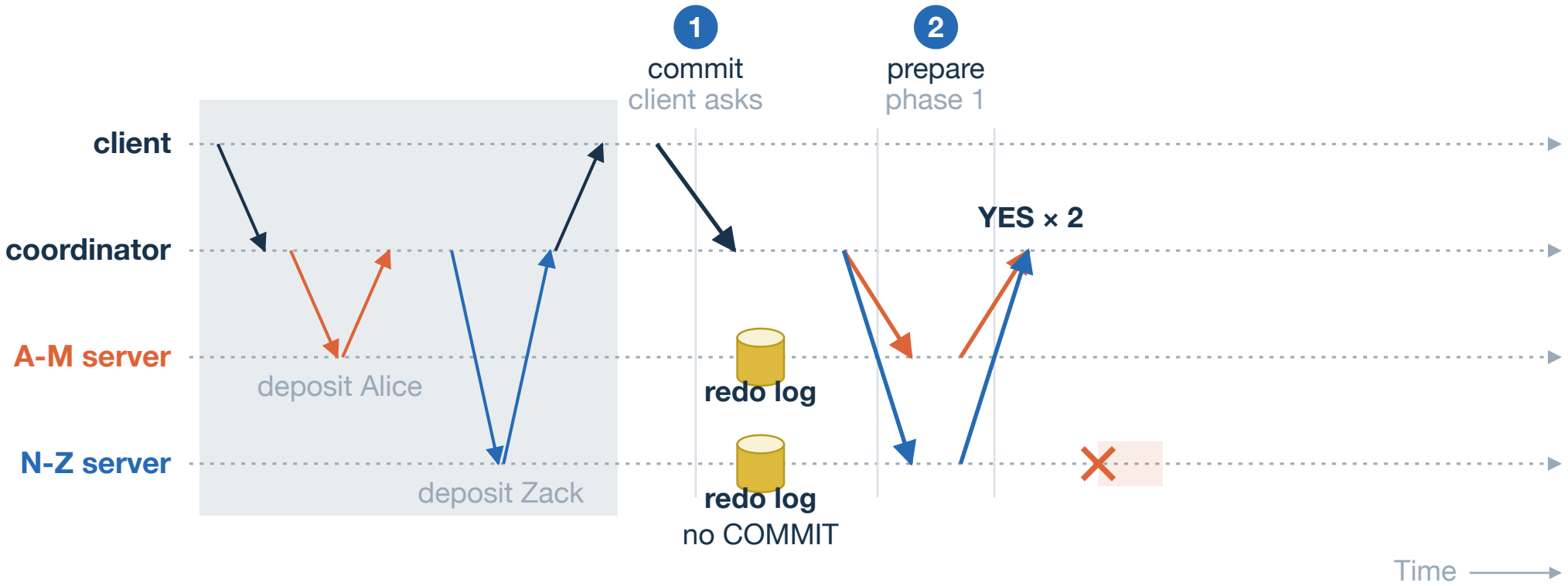
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



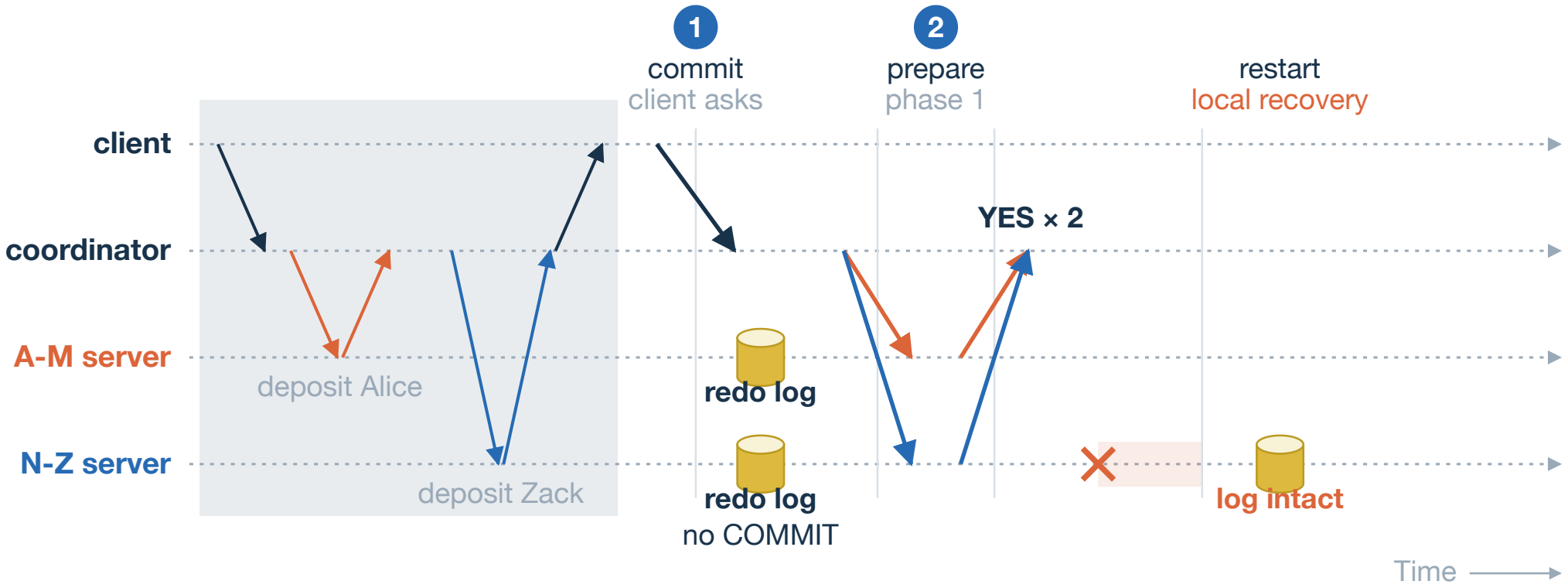
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



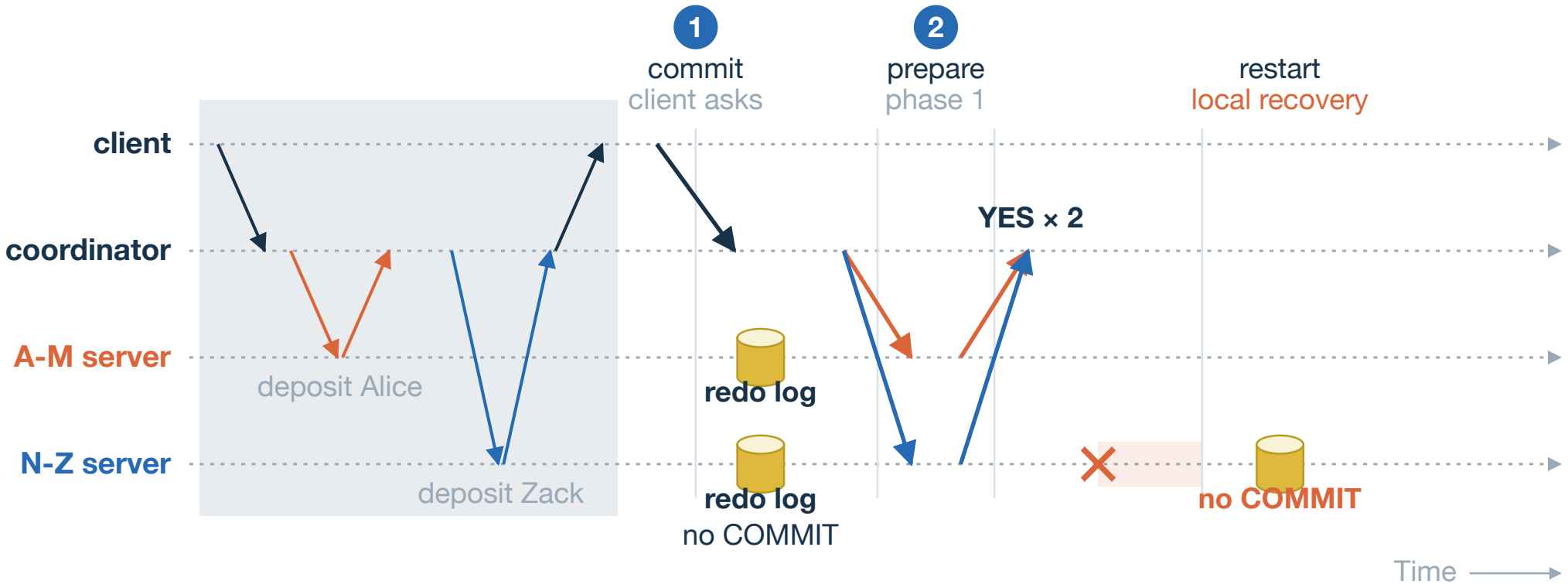
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



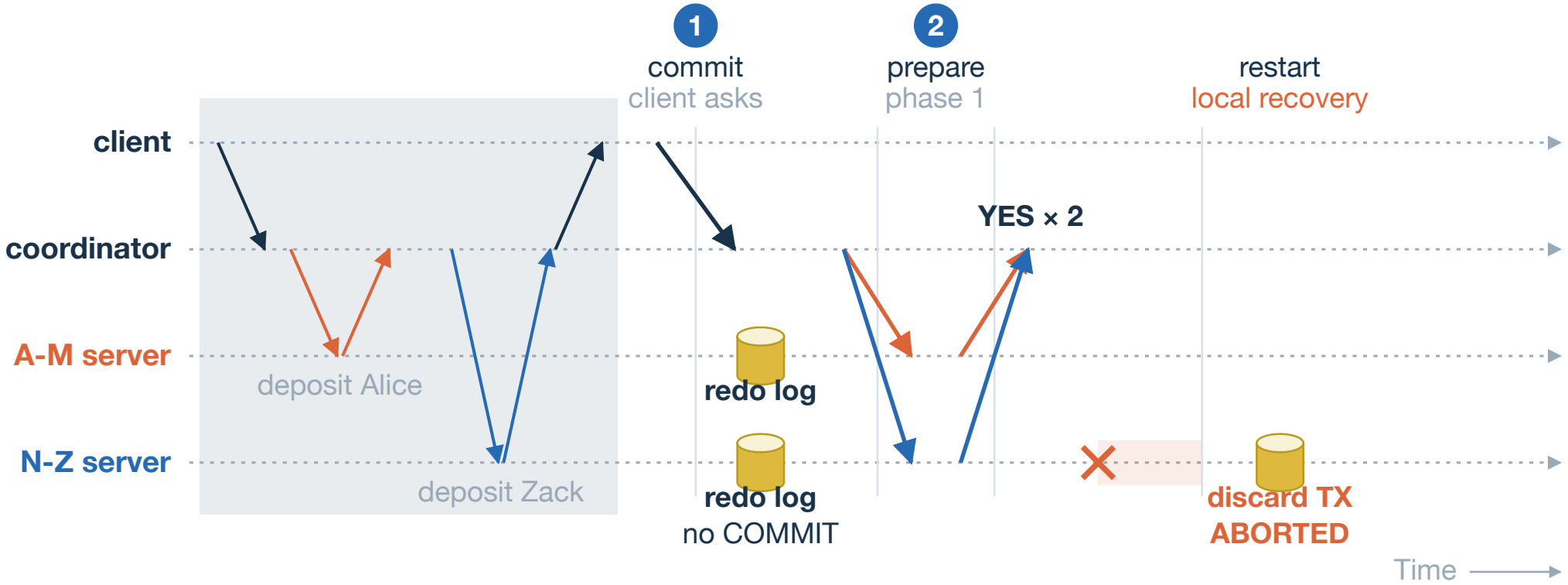
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



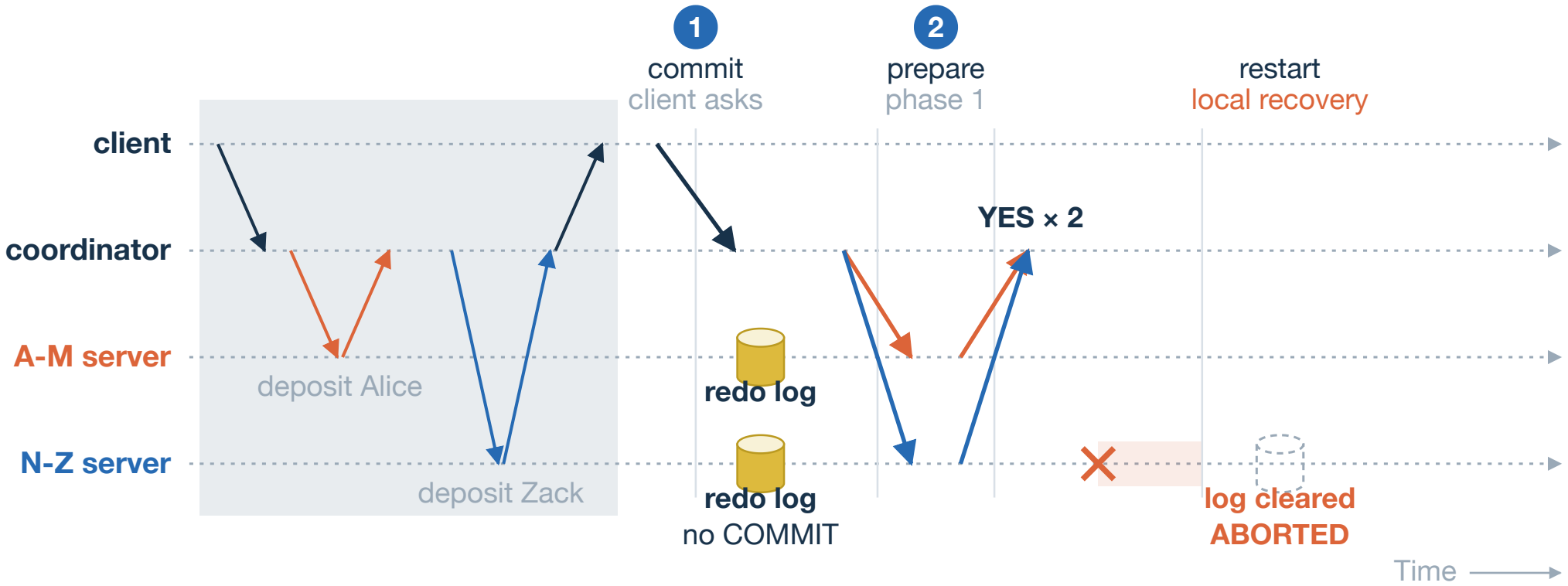
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



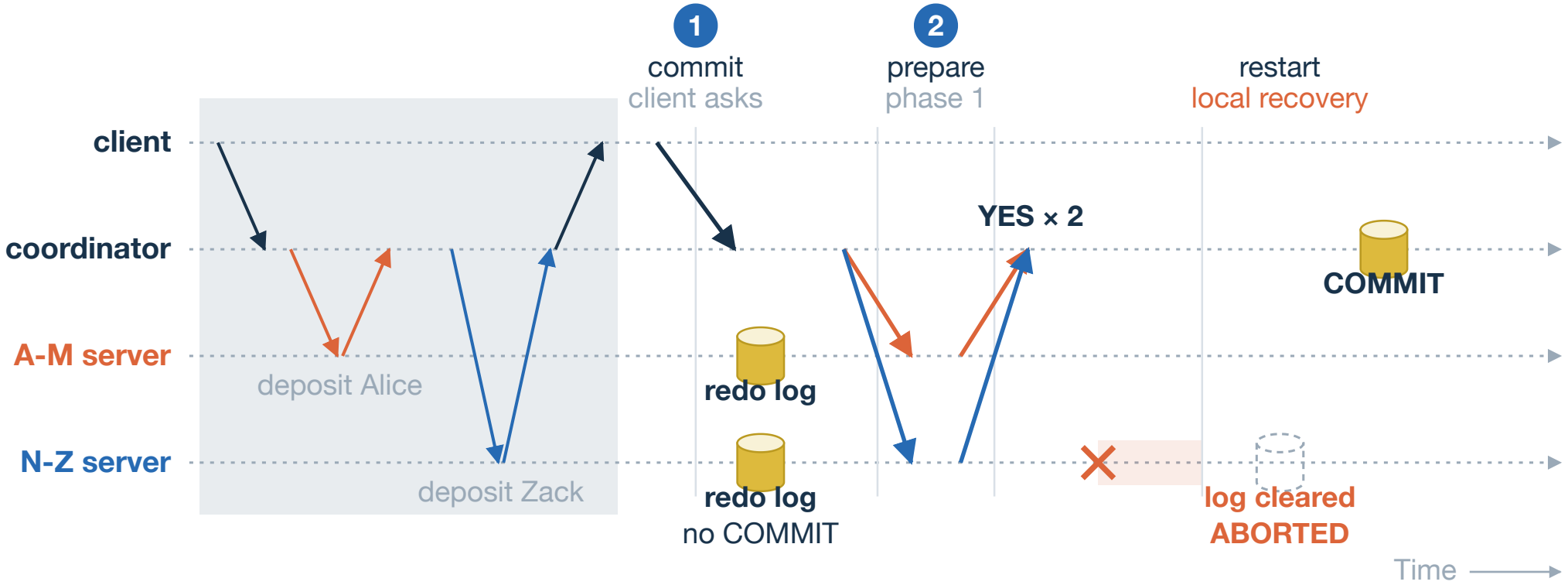
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



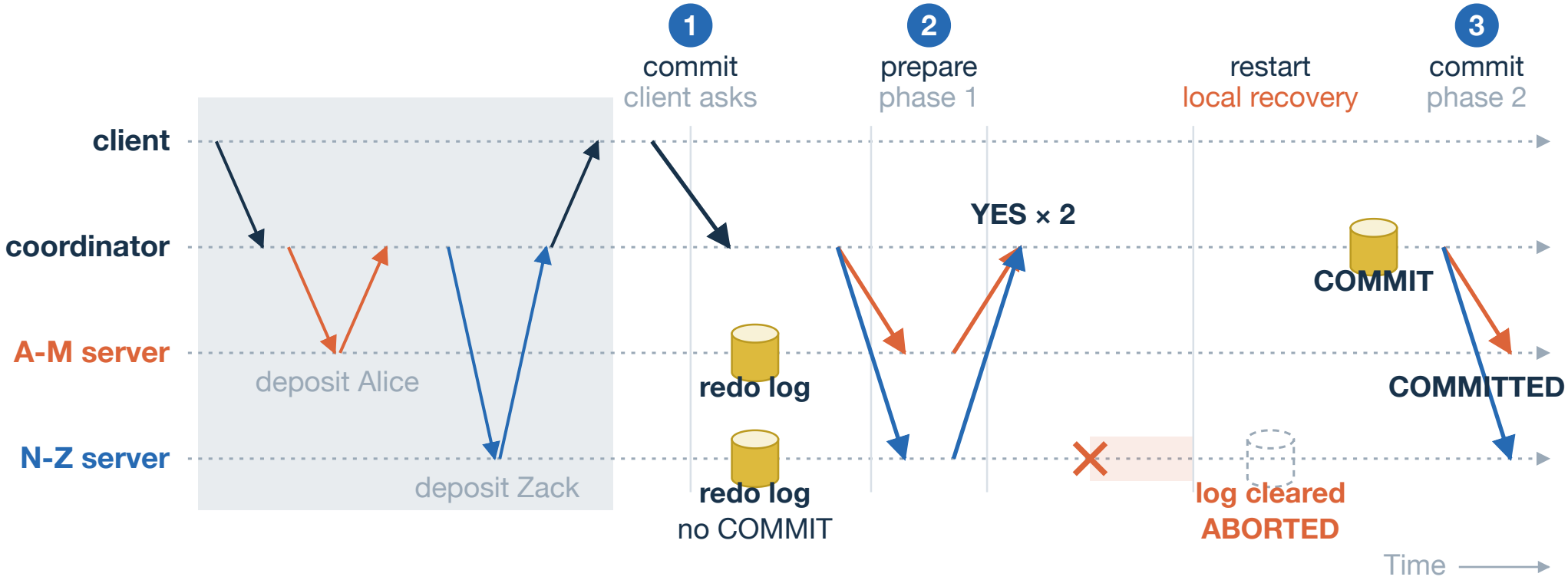
Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



Failure: worker failure during prepare

Counterexample: redo logging without PREPARED recovery



Coordinator: COMMIT; recovered worker: ABORTED

Question: how does a worker recover after a failure?

Can a worker individually decide to commit?

Multi-site atomicity

Principles:

- Following the coordinator's decision (one node makes the decision)
- Log sufficient state to tolerate failures (e.g., the coordinator's decision), because single-site log is insufficient

Coordinator (makes the decision & maintains the state)

- Collect an **ABORT**, or nothing before the timeout: **ABORT** or retry
- Collect all **COMMIT**: then **COMMIT**

Workers passively react to the coordinator's actions

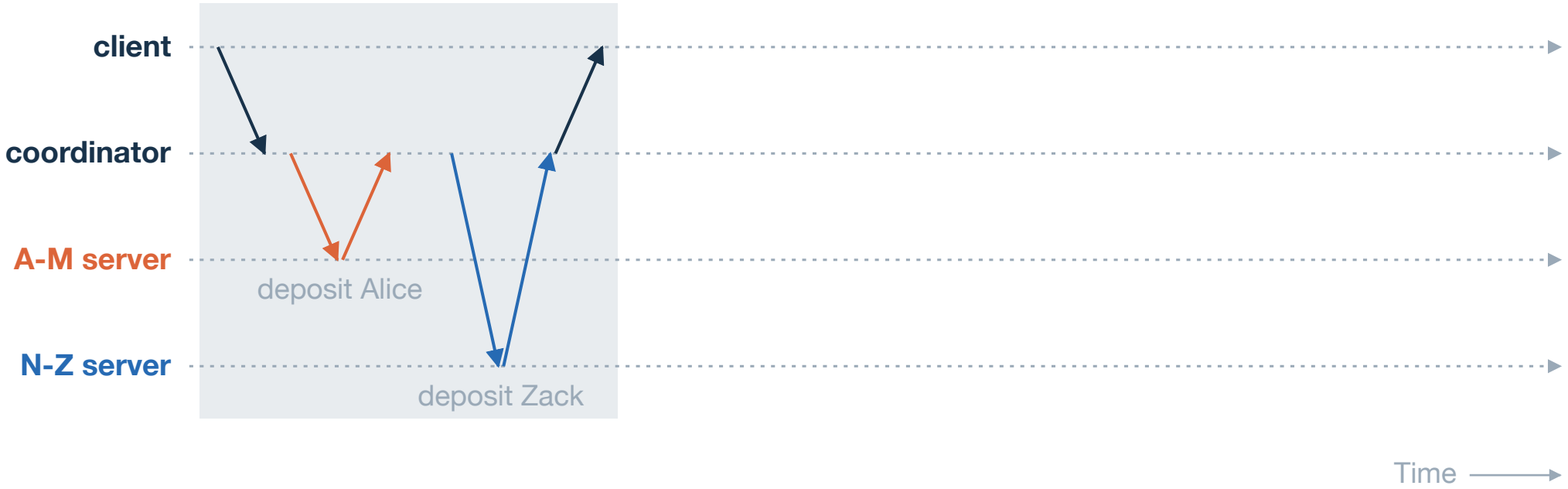
- If no message is sent from the coordinator, just wait
- When receive **COMMIT**: then **COMMIT**

Commit point: the coordinator made a decision + log the decision

What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

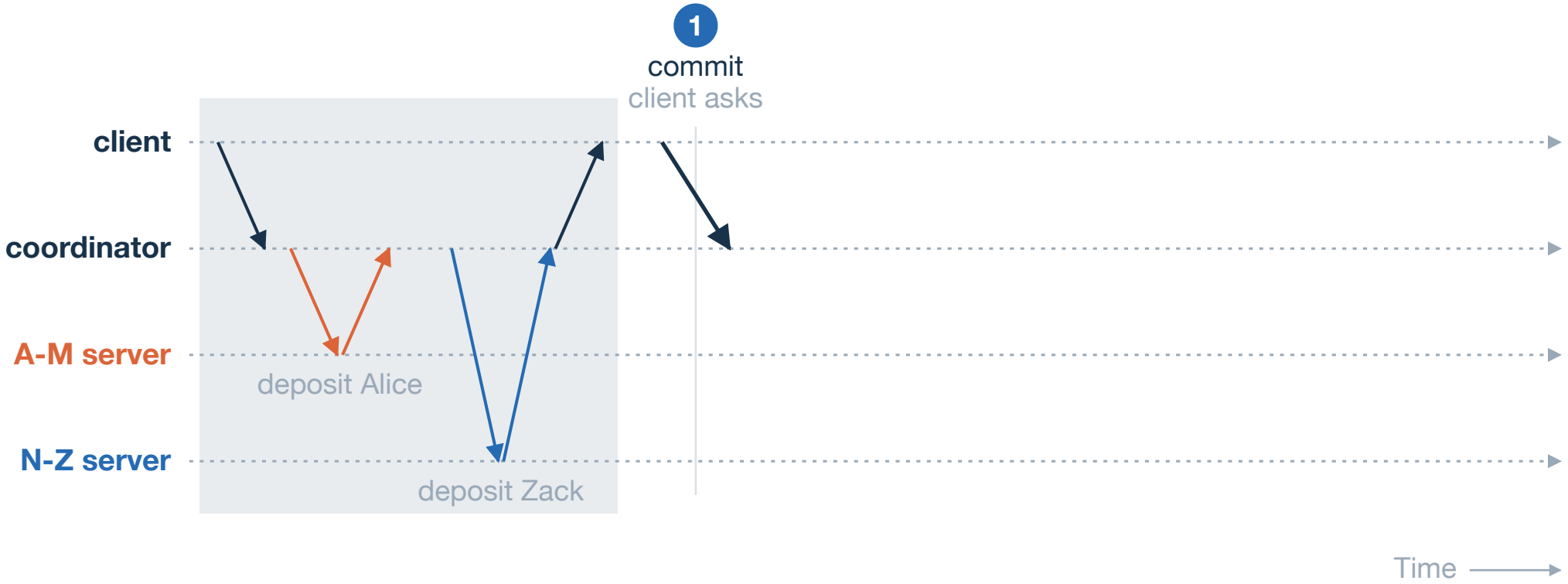
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

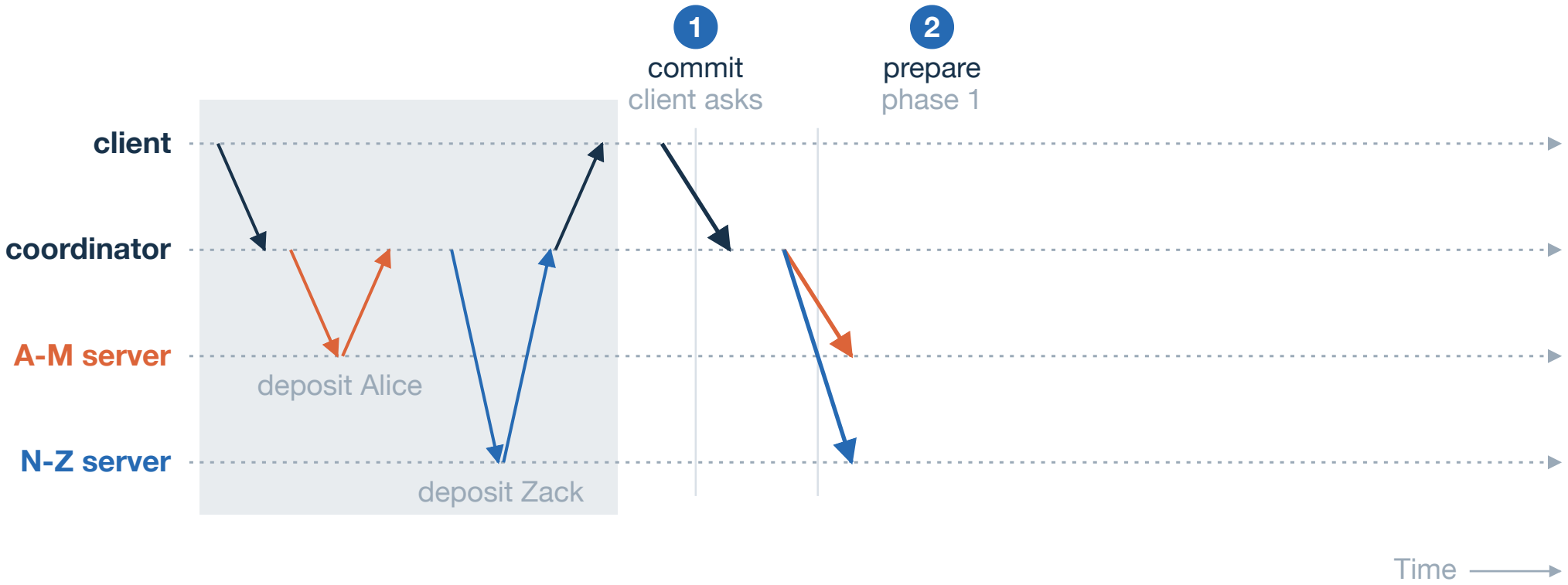
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

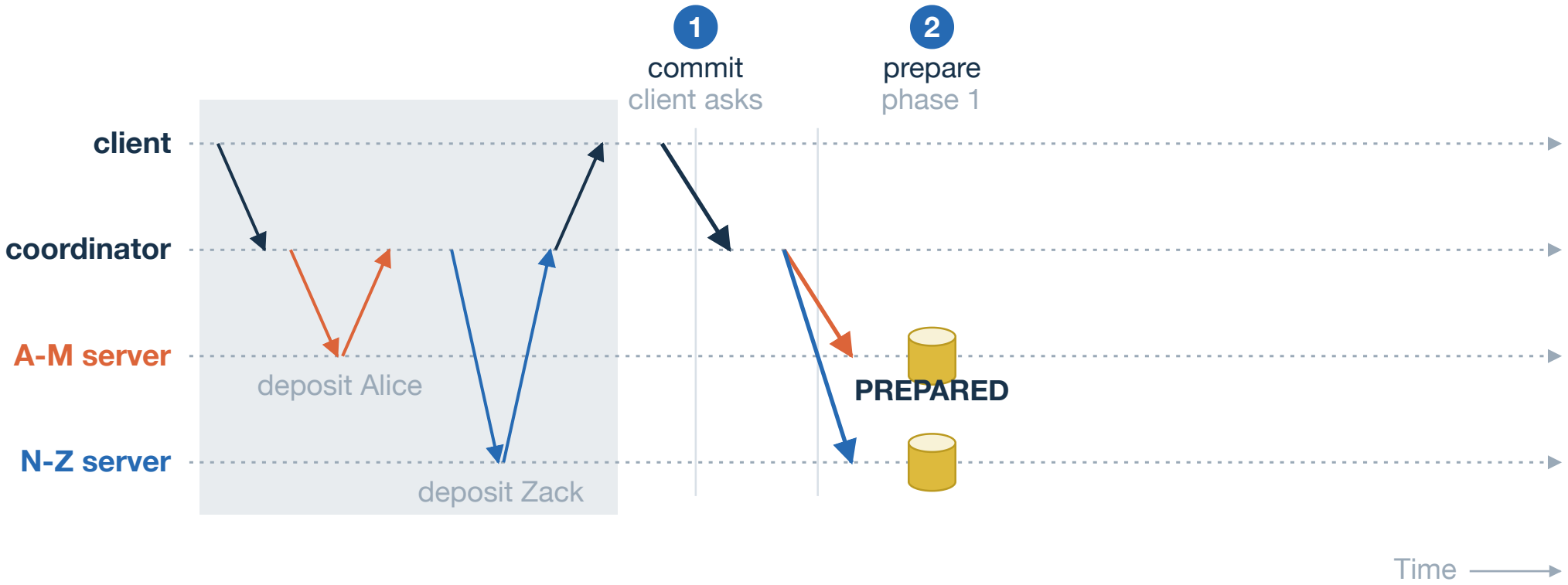
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

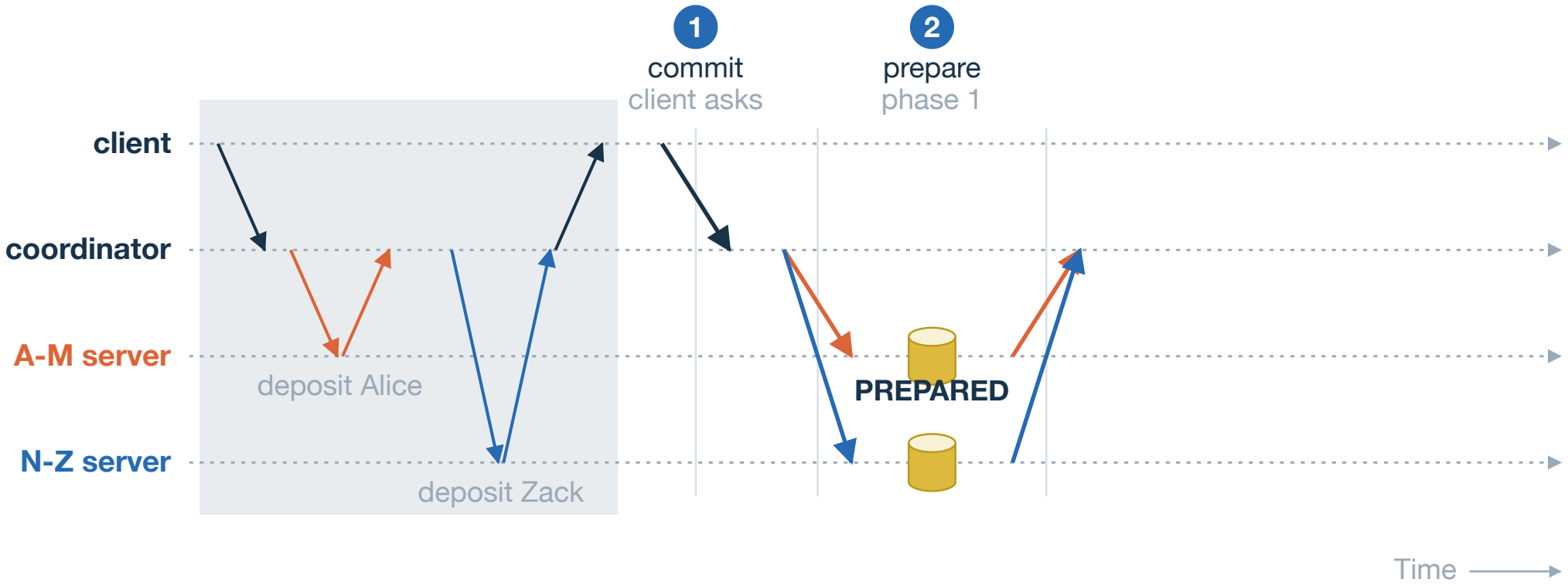
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

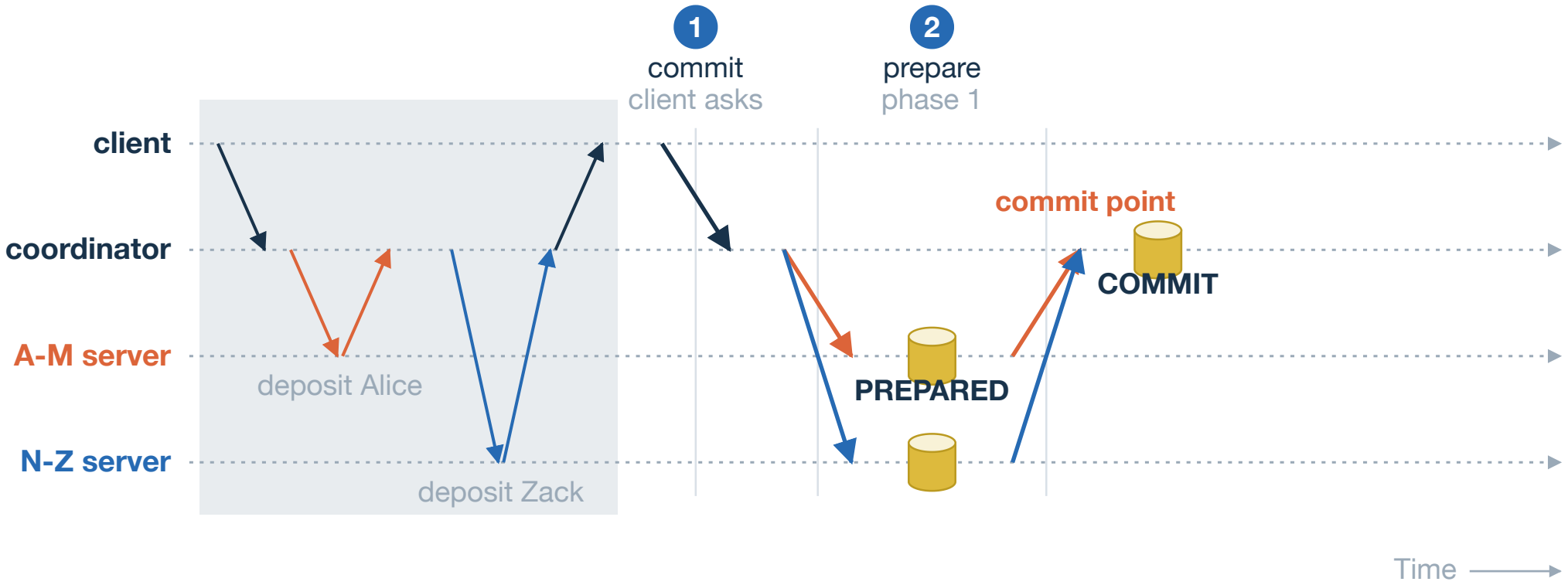
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

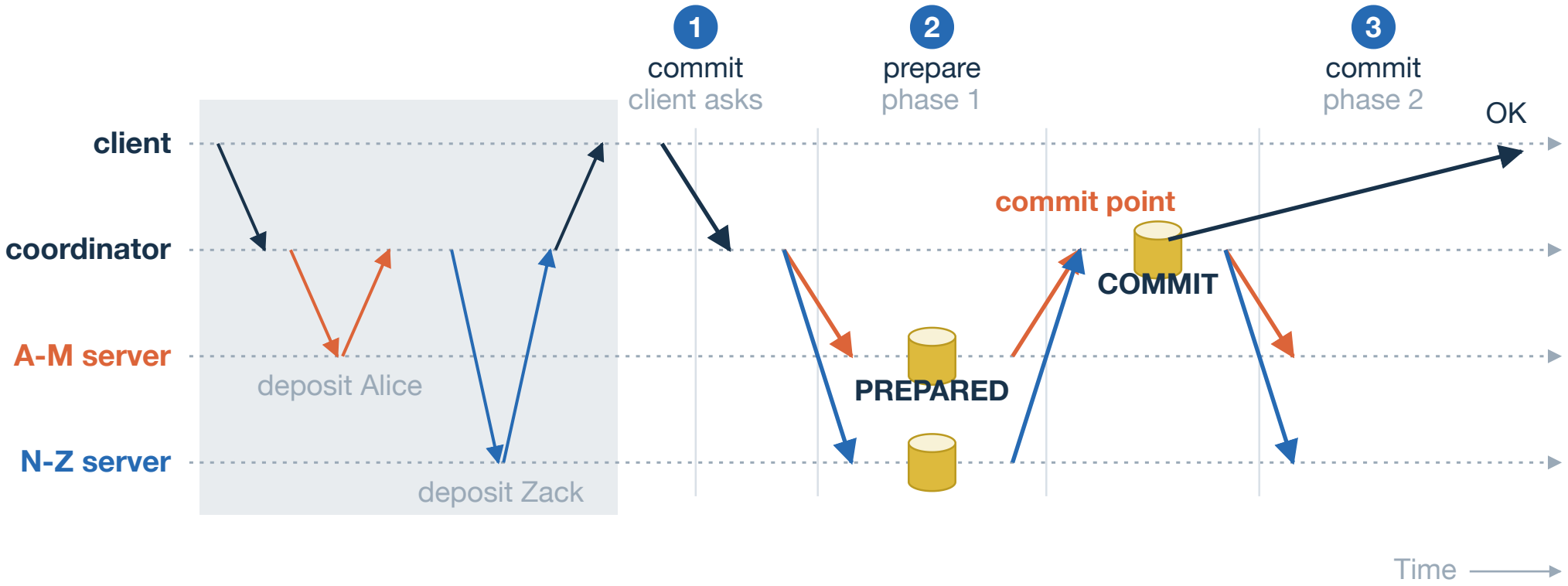
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

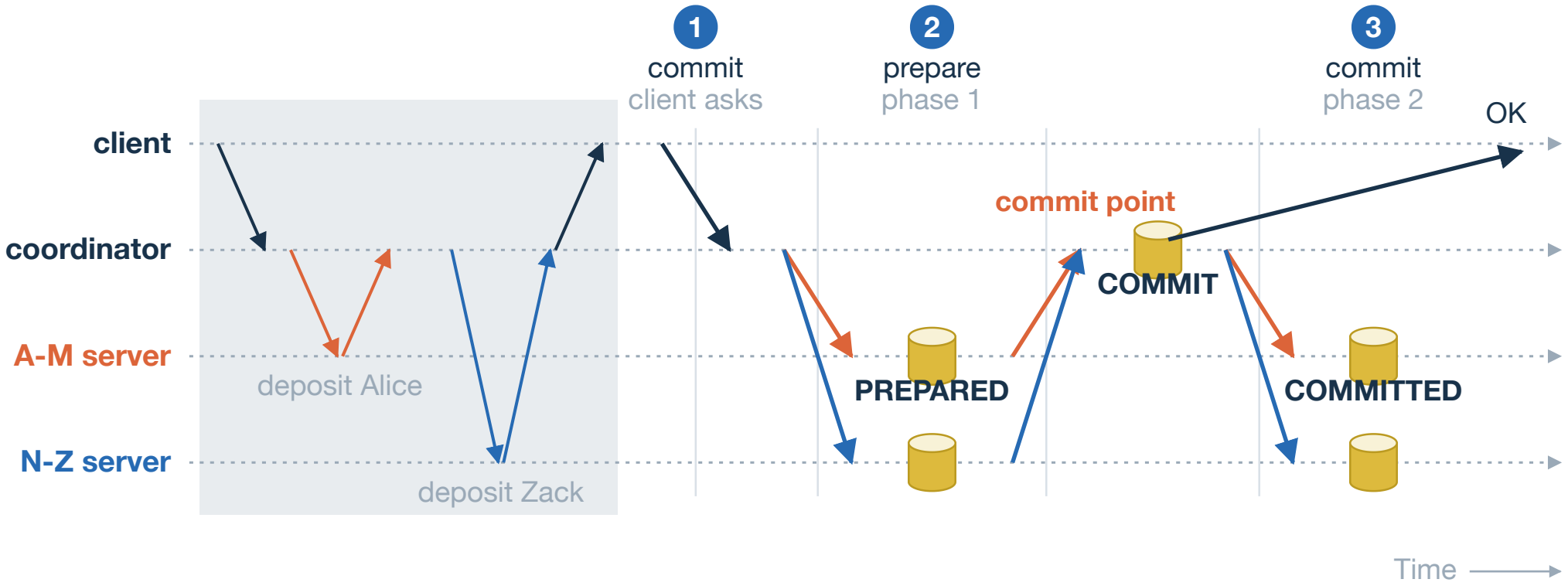
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

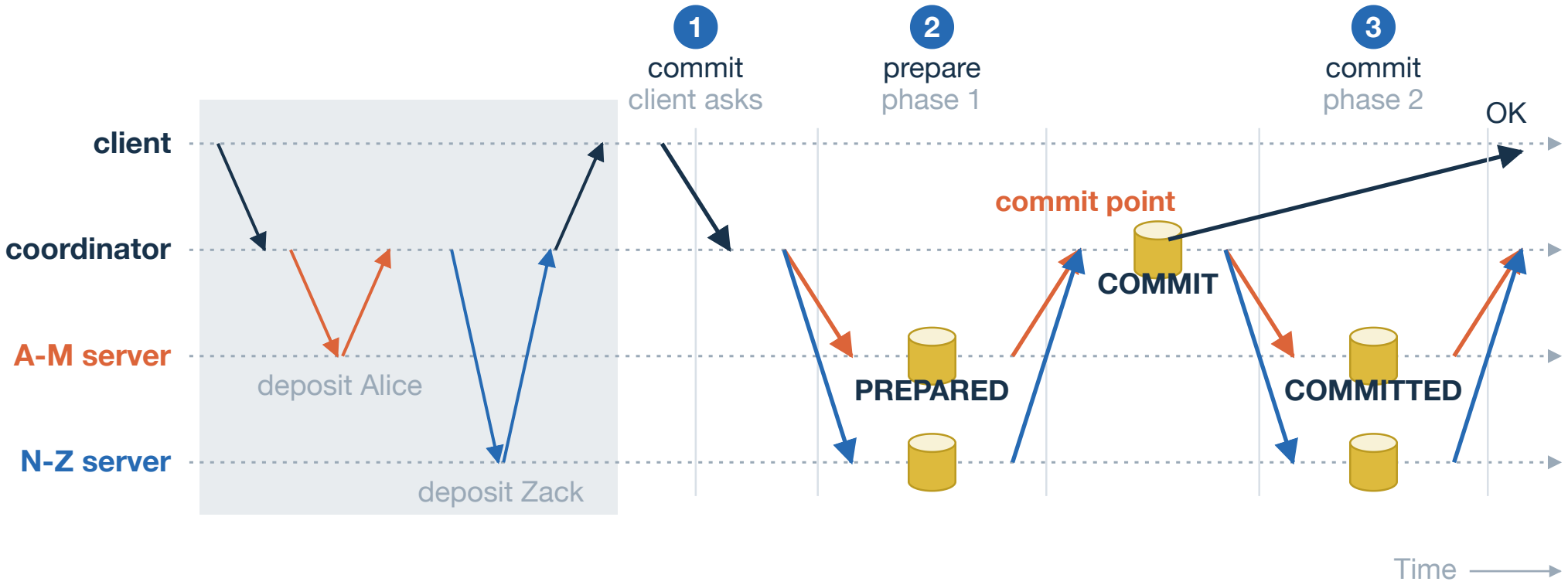
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



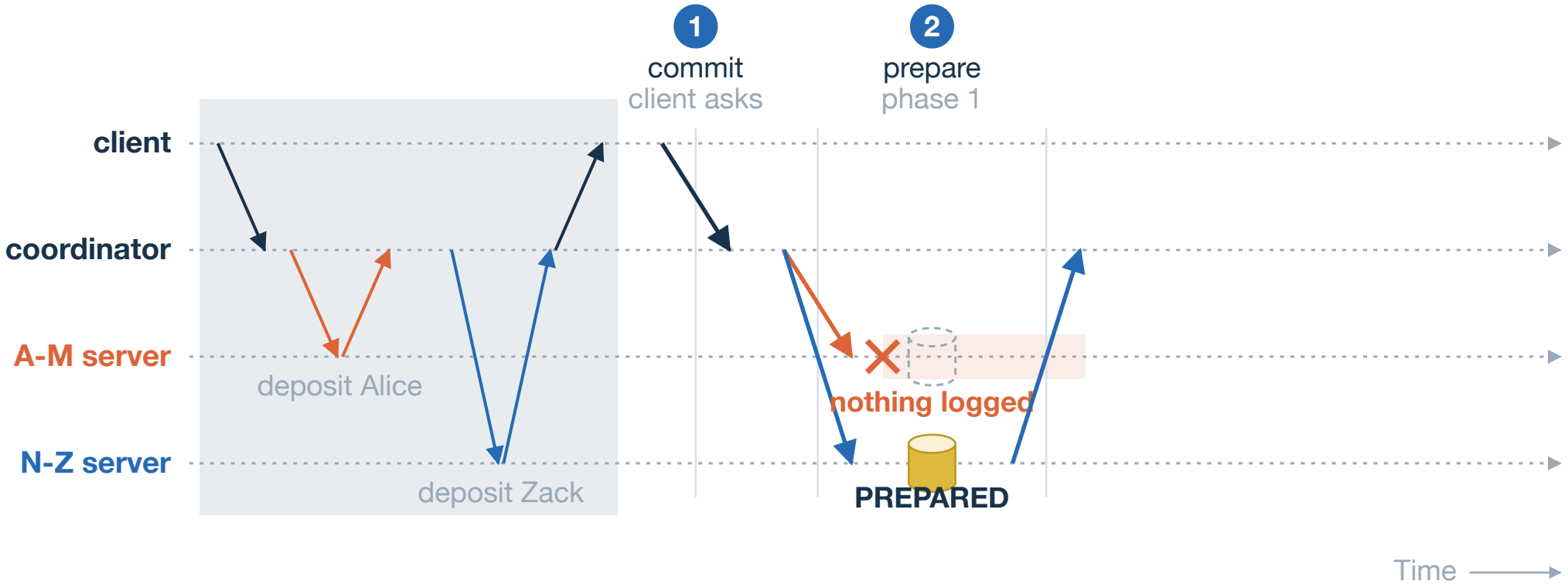
What two-phase commit adds to the log

Each site logs to its own disk before it sends the message that depends on it

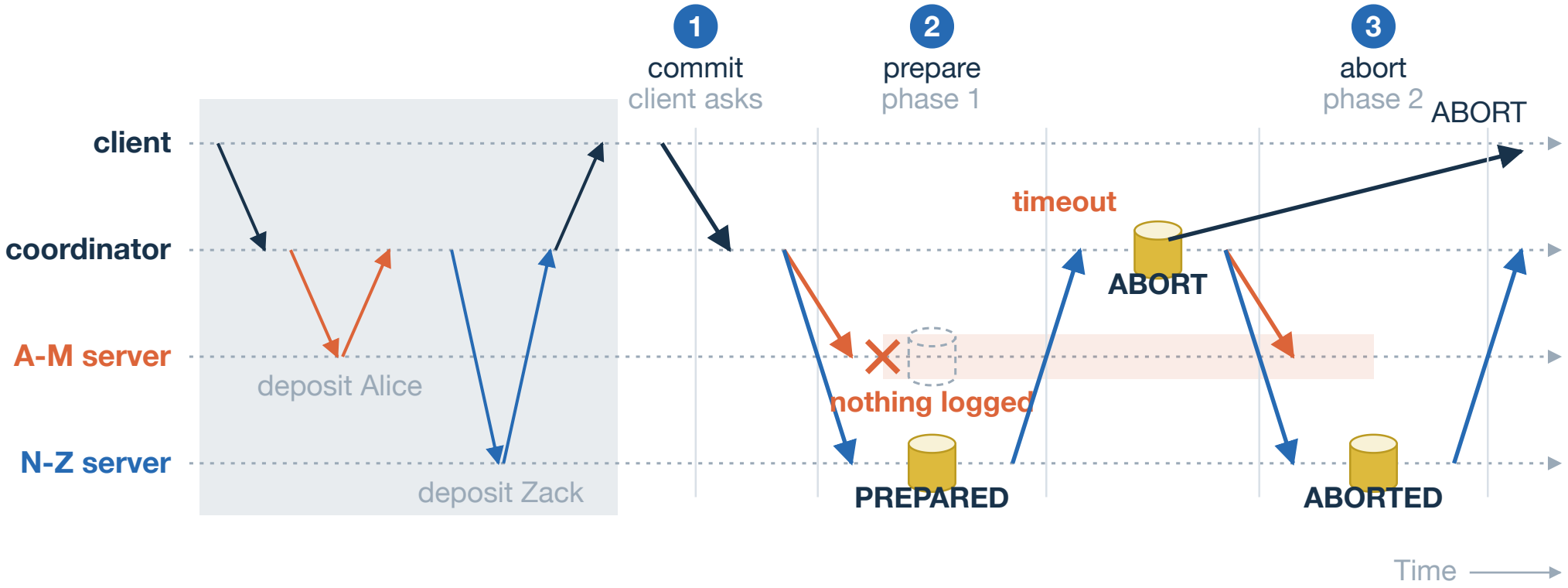
- Worker: **PREPARED** plus its updates, before voting yes — a reboot cannot take it back
- Coordinator: the decision, before sending it out — that write **is** the commit point



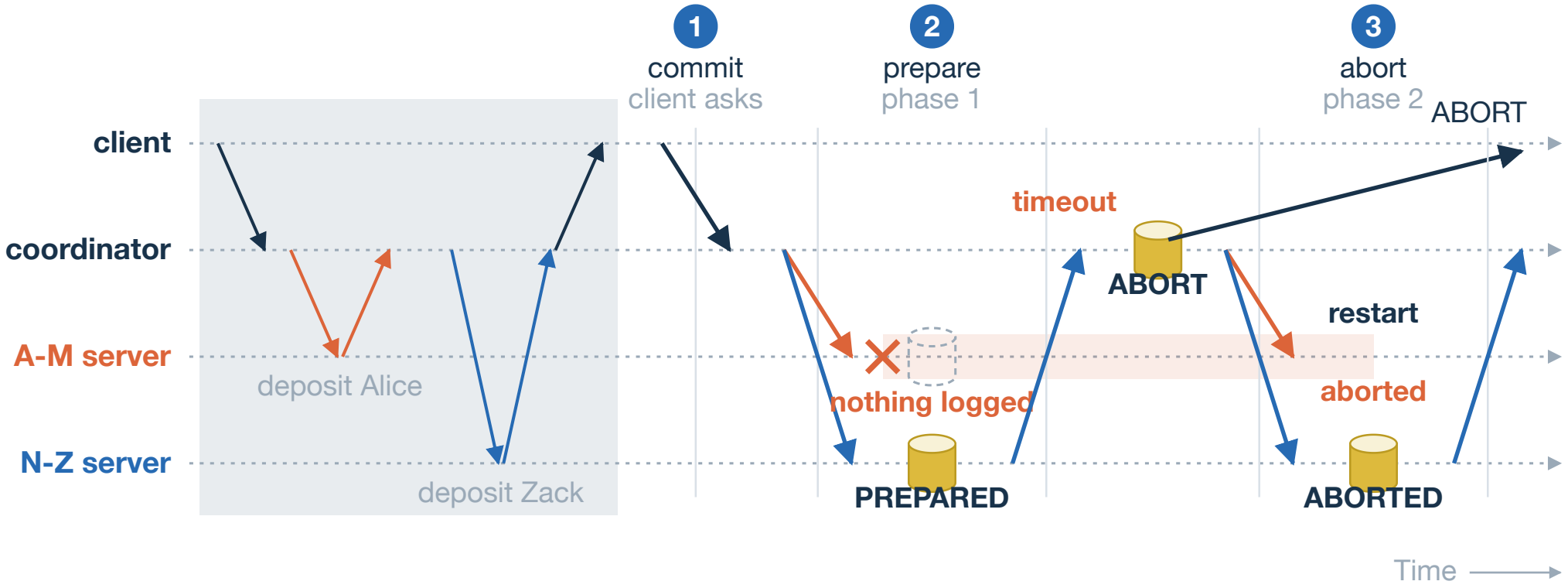
Missing a vote: abort preserves atomicity



Missing a vote: abort preserves atomicity

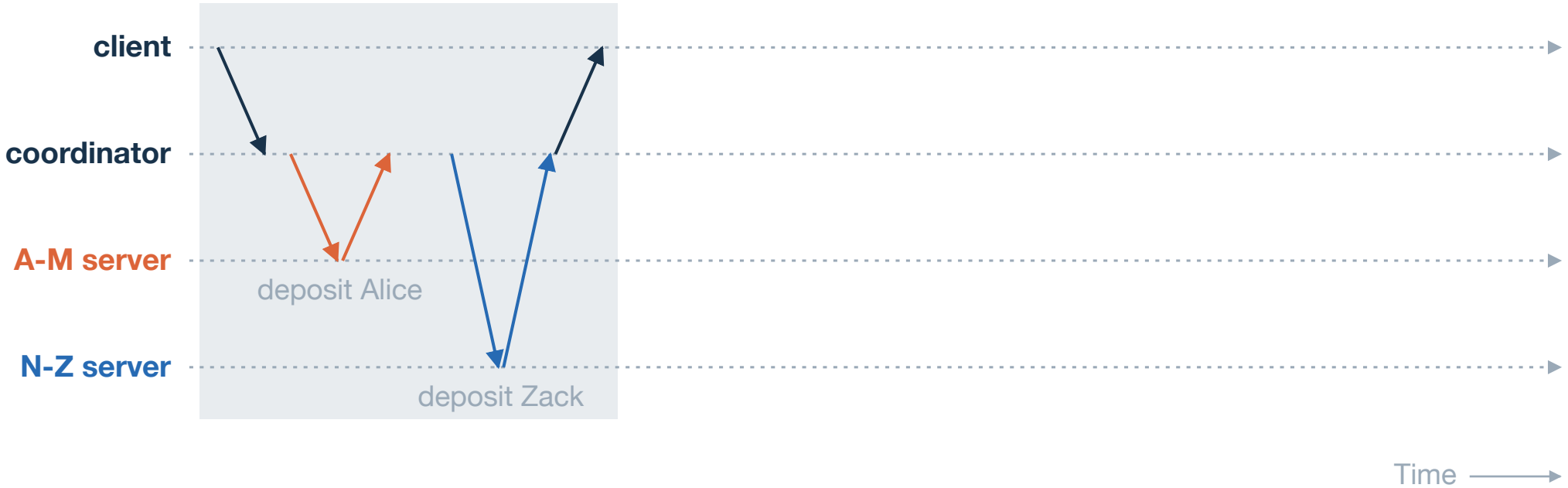


Missing a vote: abort preserves atomicity

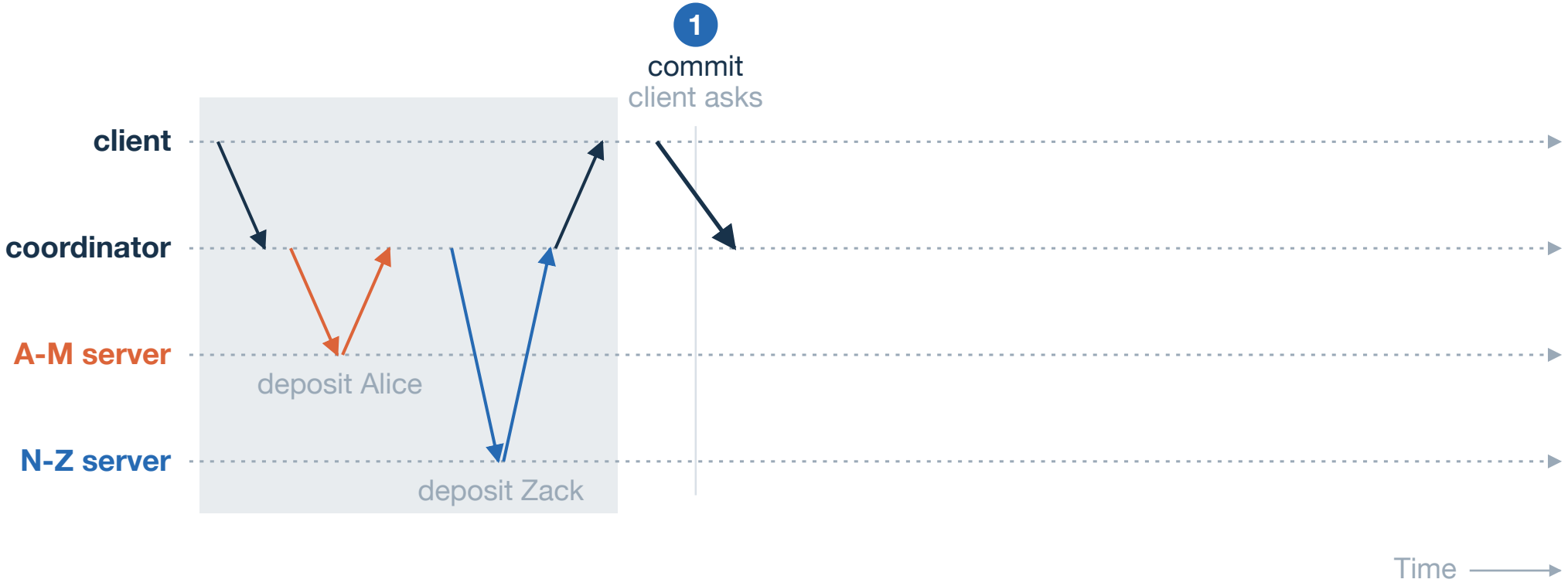


PREPARED is not COMMITTED:
both sites abort, preserving multi-site atomicity

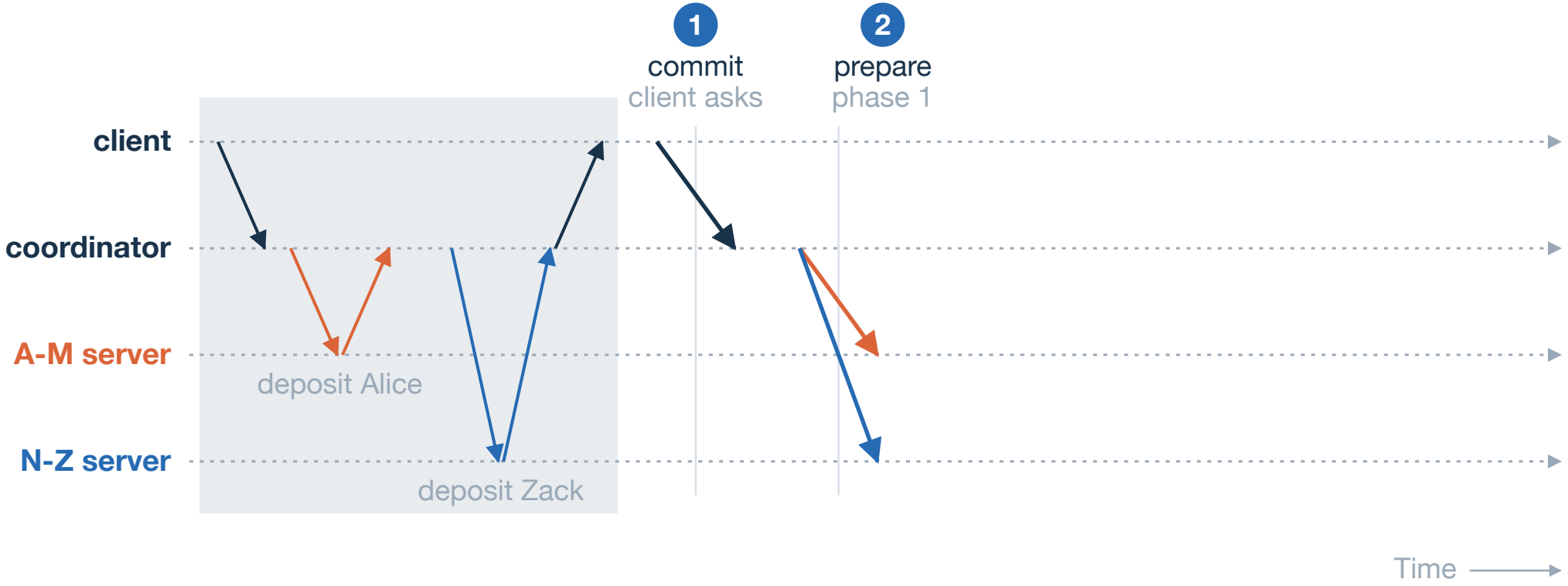
More failure cases: lost commit message



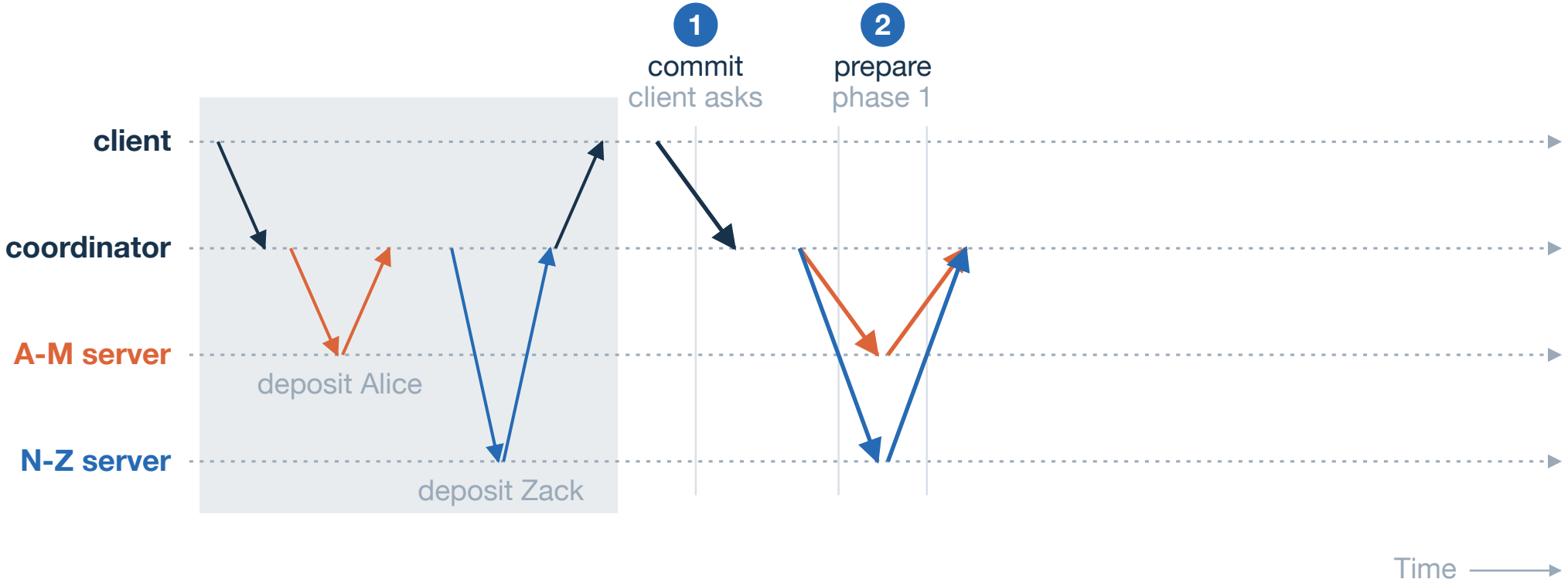
More failure cases: lost commit message



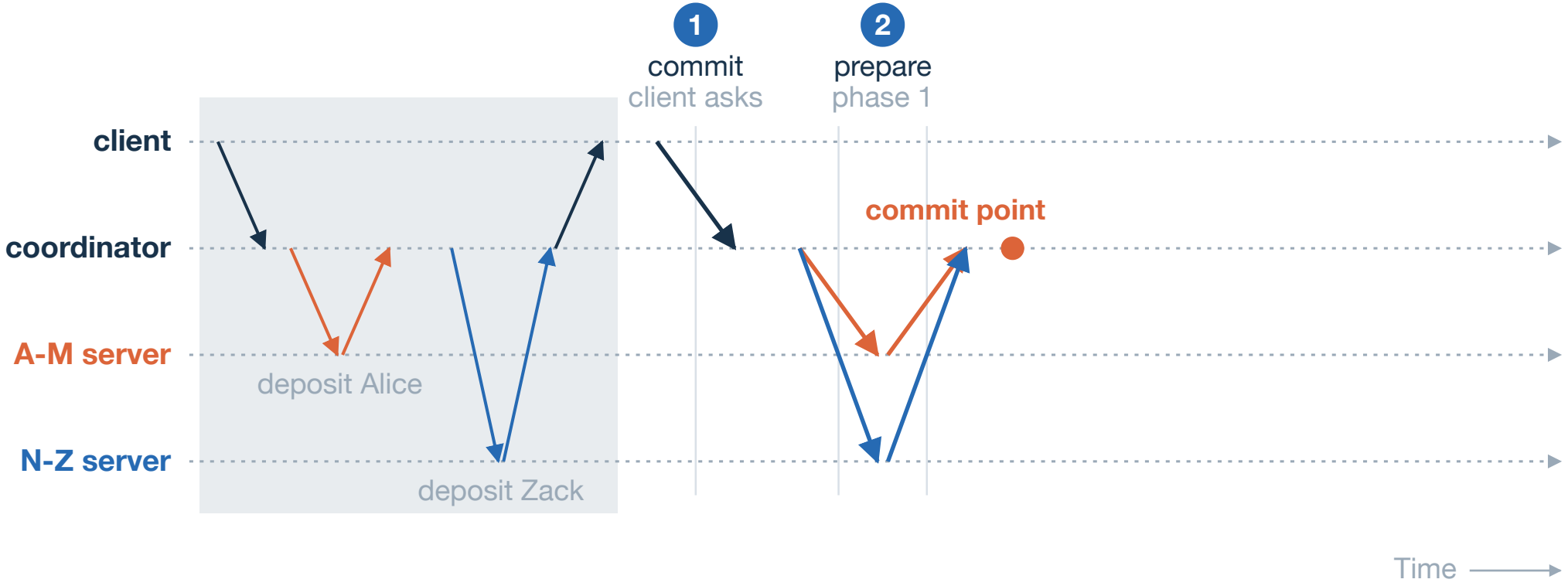
More failure cases: lost commit message



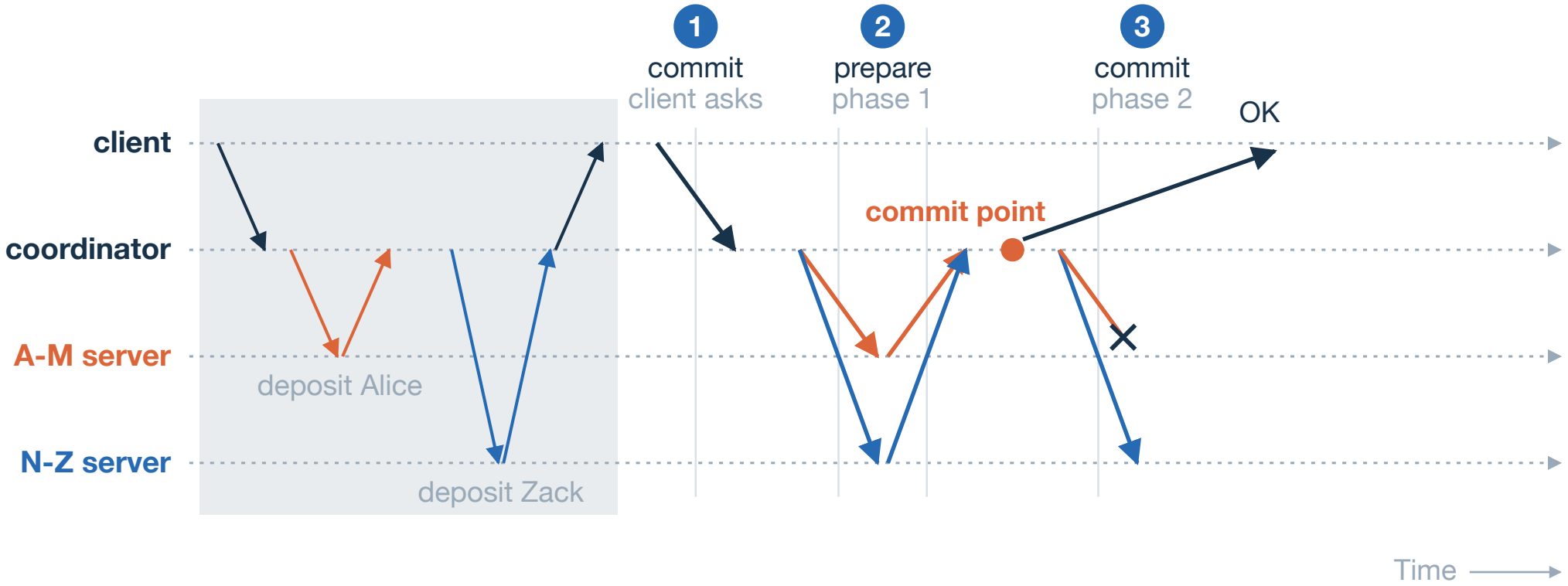
More failure cases: lost commit message



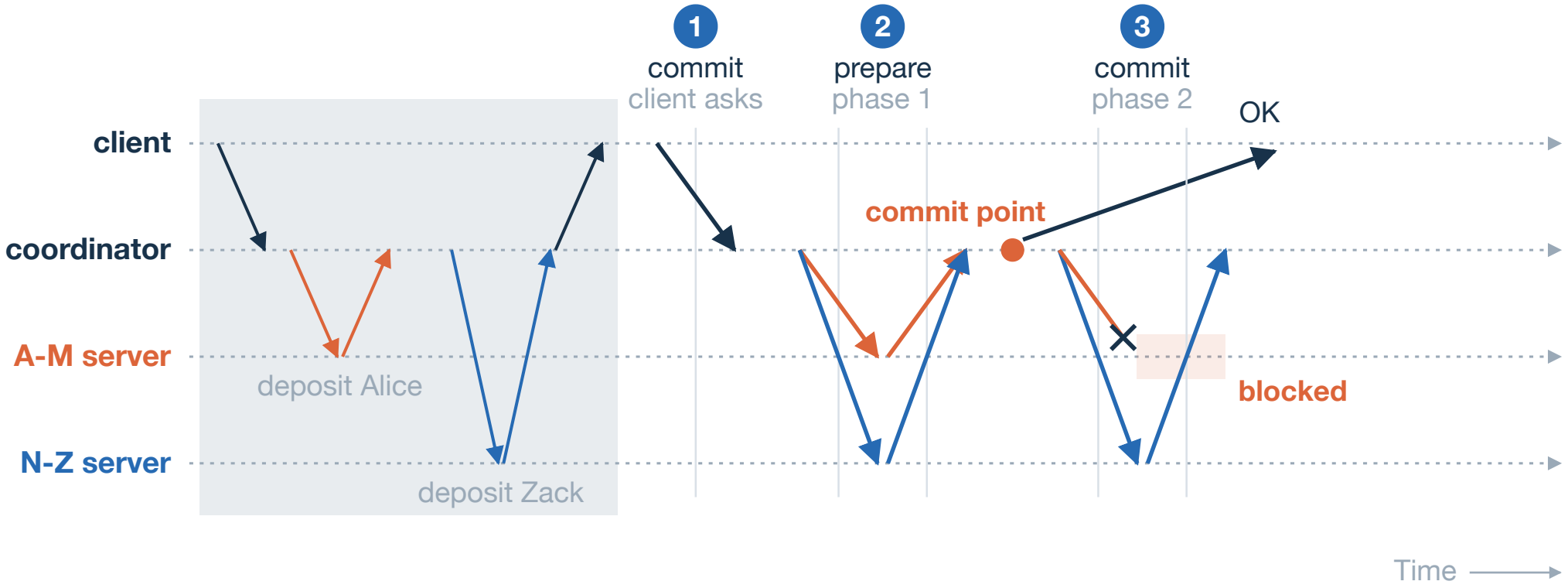
More failure cases: lost commit message



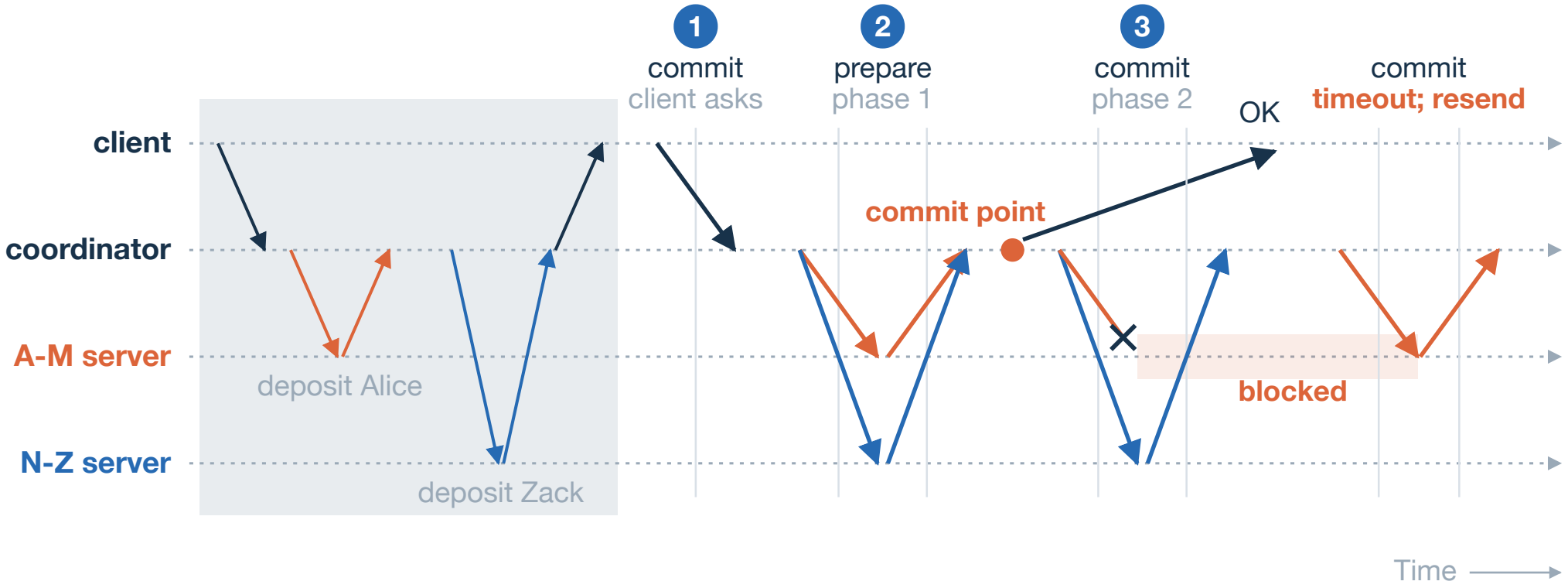
More failure cases: lost commit message



More failure cases: lost commit message

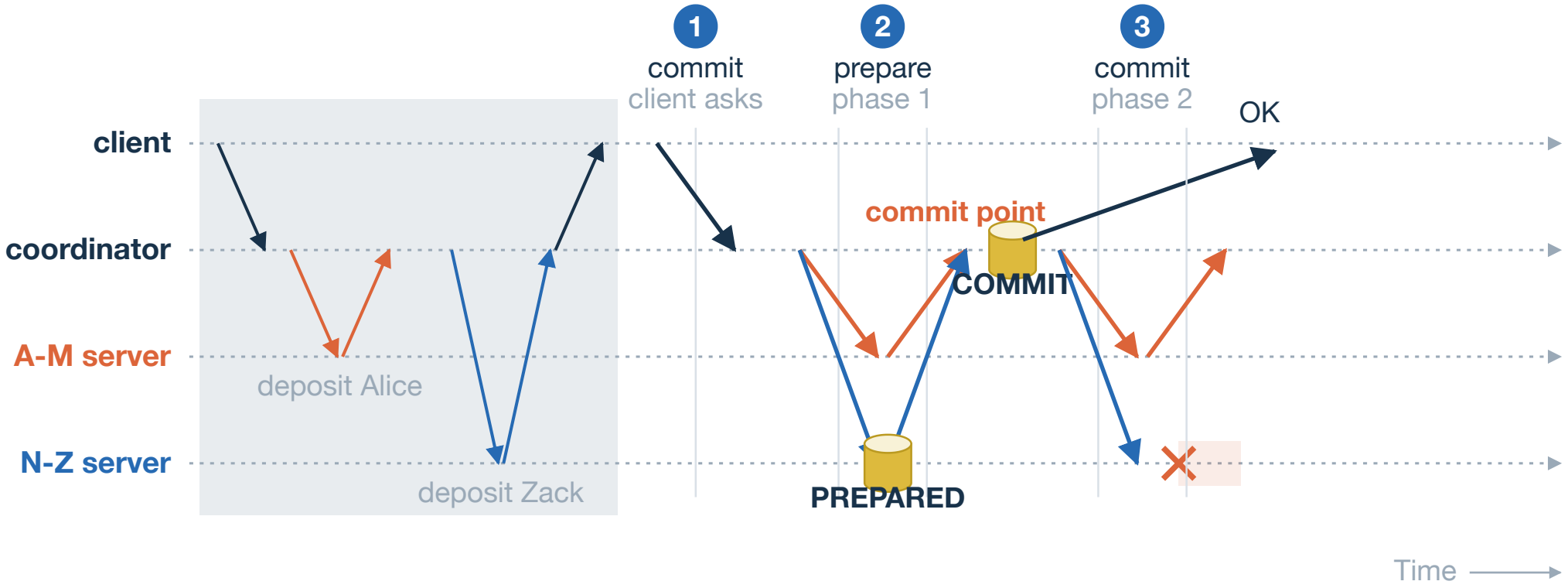


More failure cases: lost commit message

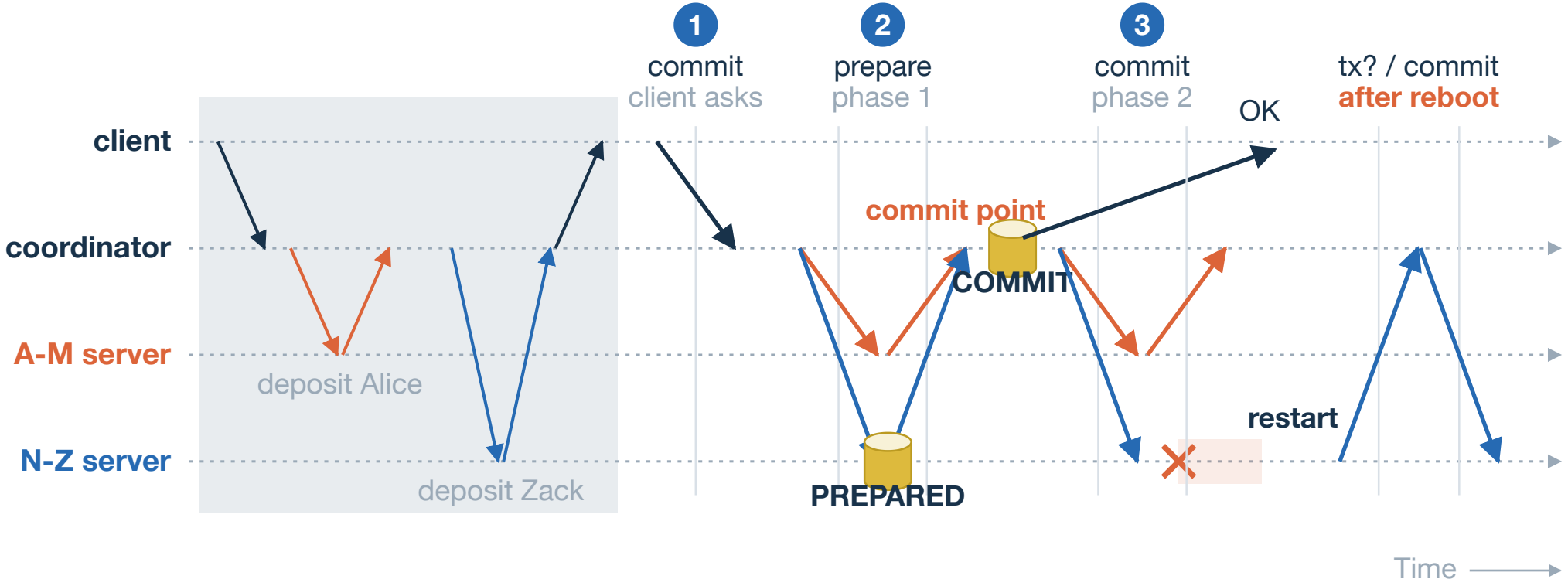


Handled: timeout & resend
but A-M is blocked until it hears

Failure: worker failure during commit

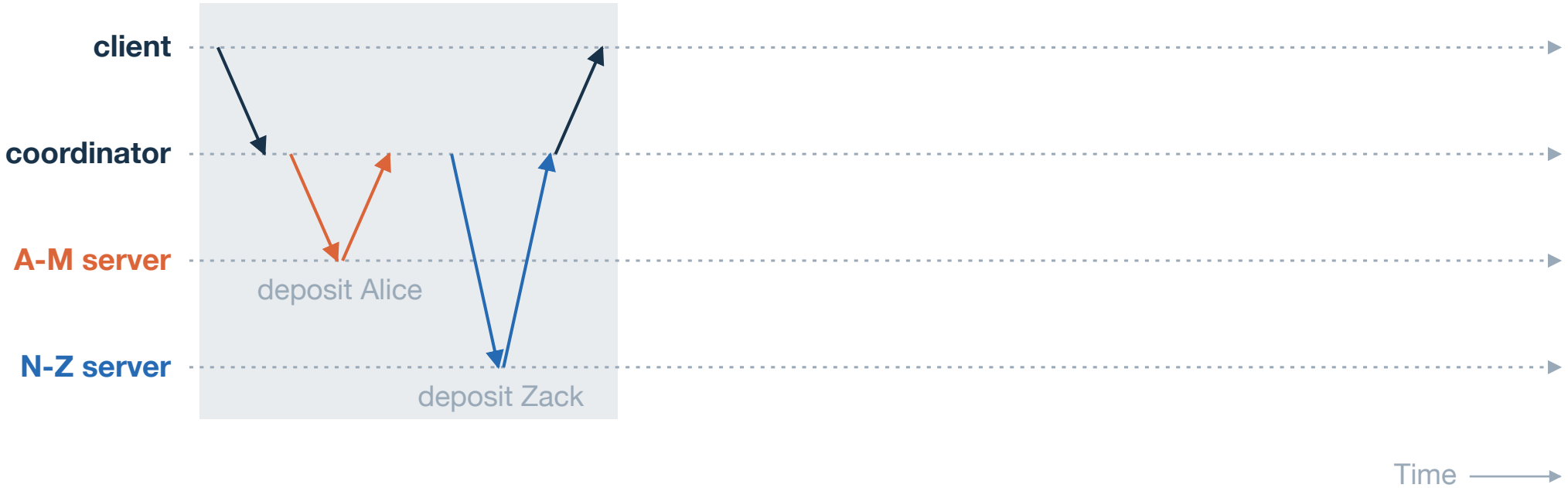


Failure: worker failure during commit

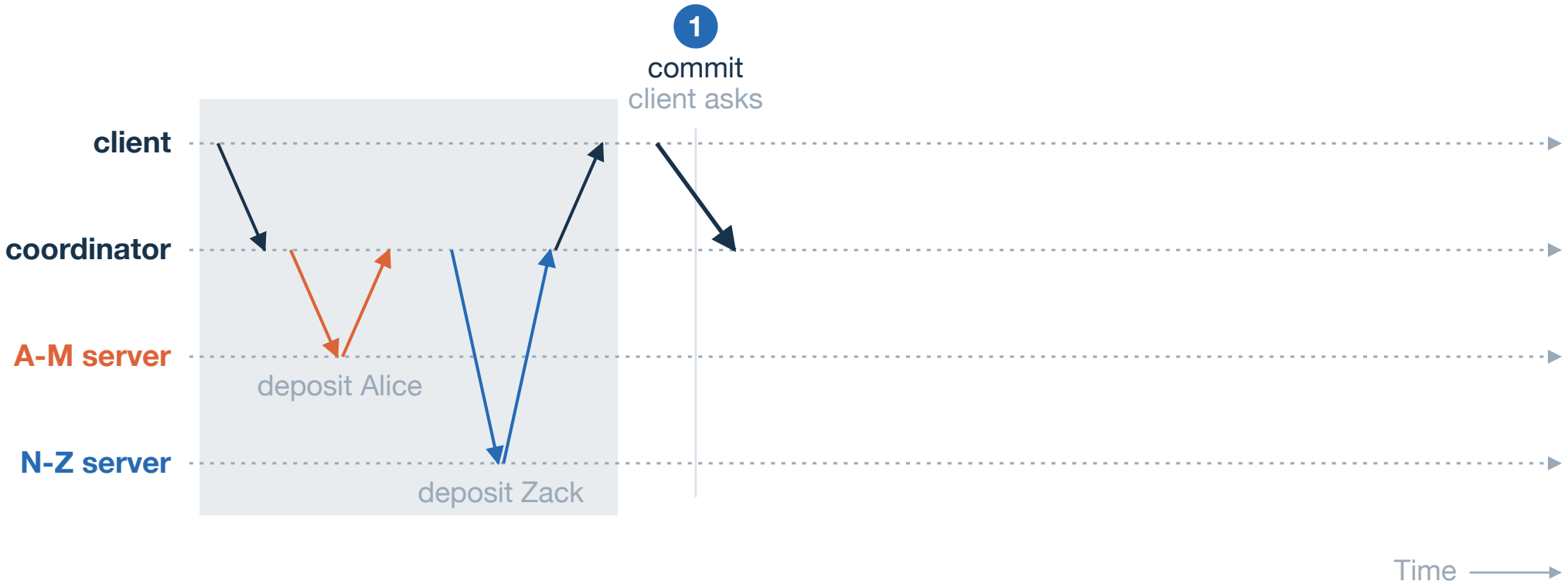


Recover into PREPARED, then ask the coordinator for the decision

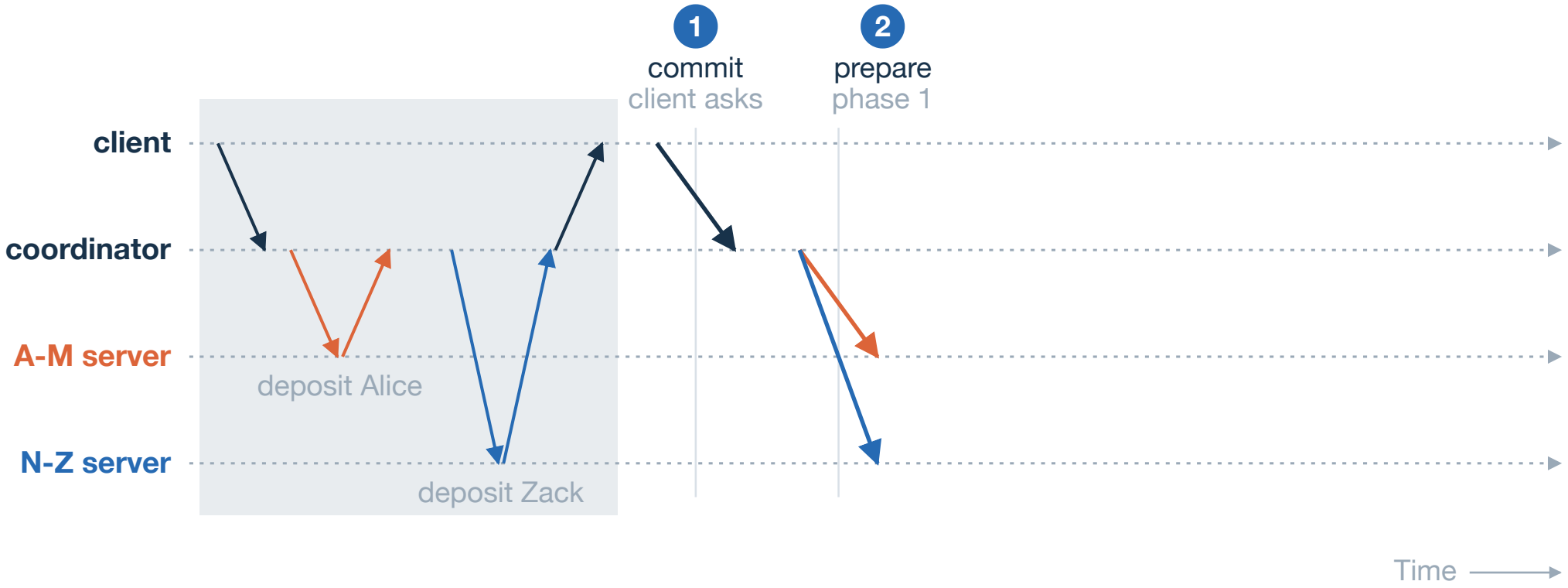
Failure: coordinator failure during prepare



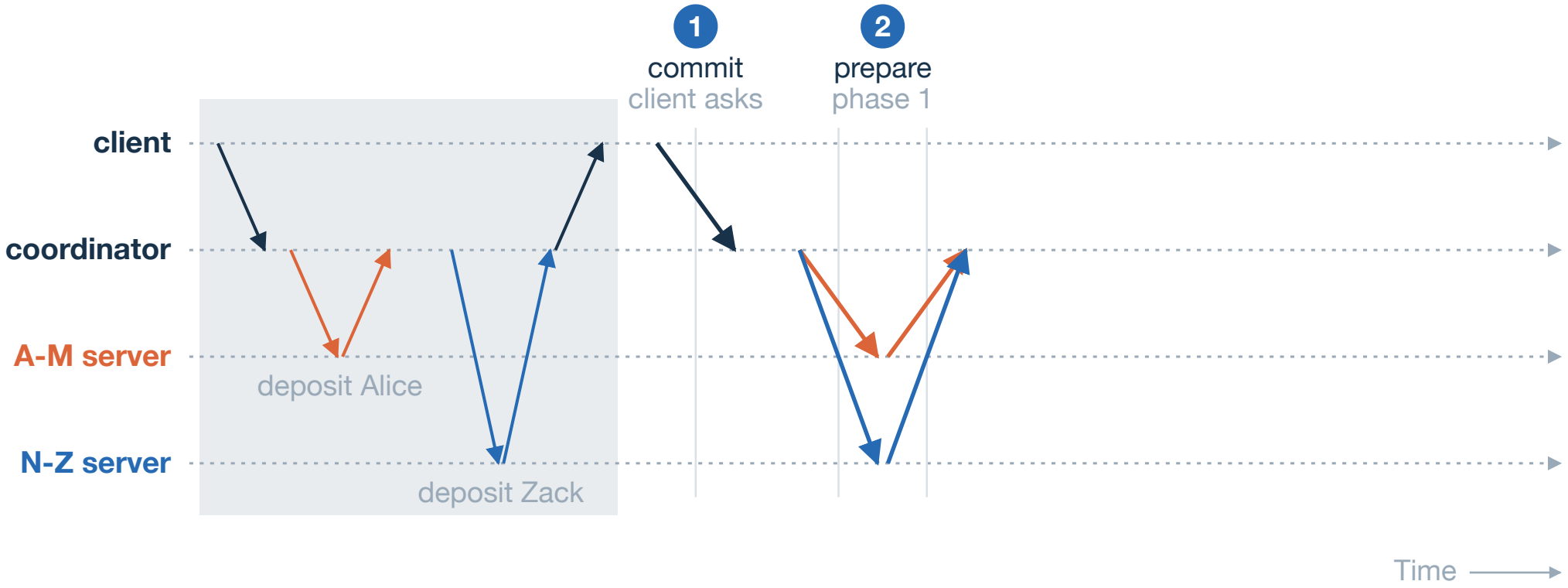
Failure: coordinator failure during prepare



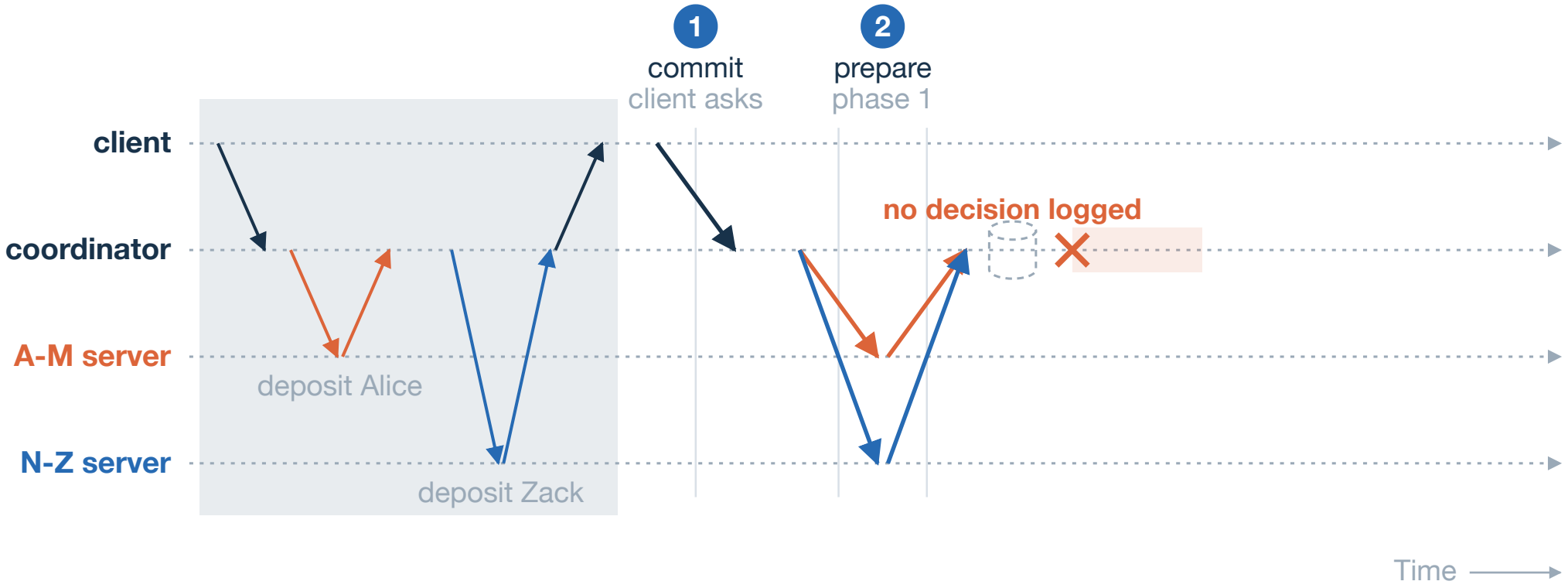
Failure: coordinator failure during prepare



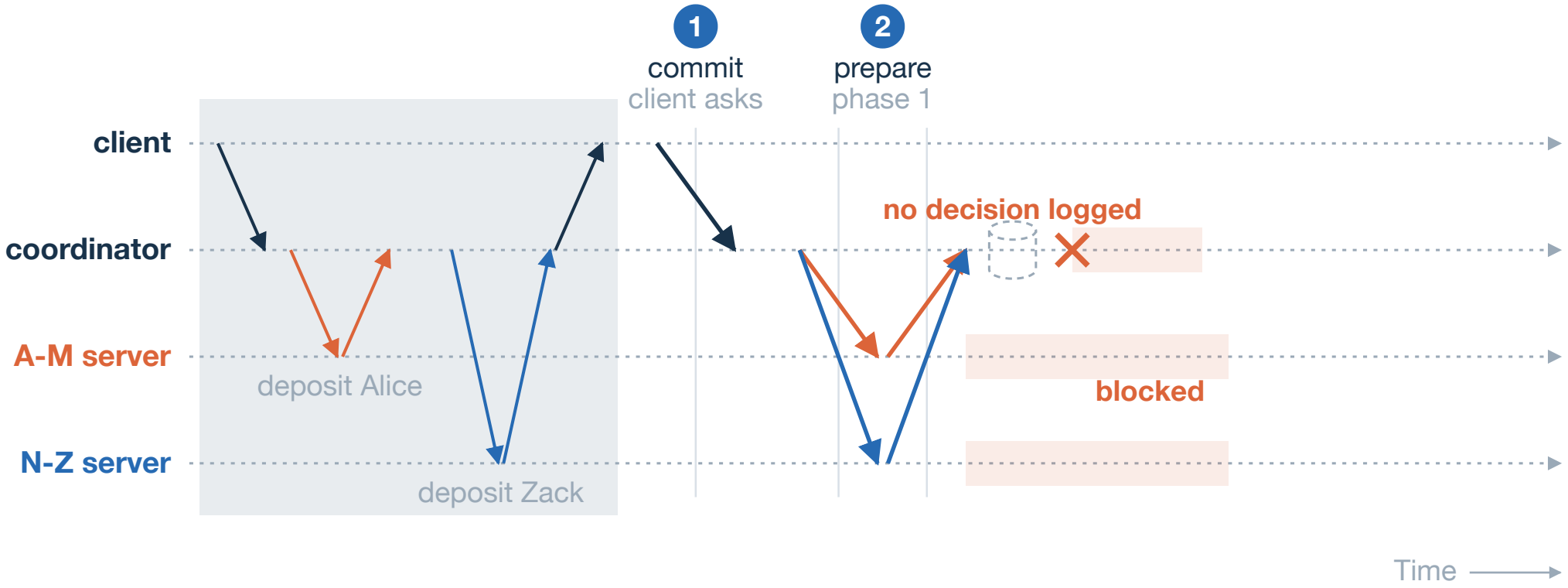
Failure: coordinator failure during prepare



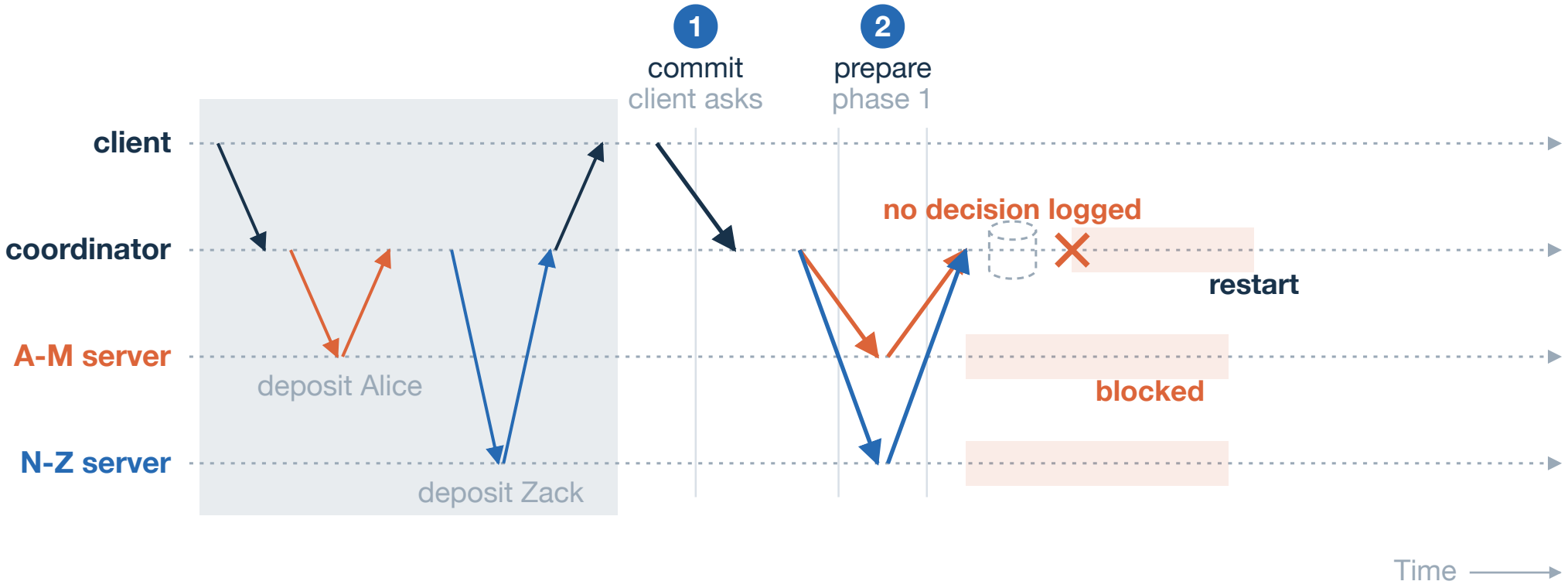
Failure: coordinator failure during prepare



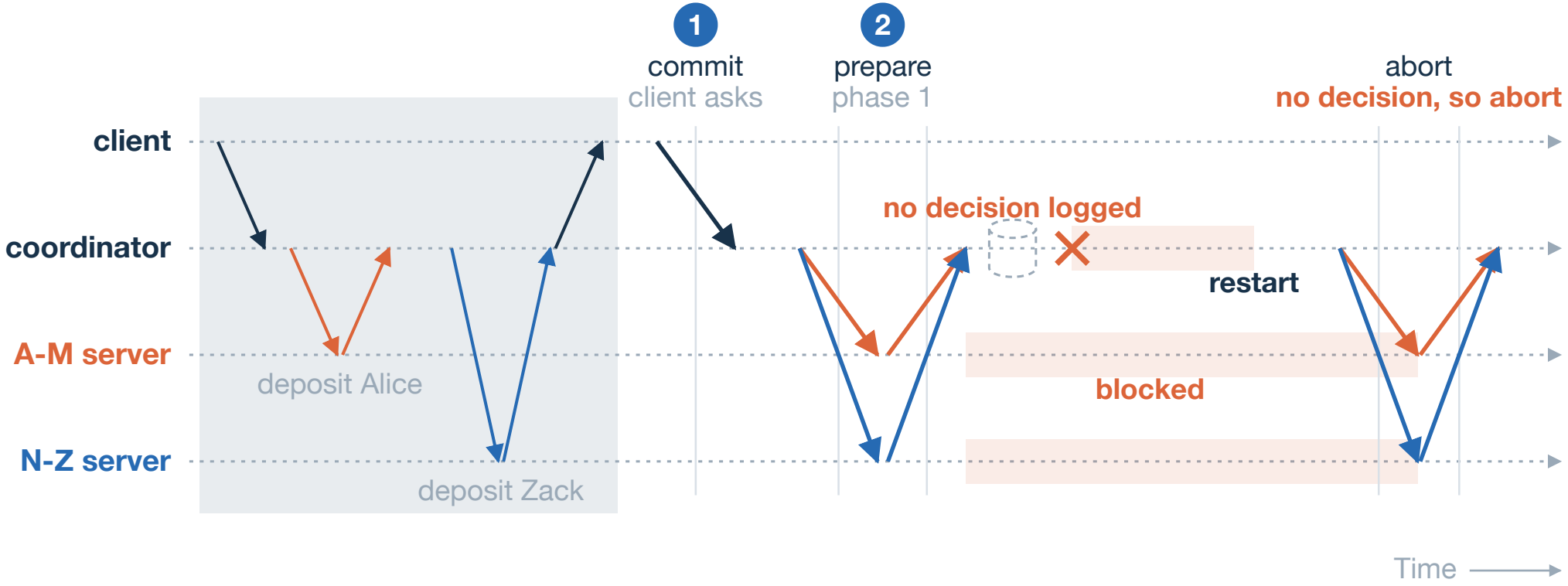
Failure: coordinator failure during prepare



Failure: coordinator failure during prepare



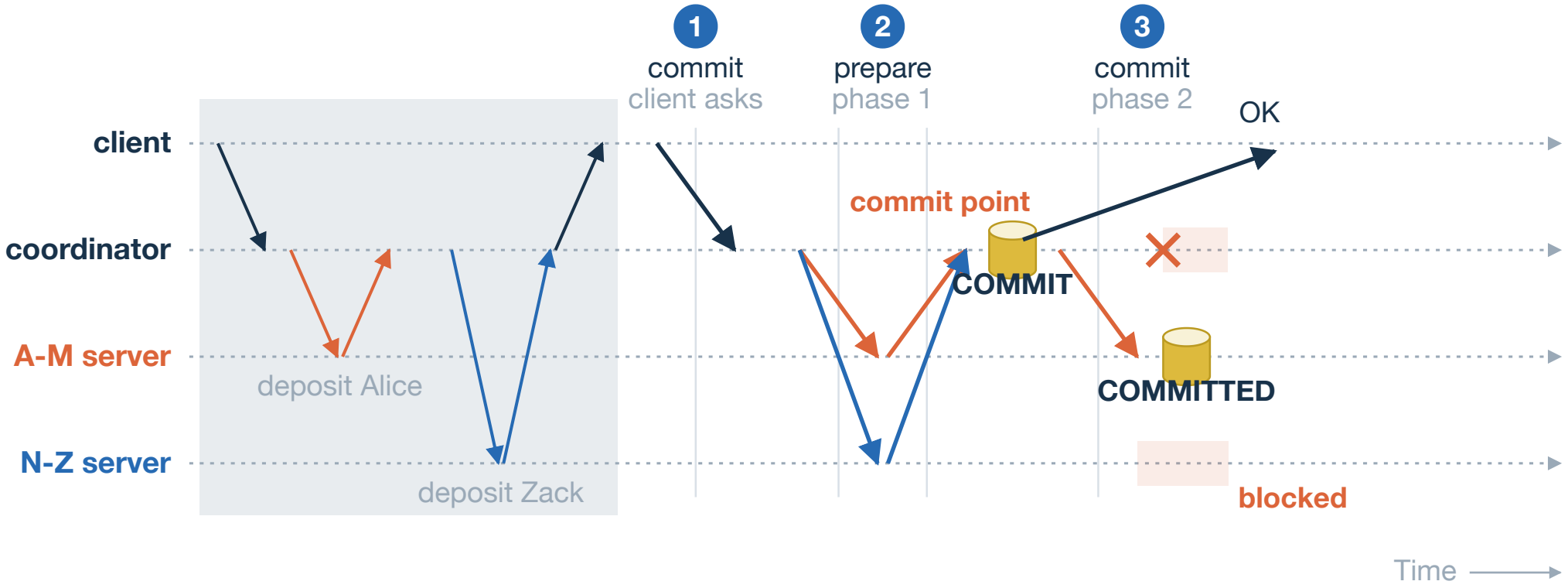
Failure: coordinator failure during prepare



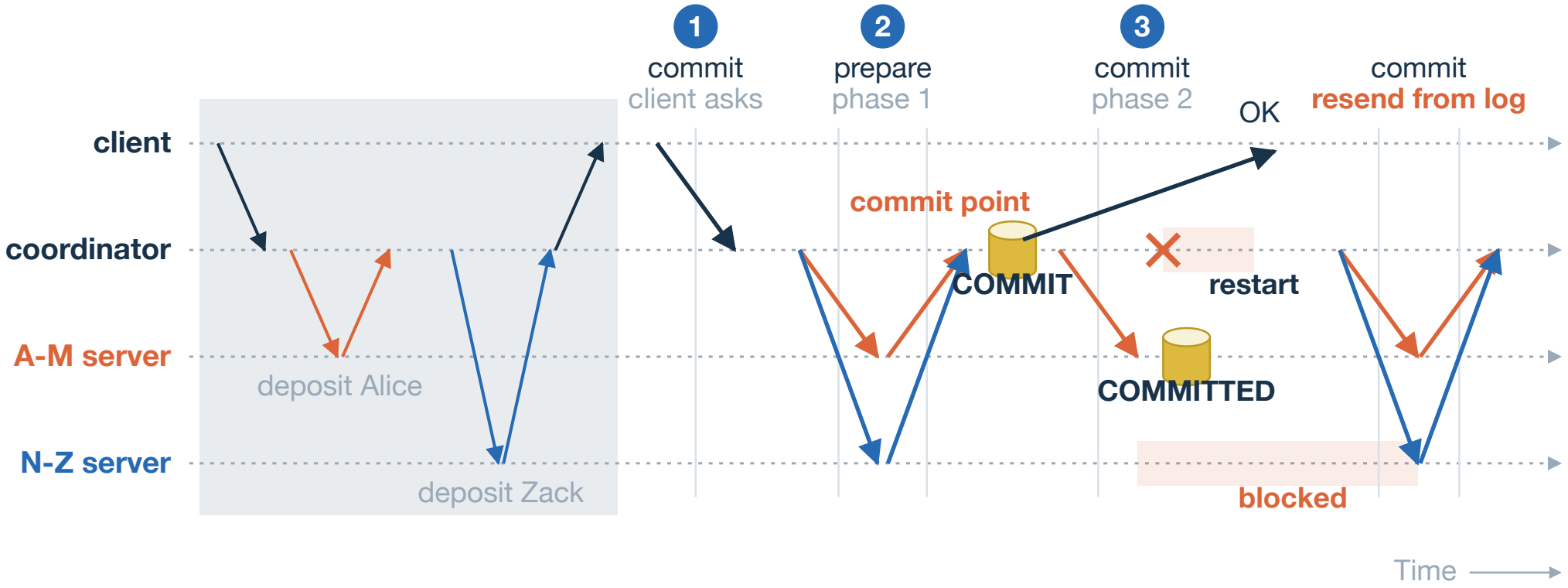
No decision on disk: abort

but both workers block until the coordinator returns

Failure: coordinator failure during commit



Failure: coordinator failure during commit



COMMIT on disk: recovery must complete the commit, not abort

Extension to single-cite atomicity methods

2PL (two-phase locking)

- 2PL: each low-layer TX cannot release its lock until the high-layer TX decides to commit

OCC: the validation & commit phases are done by the coordinator

Logging

- Nearly identical, except that the coordinator needs to log the decision, and the low-layer TX needs to mark a partial-checkpoint state

Checkpoint

- Cannot reclaim the log of a transaction in a PREPARED state
- The solution is simple: record the entry in the checkpoint record

Summary of two-phase commit

Two-phase commit allows us to achieve **multi-site atomicity**: transactions remain atomic even when they require communication with multiple machines (partial failure)

In two-phase commit, failures prior to the commit point can be aborted. If workers (or the coordinator) fail after the commit point, **they recover into the PREPARED state**, and complete the transaction

Remaining challenge: availability under coordinator failures

- Following the coordinator's decision simplifies achieving multi-site atomicity, but what if the coordinator crashes for a long time?
- We will see how to make the coordinator available in the rest of this lecture

Case study: an e-commerce site with two zones

Two zones sell the same items: each zone keeps a replica of the stock

What C and A mean for this store

- Consistency: every user sees the actual stock (not “1 left” when it is 0)
- Availability: every order gets a response, success or failure

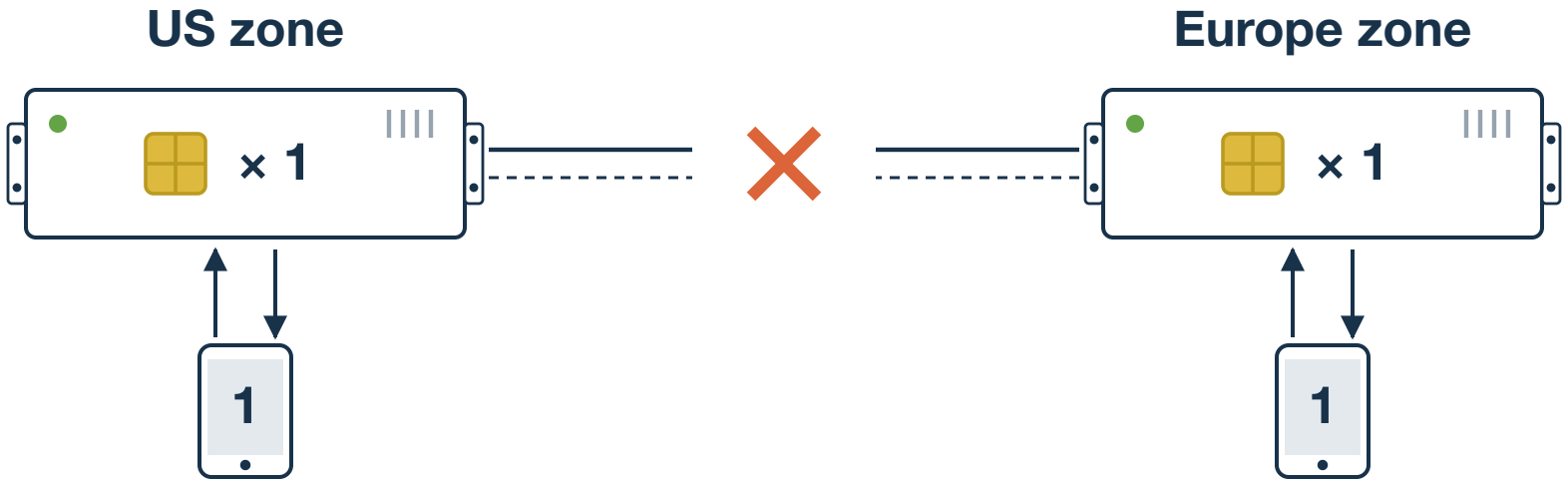


Case study: an e-commerce site with two zones

Two zones sell the same items: each zone keeps a replica of the stock

What C and A mean for this store

- Consistency: every user sees the actual stock (not “1 left” when it is 0)
- Availability: every order gets a response, success or failure



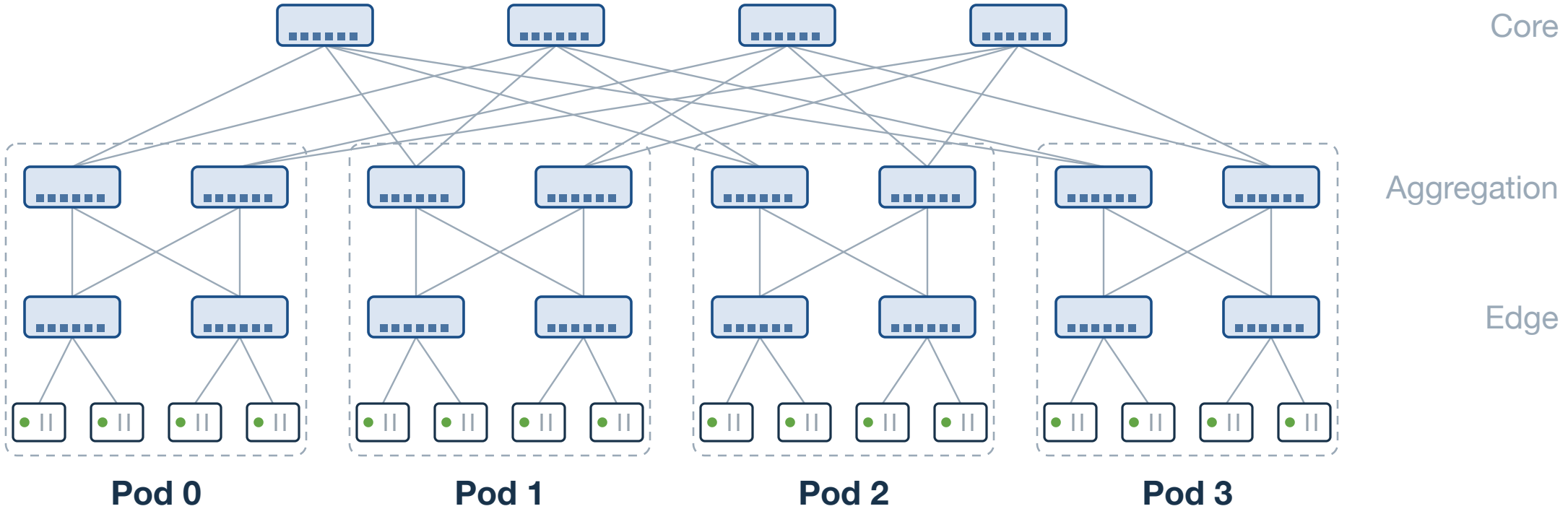
Question: the link between the zones breaks. Can we still keep both C and A?

Another system challenge: network partition

A network partition refers to network decomposition into relatively independent subnets

Can happen when switches fail or are taken offline for upgrades

- For example, suppose the datacenter adopts a fat-tree topology

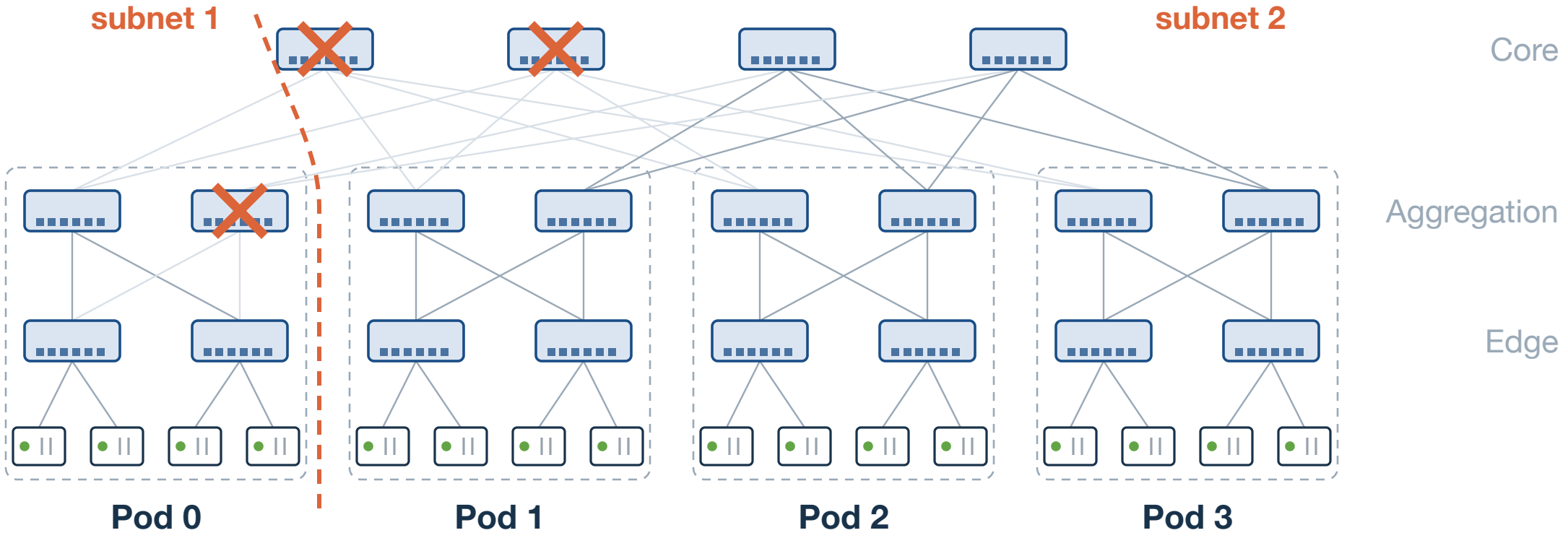


Another system challenge: network partition

A network partition refers to network decomposition into relatively independent subnets

Can happen when switches fail or are taken offline for upgrades

- For example, suppose the datacenter adopts a fat-tree topology



Another system challenge: network partition

Can even happen when being attacked by sharks

[1]

Network partition is usually a reality

- You can never count on the network connectivity

[1] Source: Slate, *Shark attacks threaten Google's undersea internet cables*, 2014, https://slate.com/technology/2014/08/shark-attacks-threaten-google-s-undersea-internet-cables-video.html

The CAP theorem

It is impossible for a distributed system to provide all three at the same time

- Consistency (all replicas act like a single copy: linearizability, all-or-nothing, …)
- Availability (every request receives a response about whether it succeeded or failed)
- Partition tolerance (the system keeps operating when network partition happens)

The famous slogan is "2 out of 3", but P is a fact, not choice

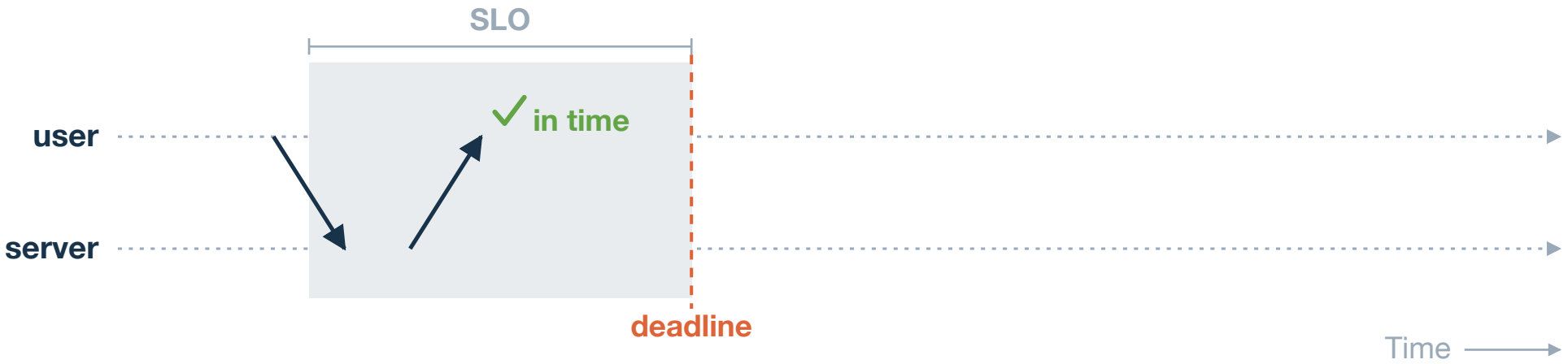
- You can never count on network connectivity: partitions are a fact of life

Question: a partition happens — what exactly stops us from keeping C and A?

First, we clarify what we mean by availability

Availability: every request gets a reply — and gets it within a deadline

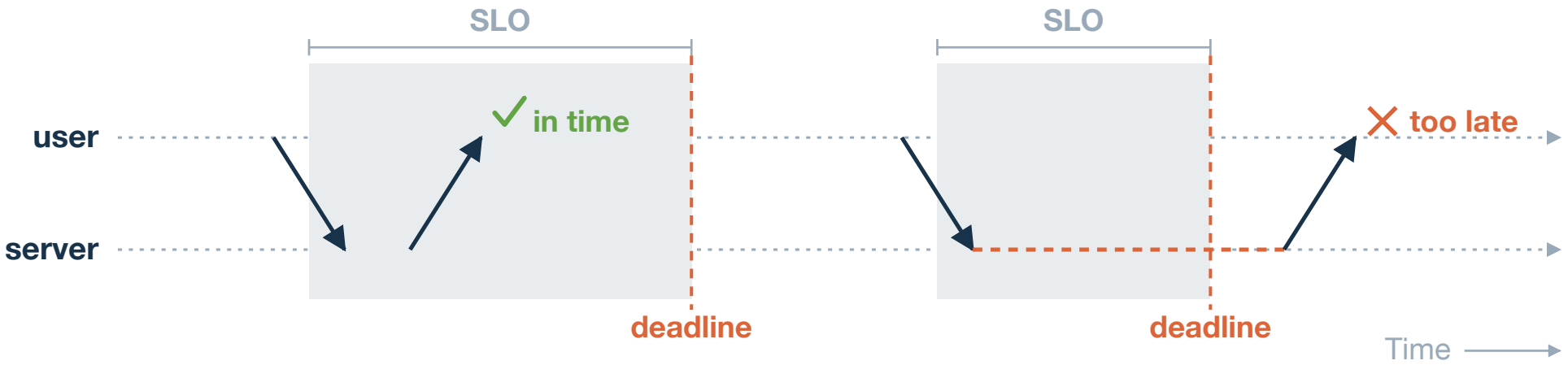
- The deadline comes from the service's SLO (service level objective), which is typically defined by both the application and the service provider



First, we clarify what we mean by availability

Availability: every request gets a reply — and gets it within a deadline

- The deadline comes from the service's SLO (service level objective), which is typically defined by both the application and the service provider

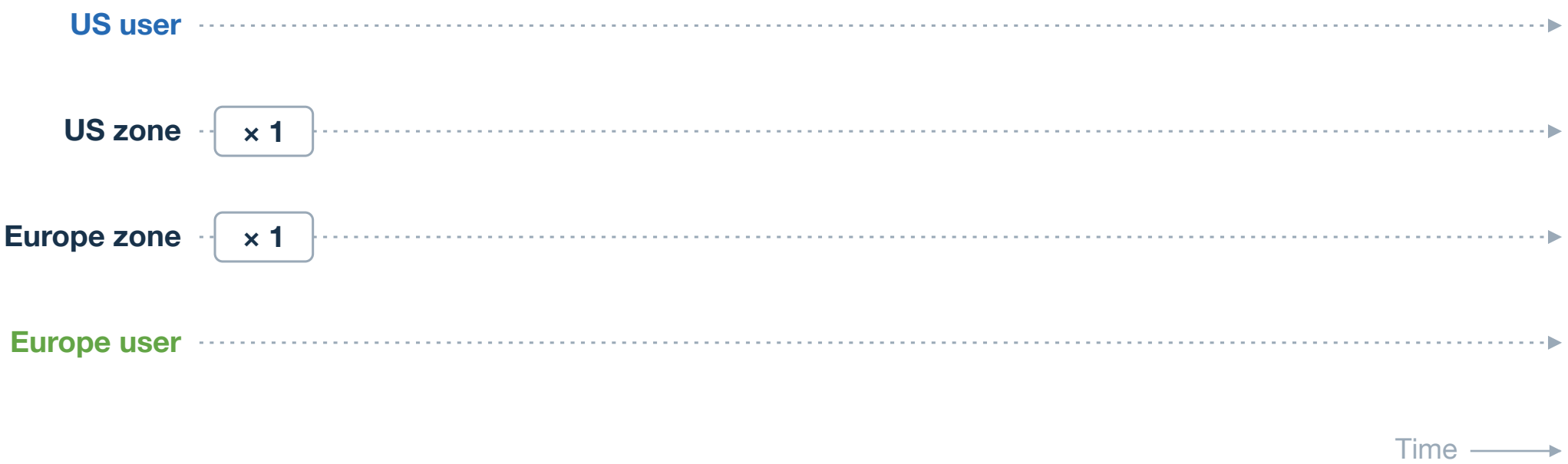


Why is it important? a late reply is no reply: the user has already left

- So there is also the notion of goodput, i.e., the achieved throughput whose latency falls within the SLO

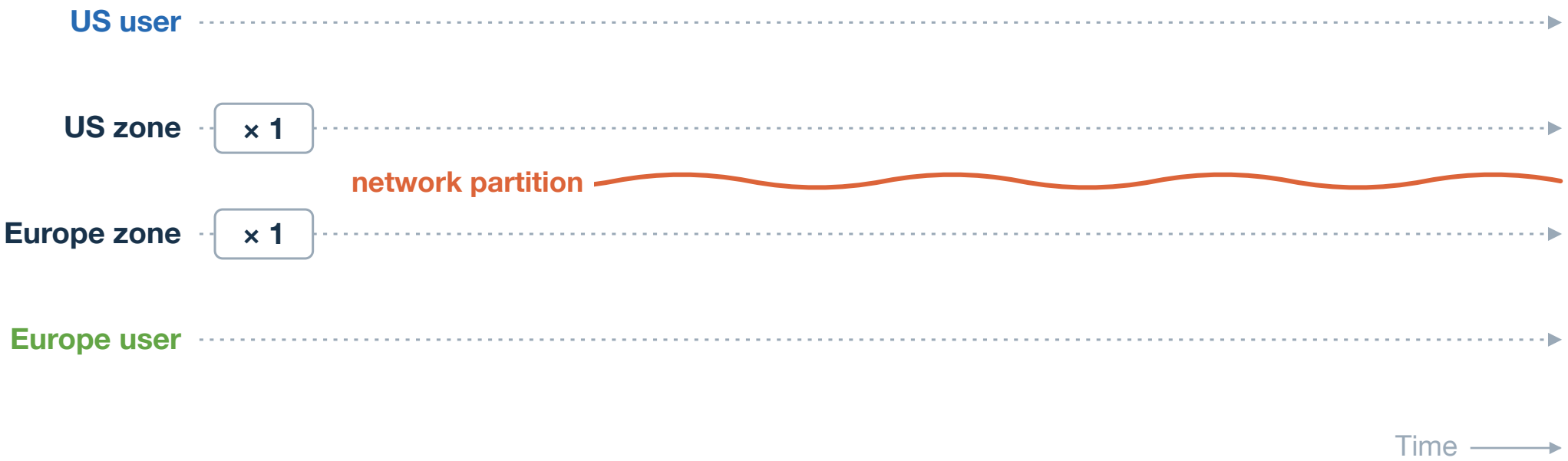
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



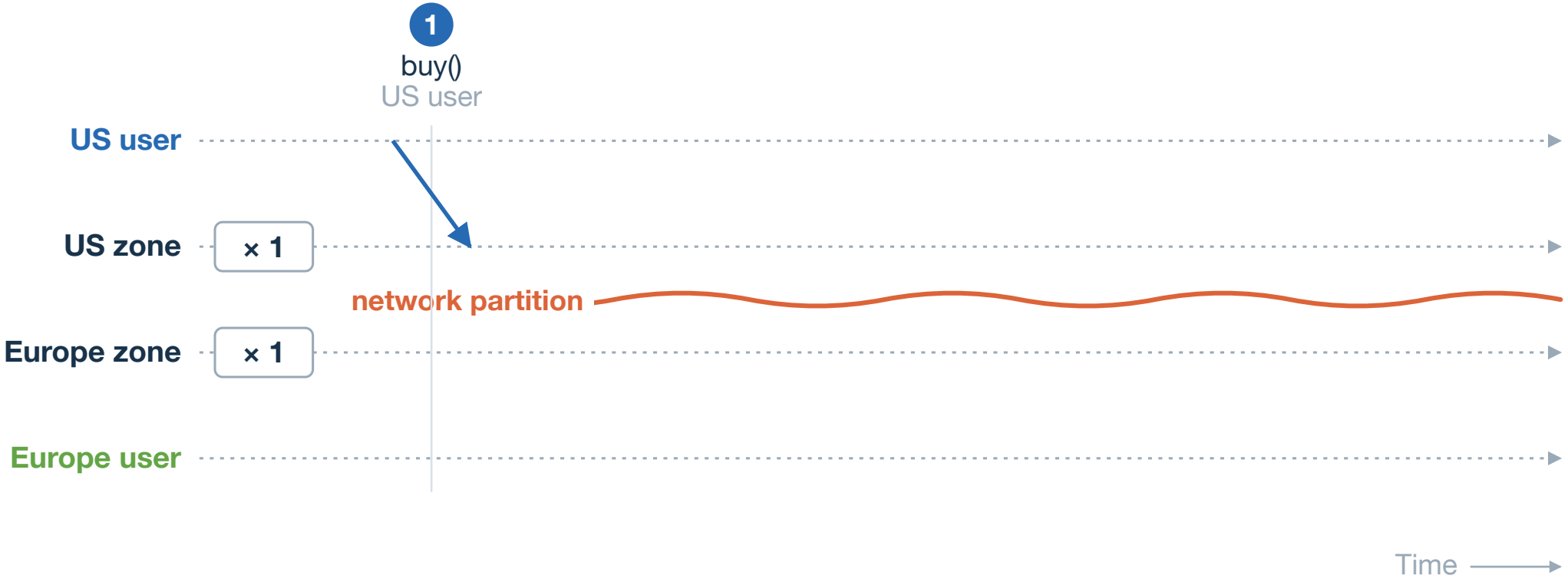
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



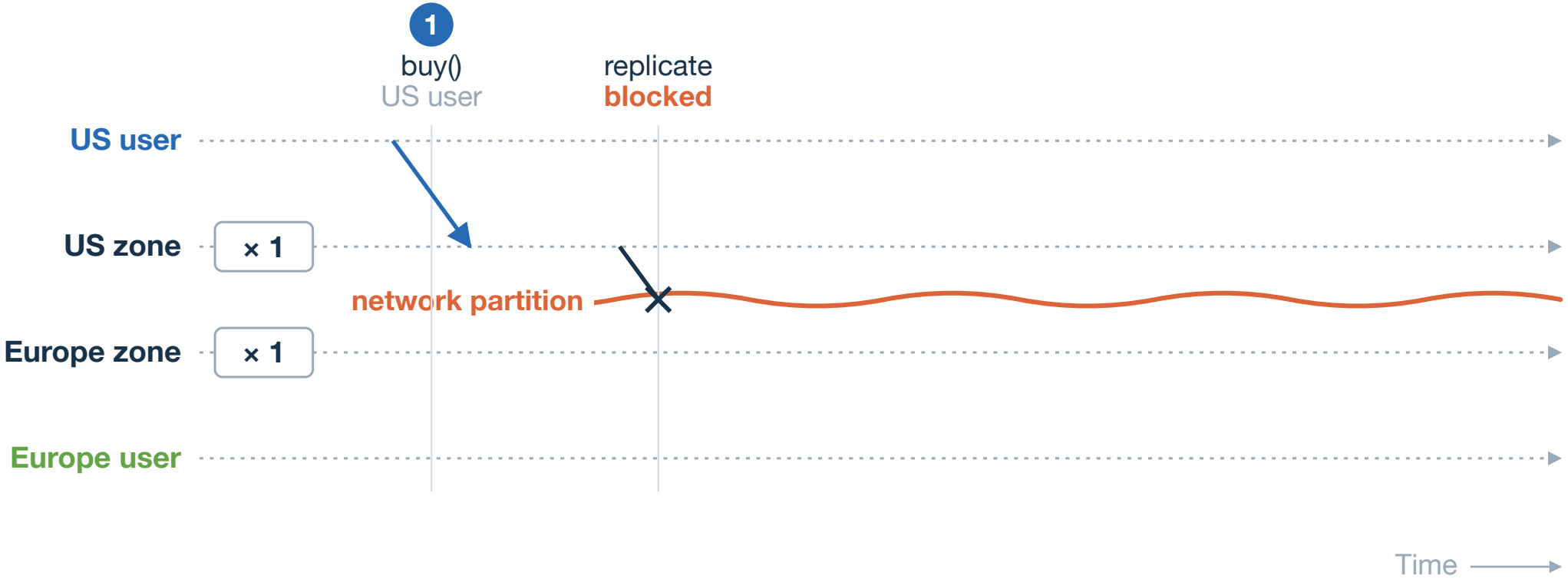
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



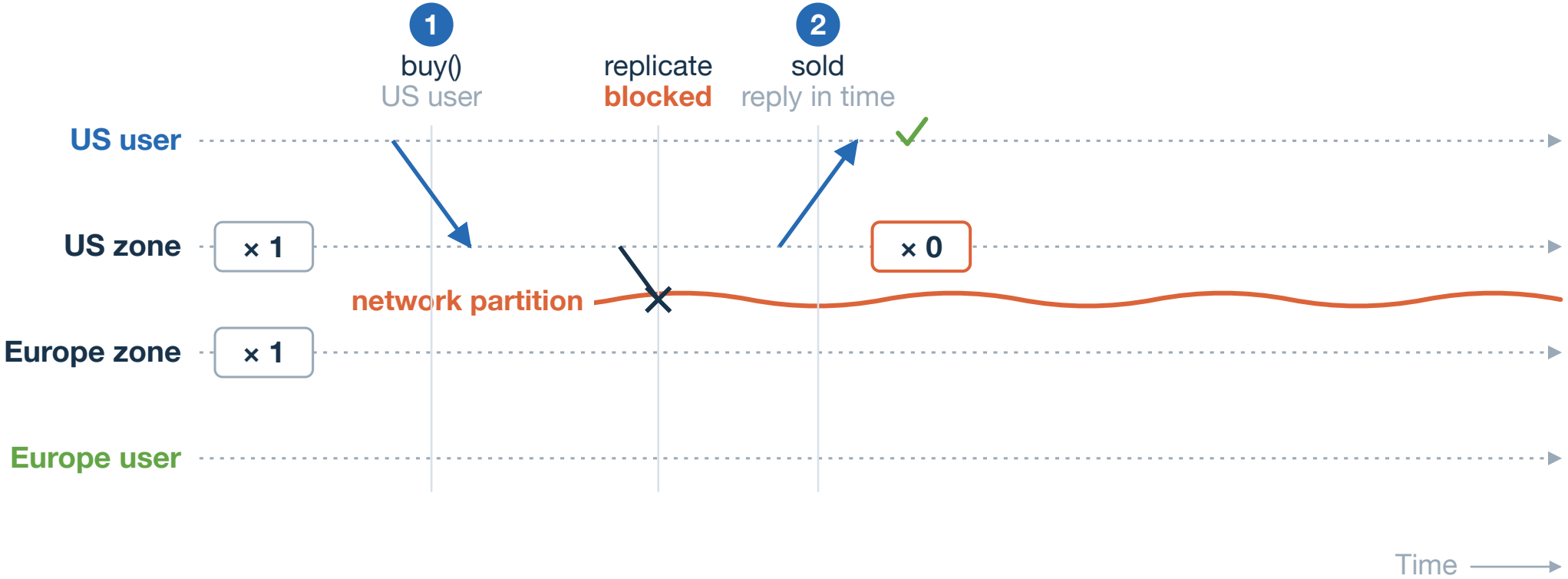
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



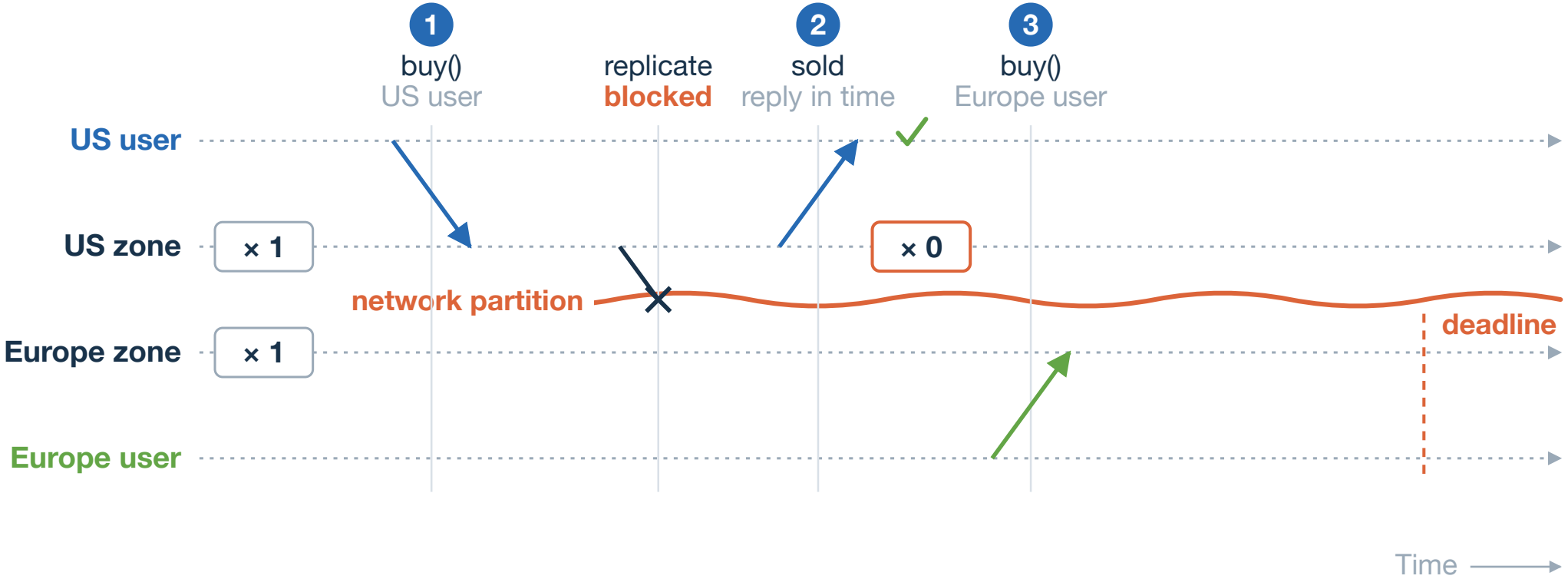
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



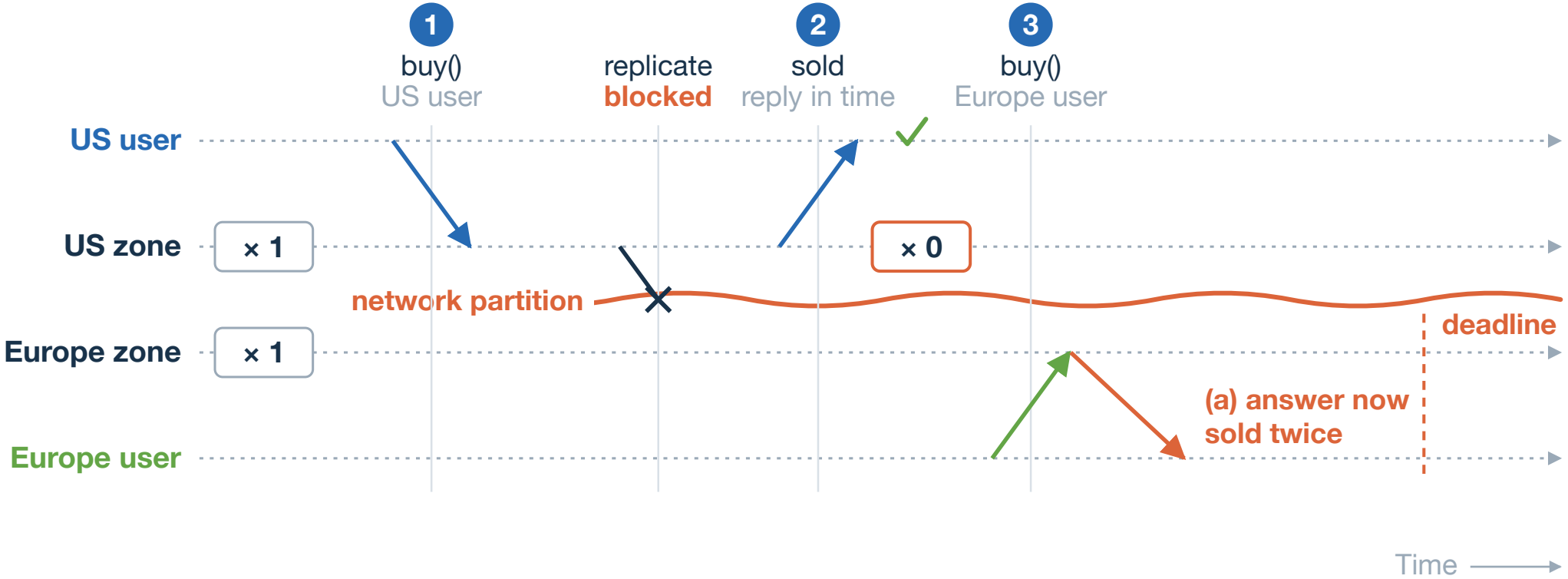
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



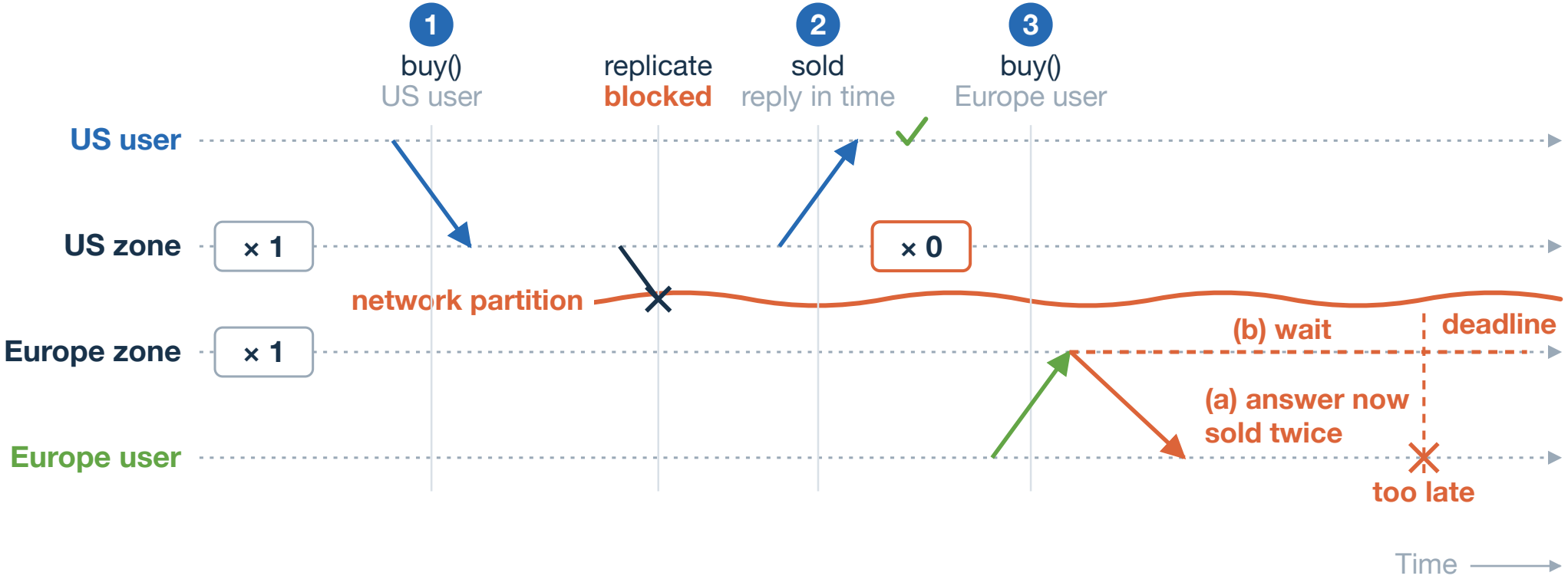
Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



Analysis: why can't we keep both C and A? (An informal argument)

Suppose a network partition happens during the operation



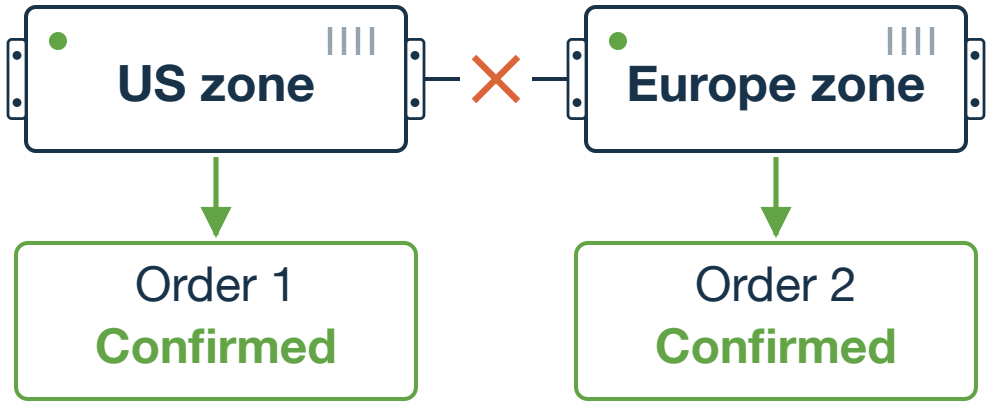
Either way, we sacrifice one property!

Analysis: what do we give up during a partition?

One item left globally. The zones cannot communicate; each receives an order.

AP

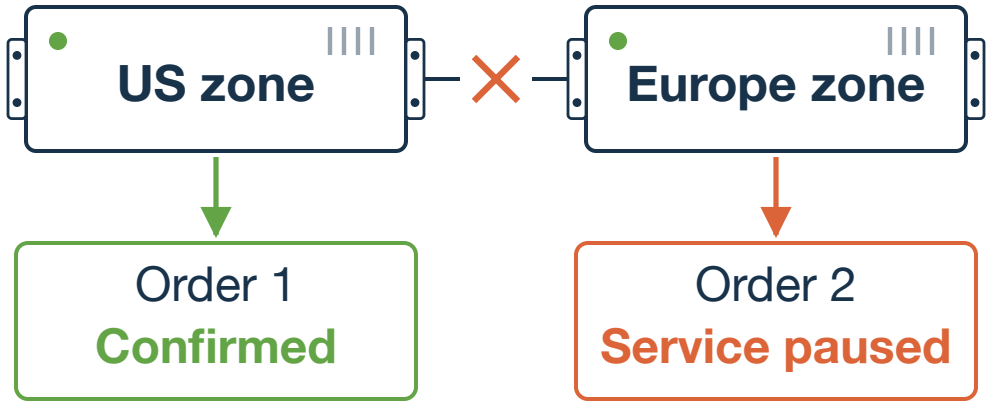
Keep serving



Oversold: C lost

CP

Pause one zone



No overselling; A lost

Service paused is not “sold out”: this zone cannot safely complete the order.

Our current 2PC is a CP protocol

2PC chooses C over A: a transaction never half-commits, whatever fails

And it trades A: waiting is inevitable if the coordinator fails

But if the network partition does not happen (or is short-lived), can we achieve both C and A?

For example: 2PC now trades A even no network partition (e.g., coordinator failures)

How to achieve (high) availability? Replication!

Replication is everywhere

Replicate a data/service multiple times

Cases we have seen so far

- GFS in distributed file system, ChatAPP

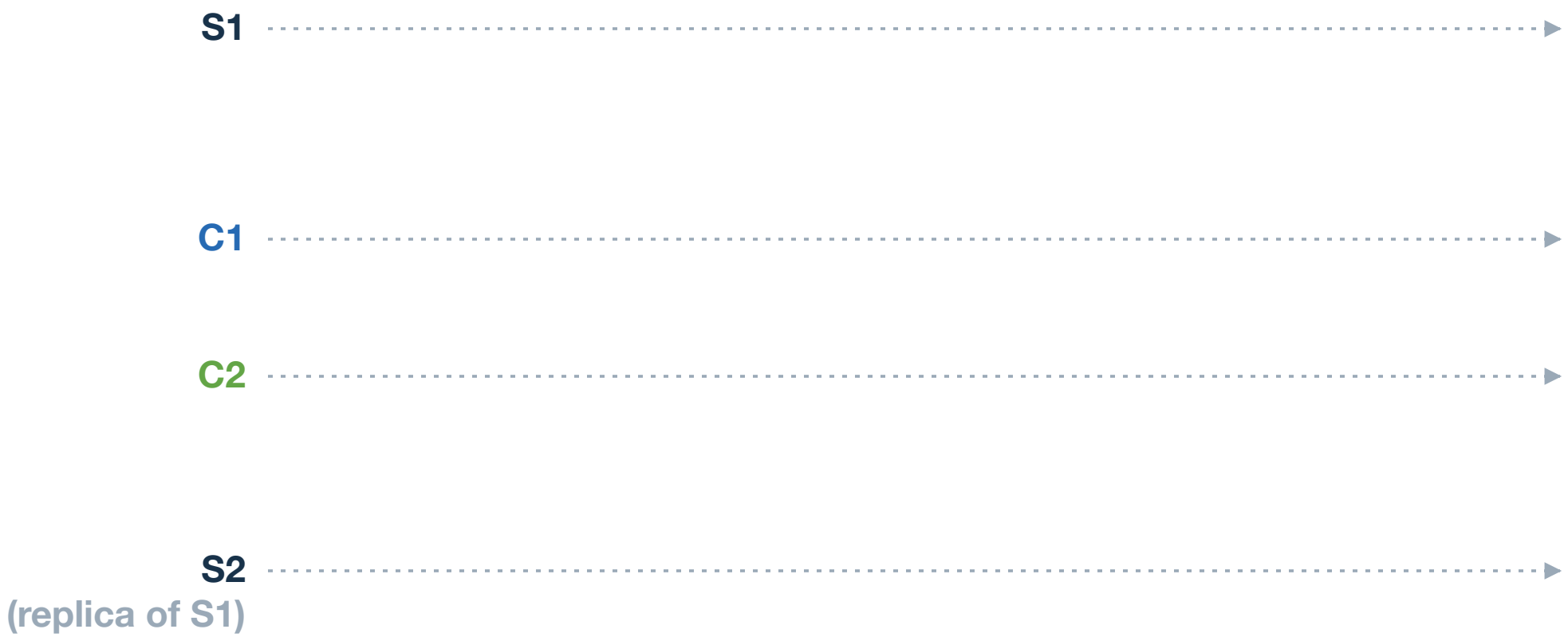
For performance

- Higher throughput: replicas can serve concurrently
- Lower latency: cache is also a form of replication, and requests can be spread across replicas

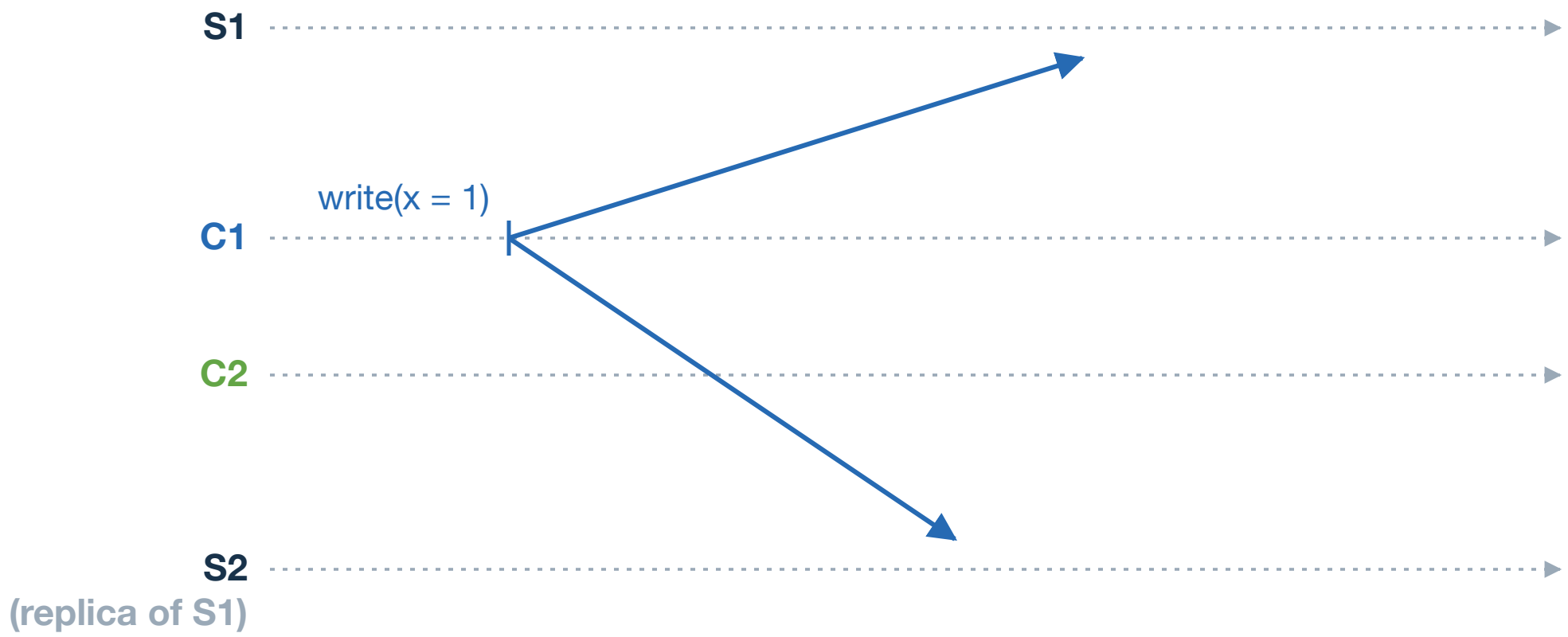
For fault tolerance

- Maintain availability even if some replicas fail

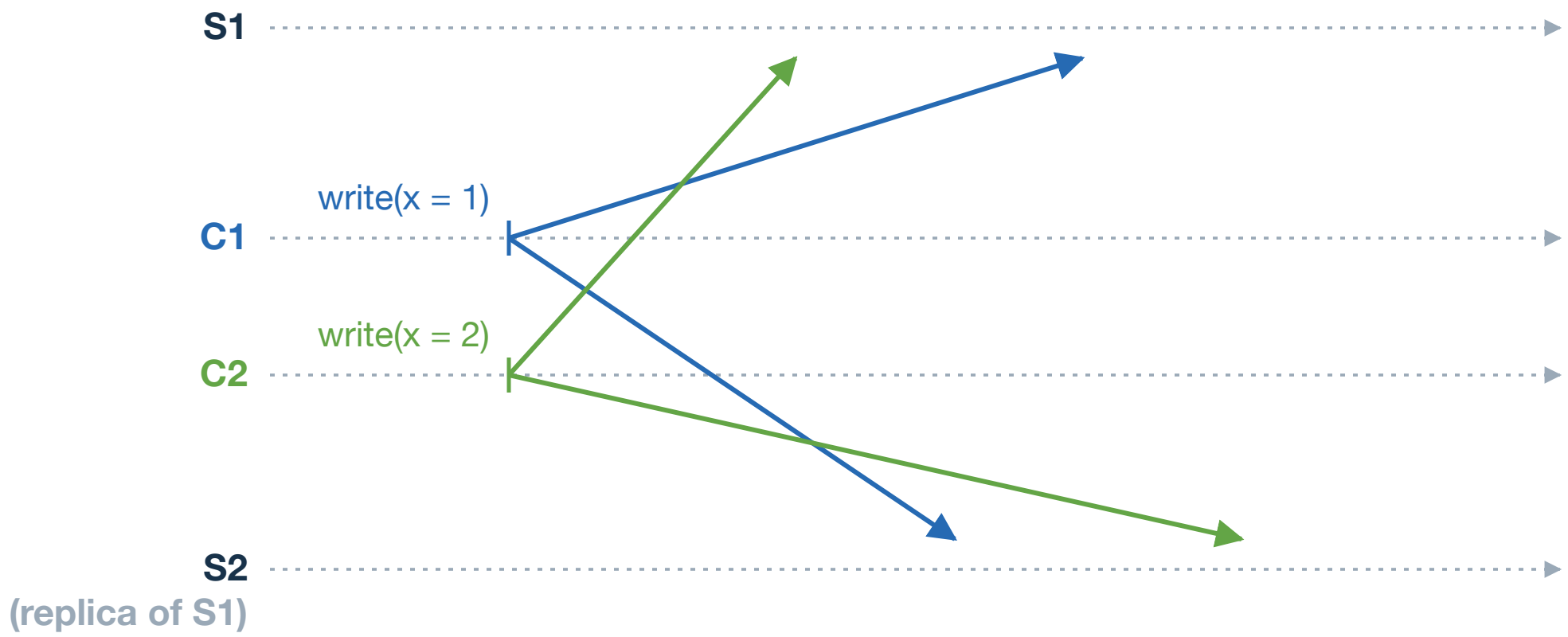
Challenge: inconsistency due to reordered operations



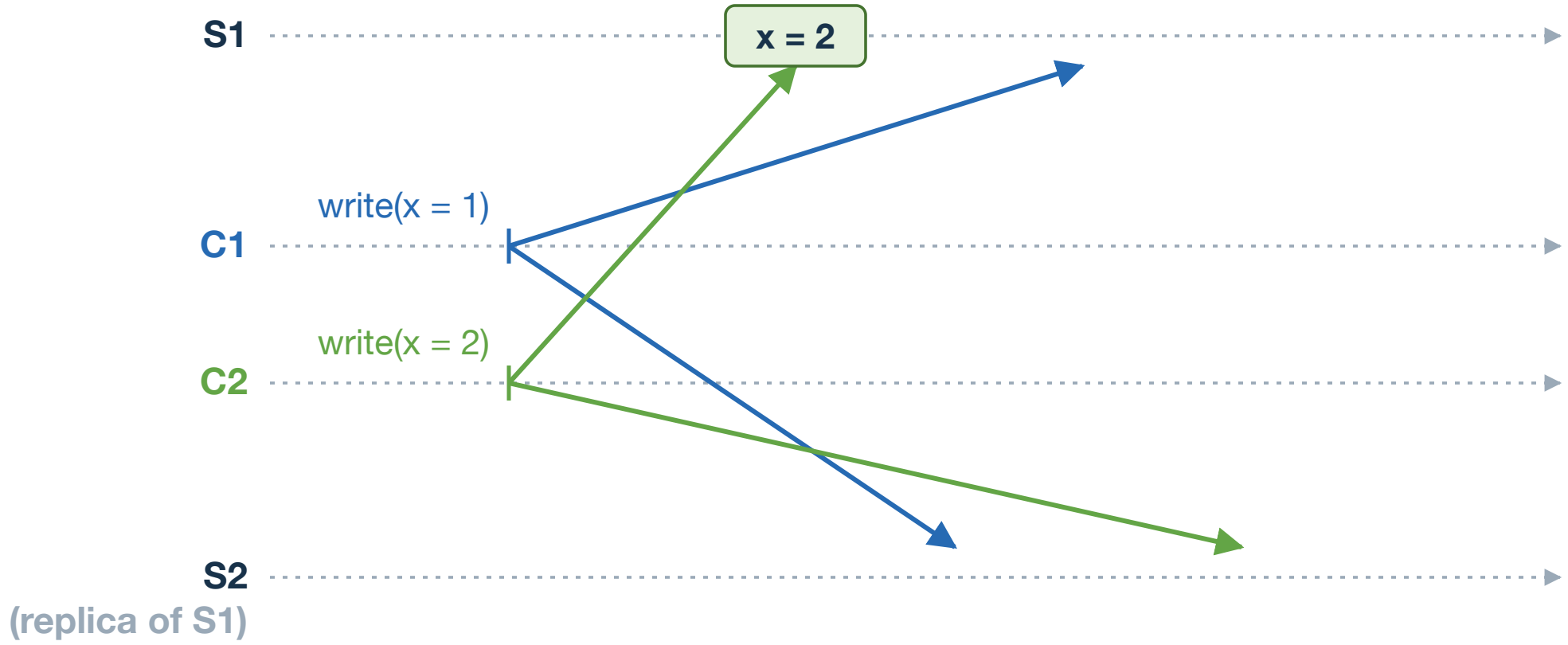
Challenge: inconsistency due to reordered operations



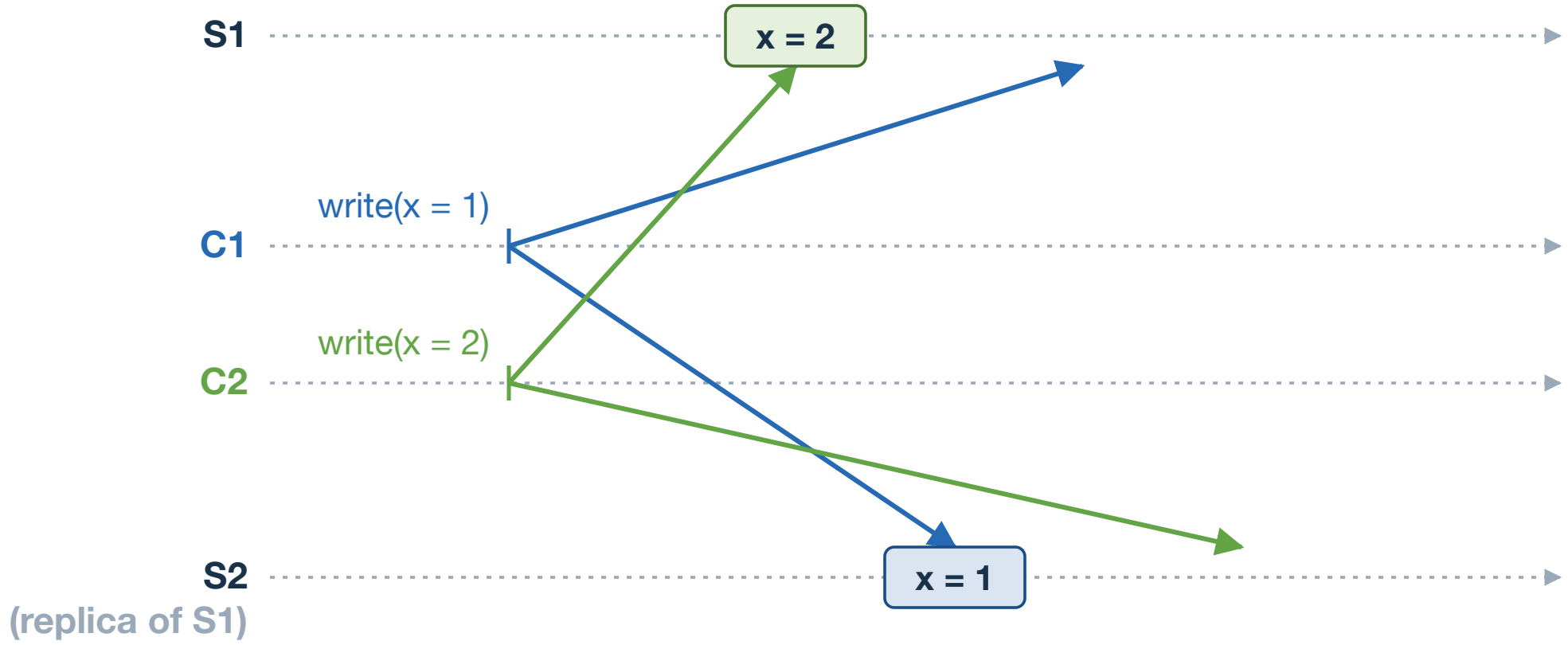
Challenge: inconsistency due to reordered operations



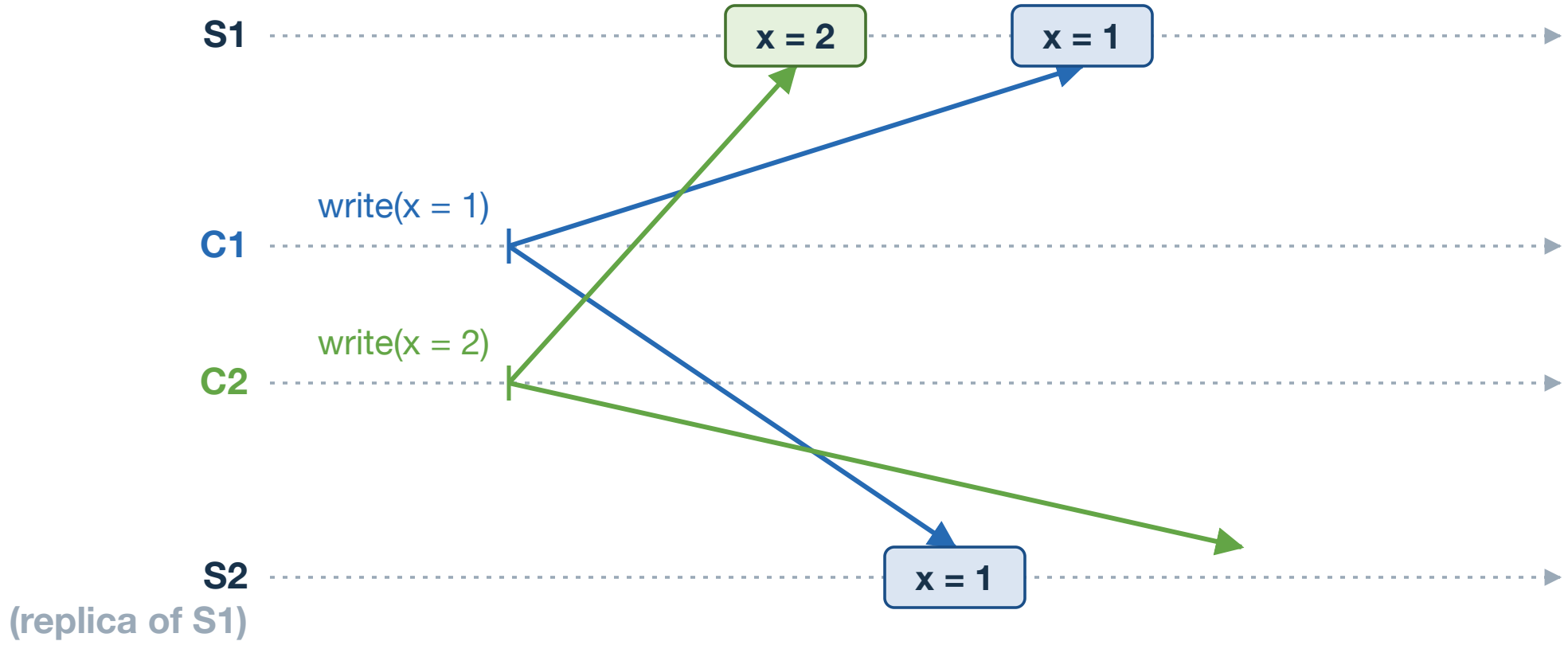
Challenge: inconsistency due to reordered operations



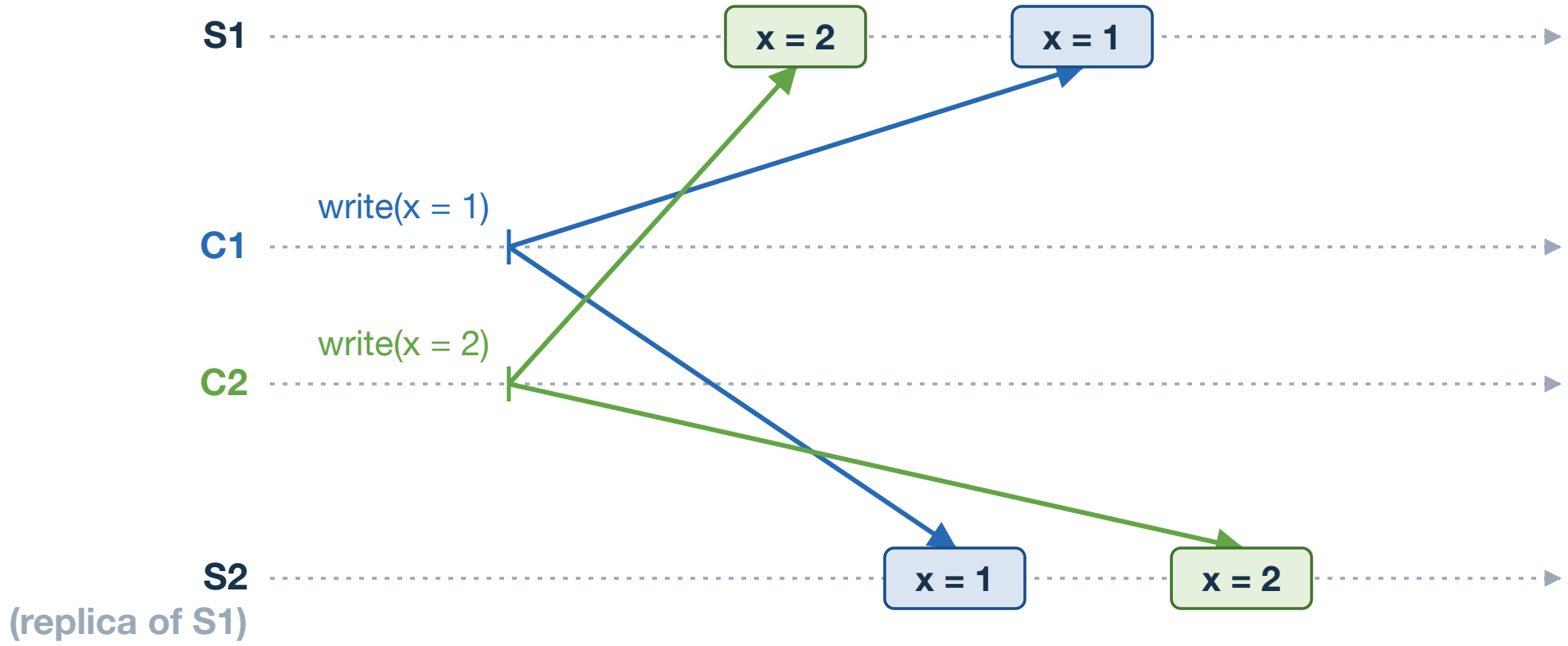
Challenge: inconsistency due to reordered operations



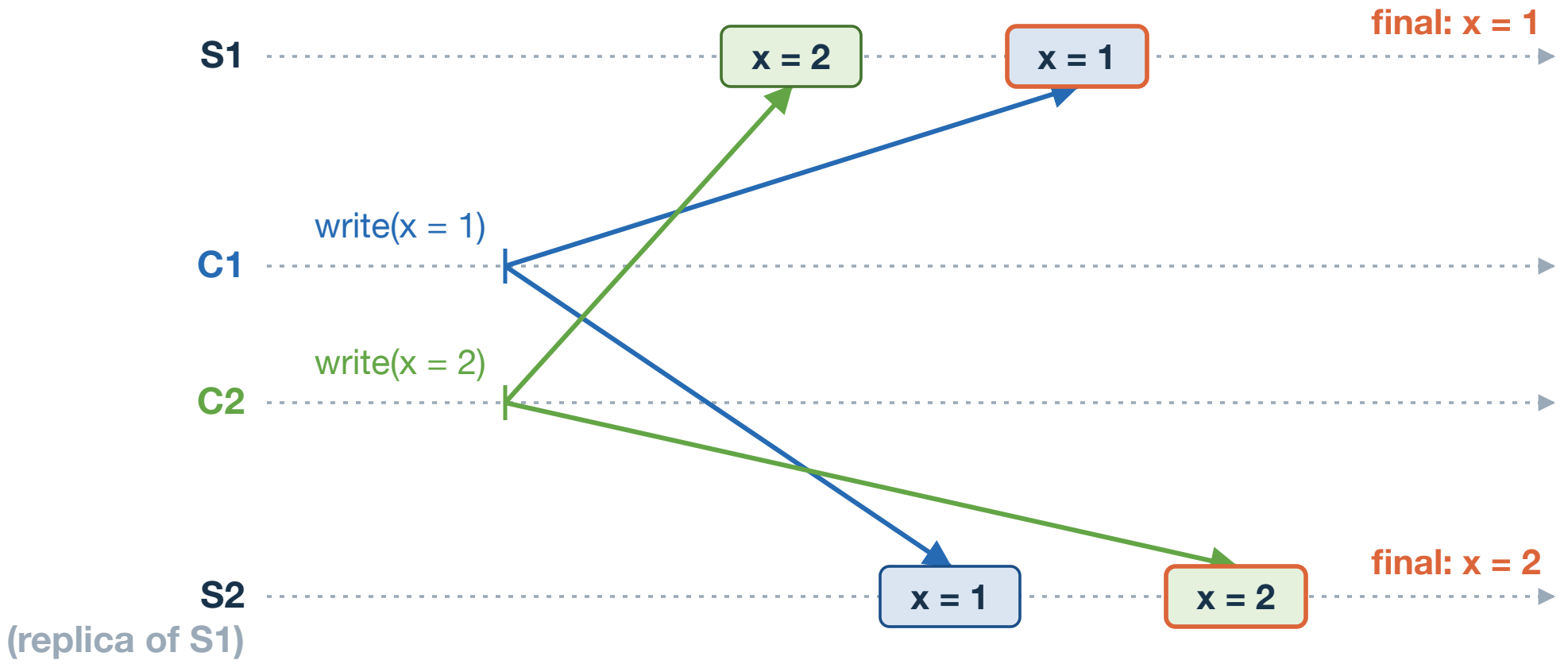
Challenge: inconsistency due to reordered operations



Challenge: inconsistency due to reordered operations



Challenge: inconsistency due to reordered operations



Question: do replicas see the same operation order?

Requests sent to different servers can arrive in different order

- How do we ensure the servers remain consistent?

Optimistic Replication (e.g., eventual consistency)

- Tolerate inconsistency, and fix things up later
- Works well when out-of-sync replicas are acceptable

Pessimistic Replication (e.g., primary-backup for linearizability discussed before)

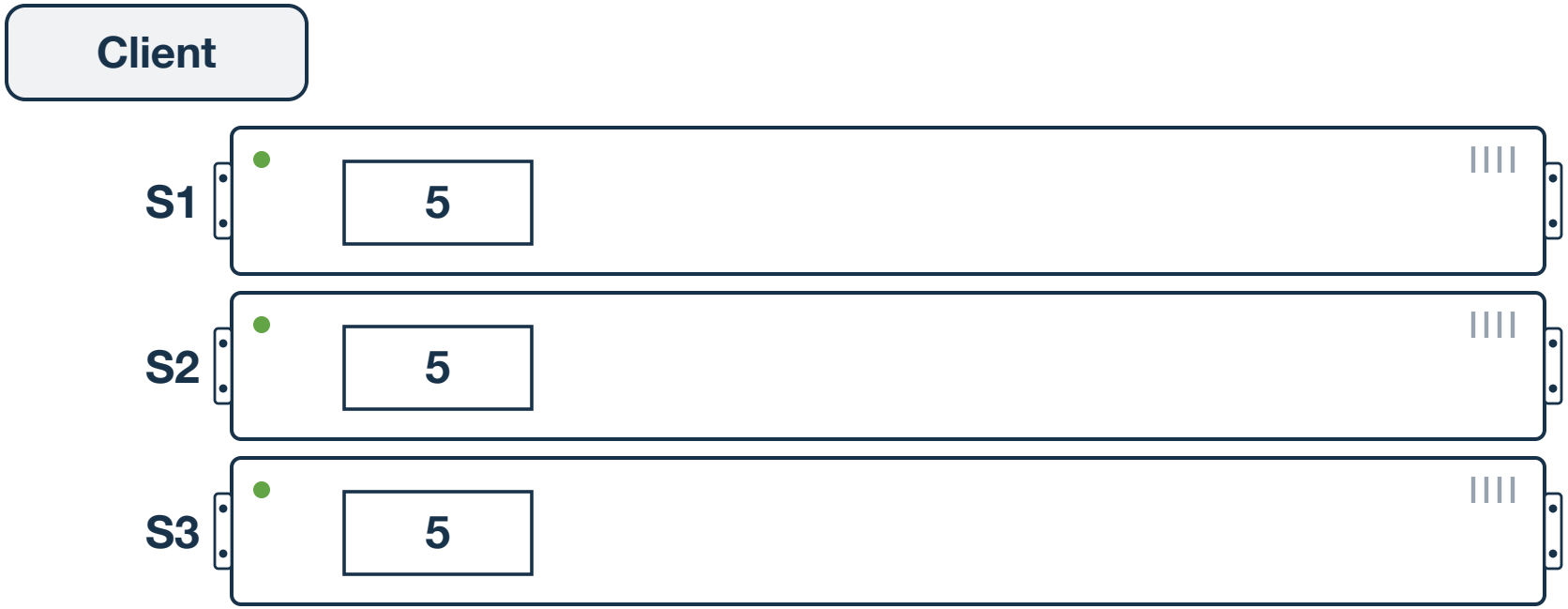
- Ensure strong consistency between replicas
- However, we have not discussed how to provide availability yet

Question: which replication is desirable for 2PC?

Replicated state machine (RSM) for pessimistic replication

Replicated state machine (RSM): A general model to simplify consistent replicas of a server:

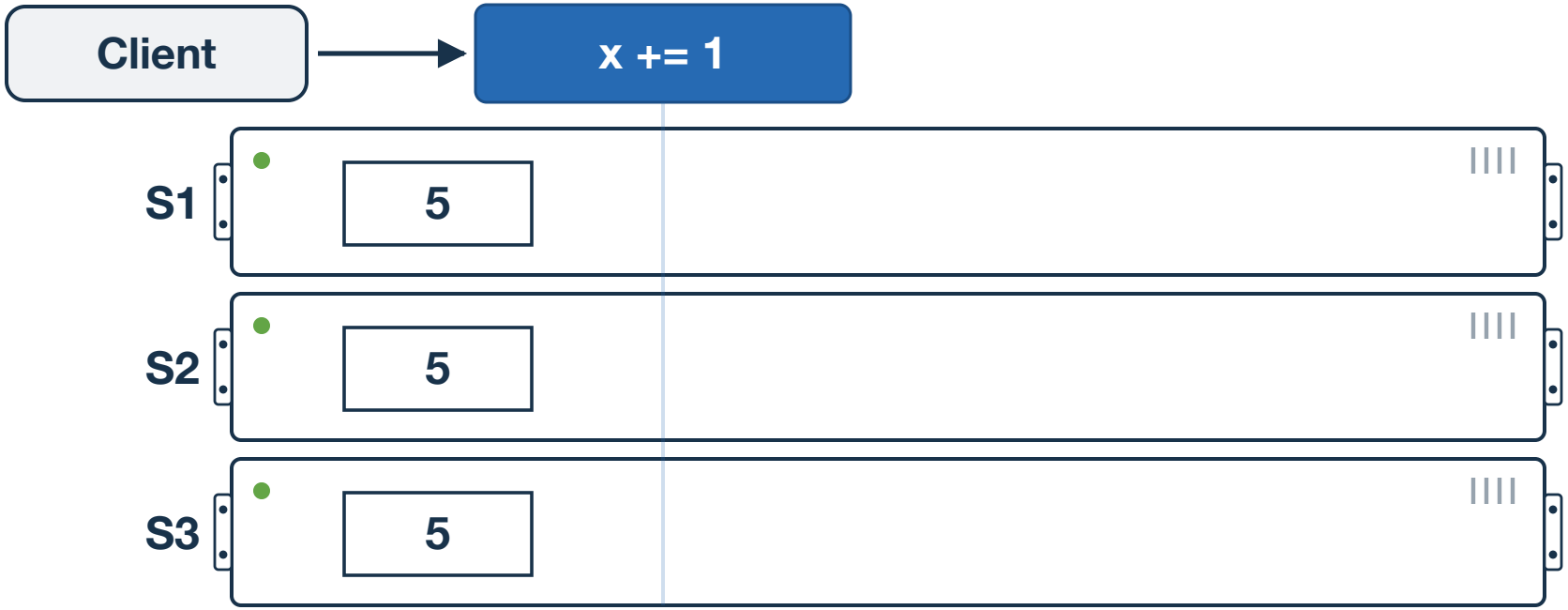
- Each replica is modelled as a state machine
- Suppose we ensure all replicas receive the same operations (including their order)
- Assuming each operation is deterministic, then all replicas must be in the same state



Replicated state machine (RSM) for pessimistic replication

Replicated state machine (RSM): A general model to simplify consistent replicas of a server:

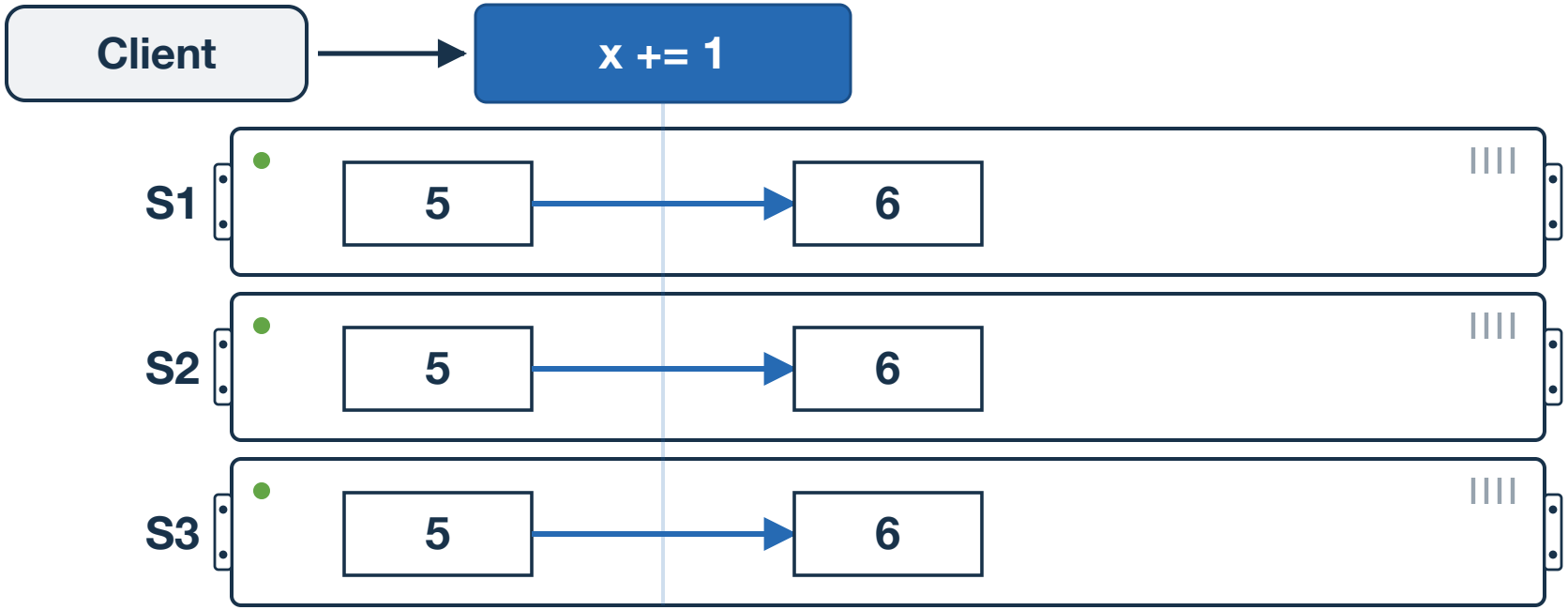
- Each replica is modelled as a state machine
- Suppose we ensure all replicas receive the same operations (including their order)
- Assuming each operation is deterministic, then all replicas must be in the same state



Replicated state machine (RSM) for pessimistic replication

Replicated state machine (RSM): A general model to simplify consistent replicas of a server:

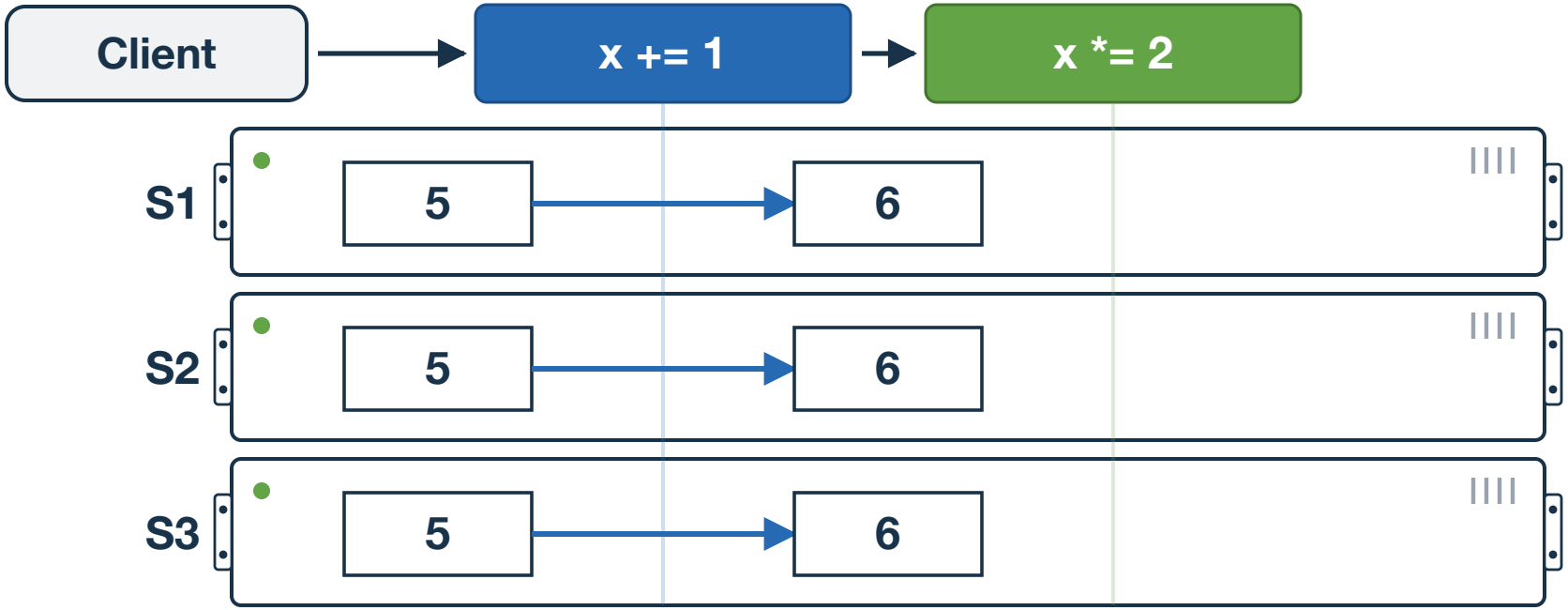
- Each replica is modelled as a state machine
- Suppose we ensure all replicas receive the same operations (including their order)
- Assuming each operation is deterministic, then all replicas must be in the same state



Replicated state machine (RSM) for pessimistic replication

Replicated state machine (RSM): A general model to simplify consistent replicas of a server:

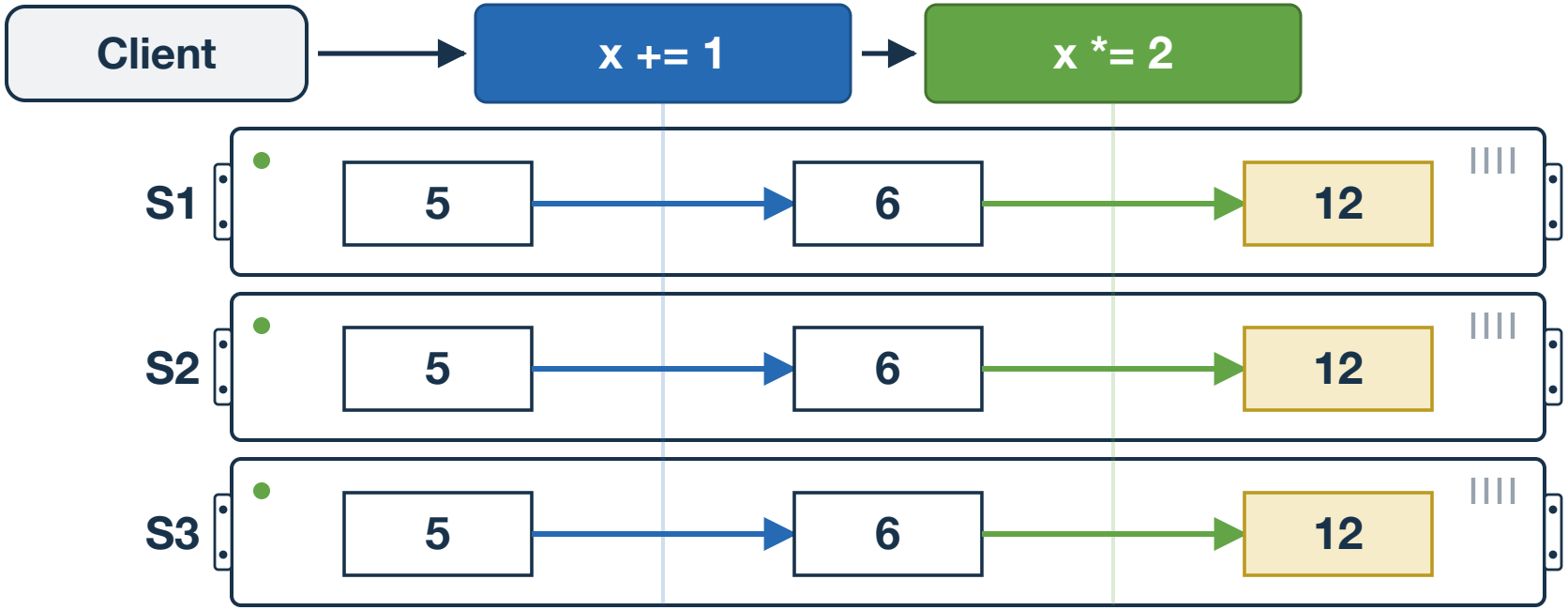
- Each replica is modelled as a state machine
- Suppose we ensure all replicas receive the same operations (including their order)
- Assuming each operation is deterministic, then all replicas must be in the same state



Replicated state machine (RSM) for pessimistic replication

Replicated state machine (RSM): A general model to simplify consistent replicas of a server:

- Each replica is modelled as a state machine
- Suppose we ensure all replicas receive the same operations (including their order)
- Assuming each operation is deterministic, then all replicas must be in the same state



Primary-backup model revisited

One server is responsible for ordering inputs received from clients

Primary does important stuff

- Ensures that it sends all inputs to the backup before ACKing the ops
- Chooses an ordering for all operations, so that the primary and backup agree (i.e., one writer)
- Decides all non-deterministic values (e.g., random(), time())

It can also recruit new backups after servers fail, but needs to take care (see later)

Primary-backup can trivially support linearizability,
but it (our current version) does not support availability

What if the primary fails?

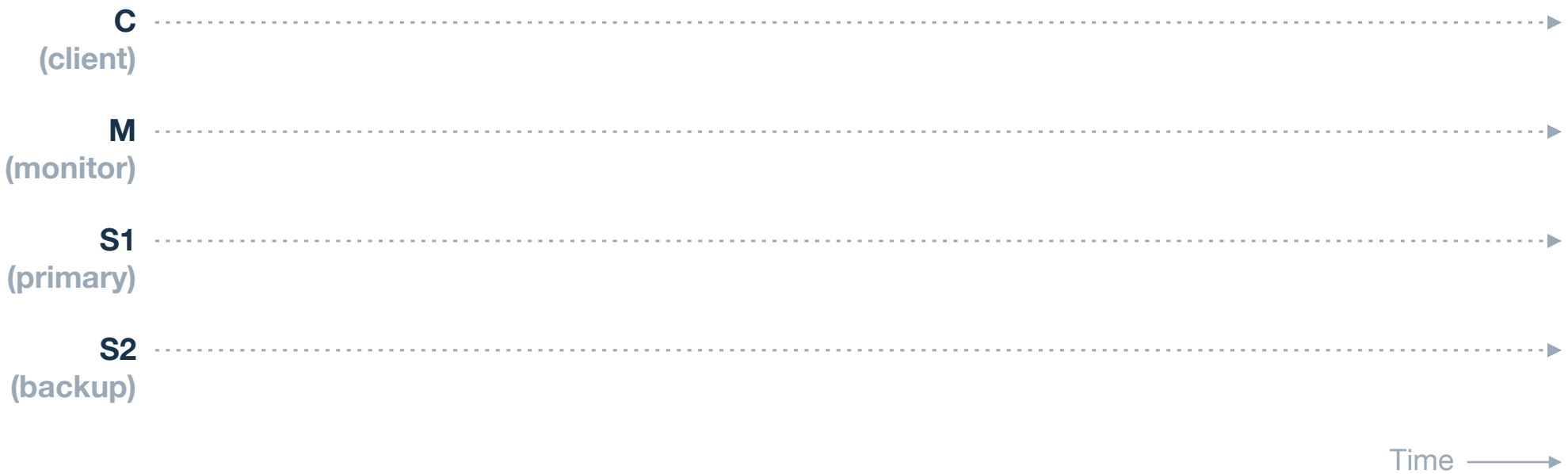
Idea 1: Have human/agent decide when to switch from primary to backup

- A human is reasonable for a small web service, but not at large scale
- Agents cannot guarantee success

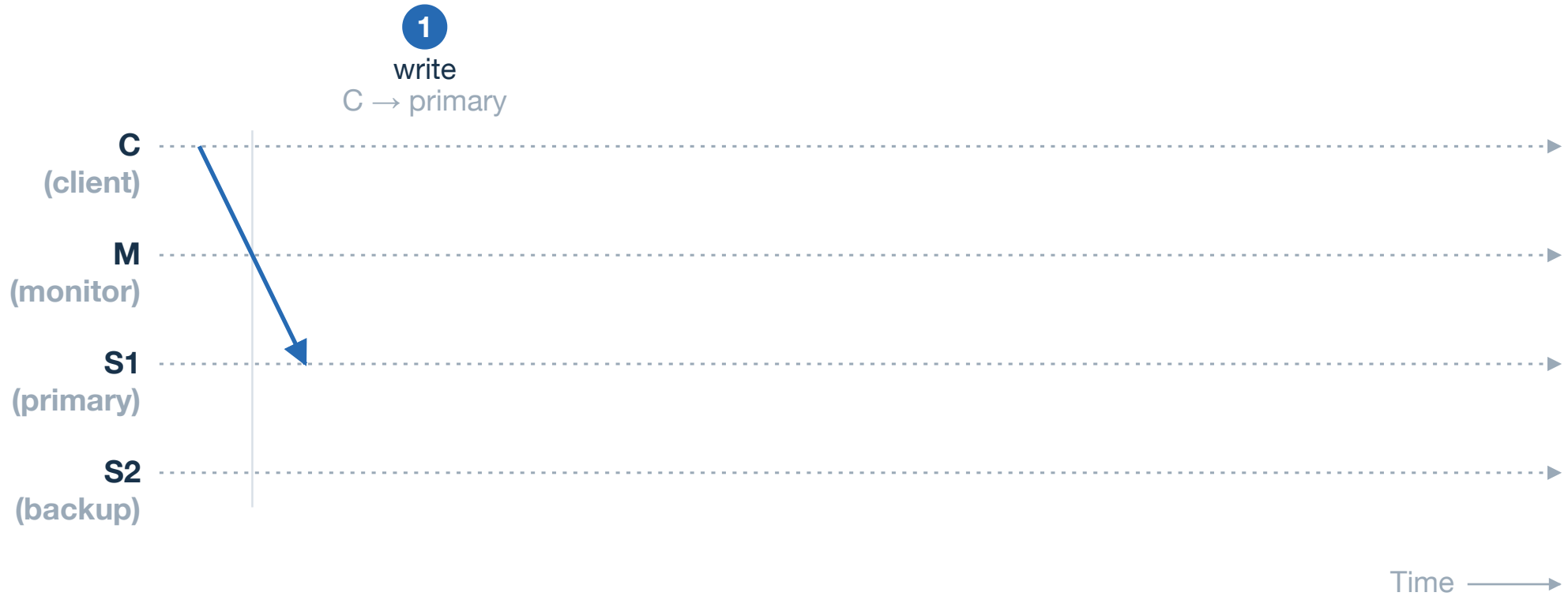
Idea 2: Monitor (a computer program, not driven by AI) automatically switches the primary

- The monitor only watches the primary (via heartbeats); clients still write the primary directly
- More challenging than you might think, e.g., split brain

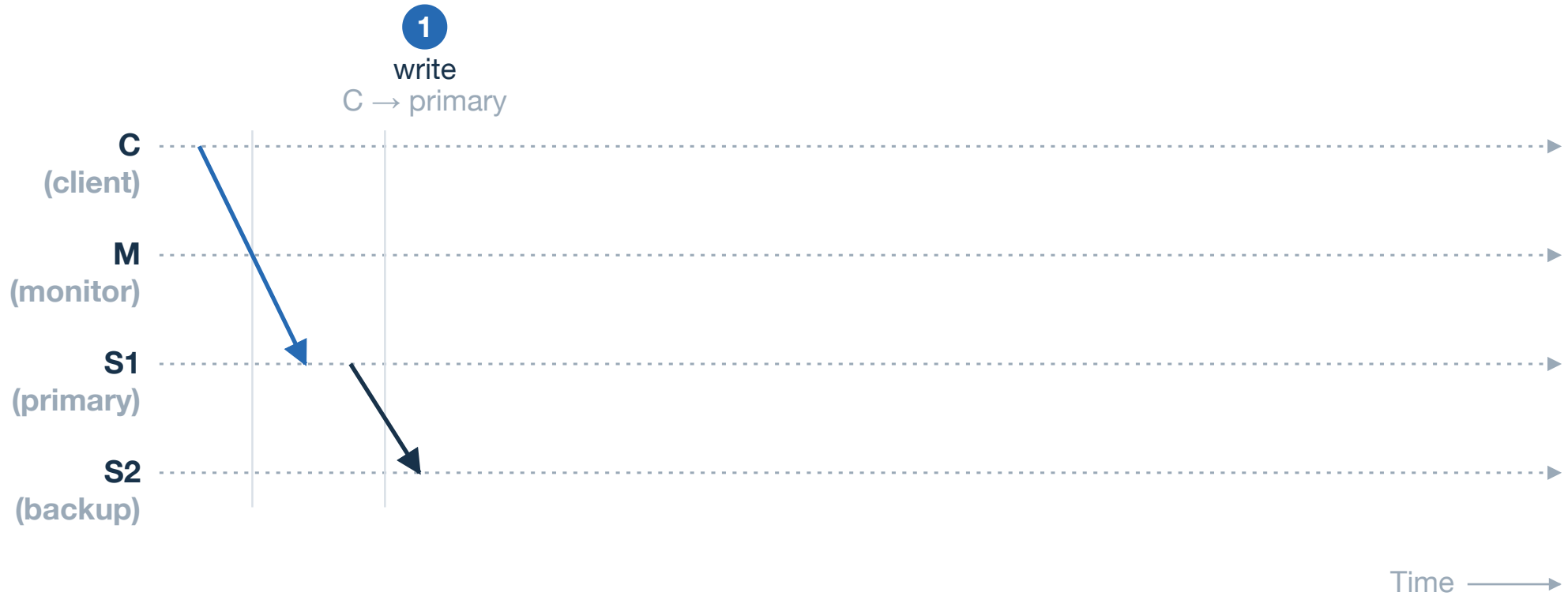
Primary-backup model with availability: the first try



Primary-backup model with availability: the first try



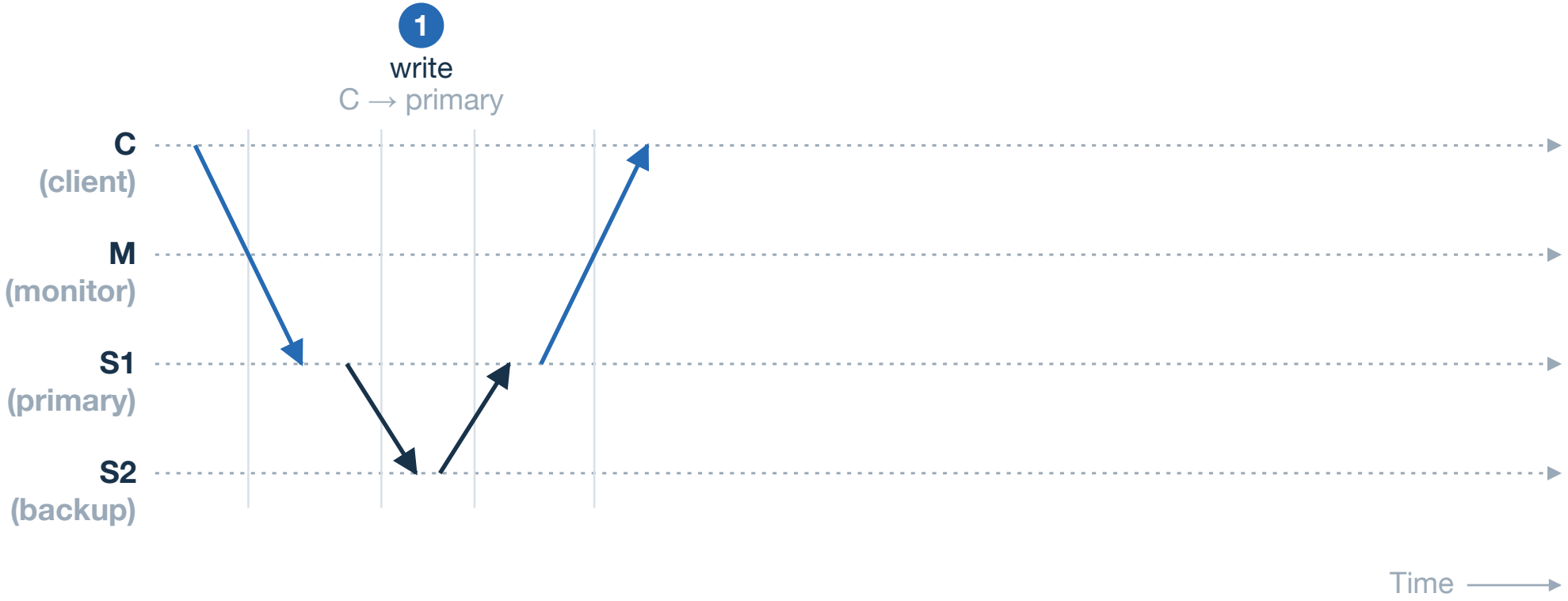
Primary-backup model with availability: the first try



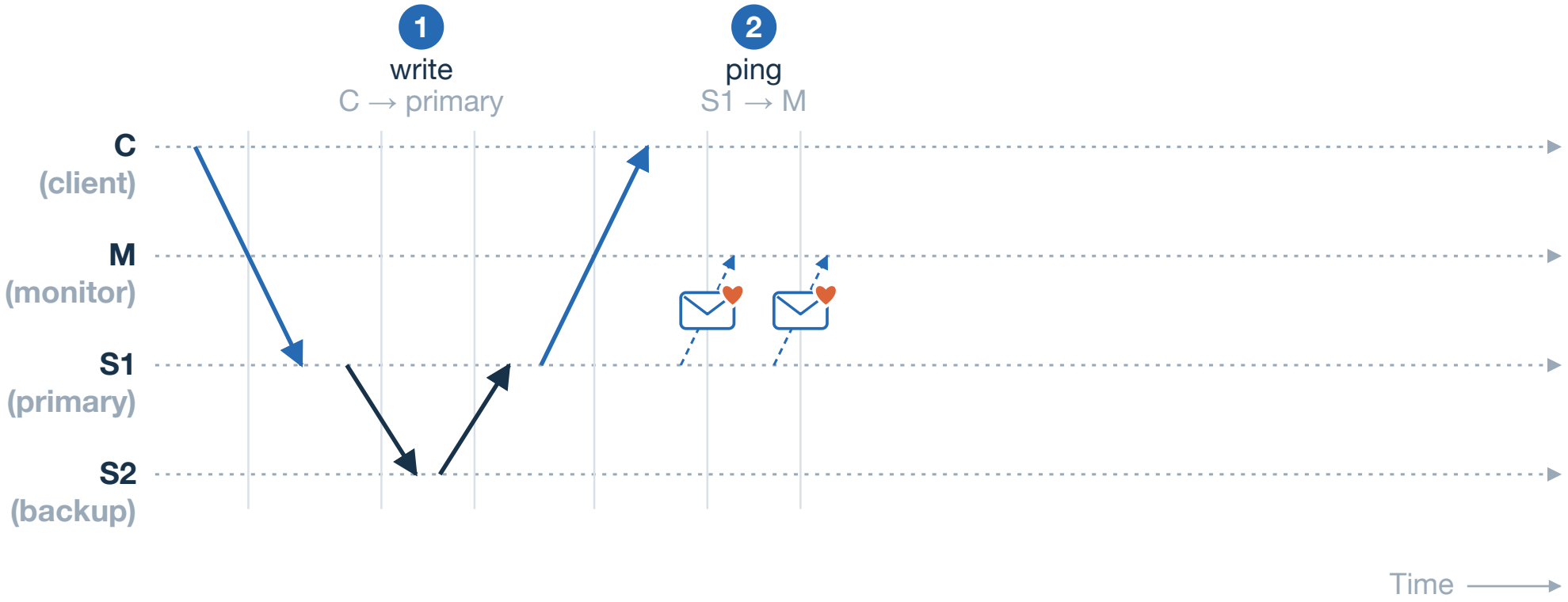
Primary-backup model with availability: the first try



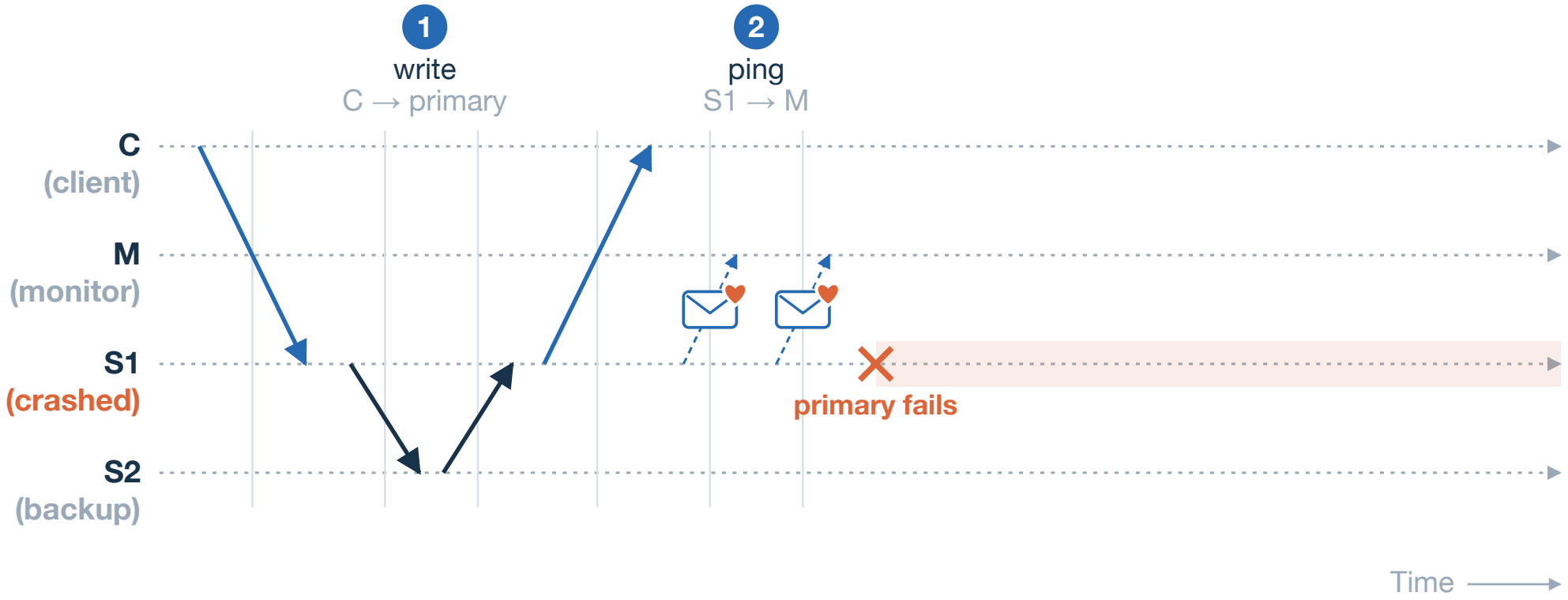
Primary-backup model with availability: the first try



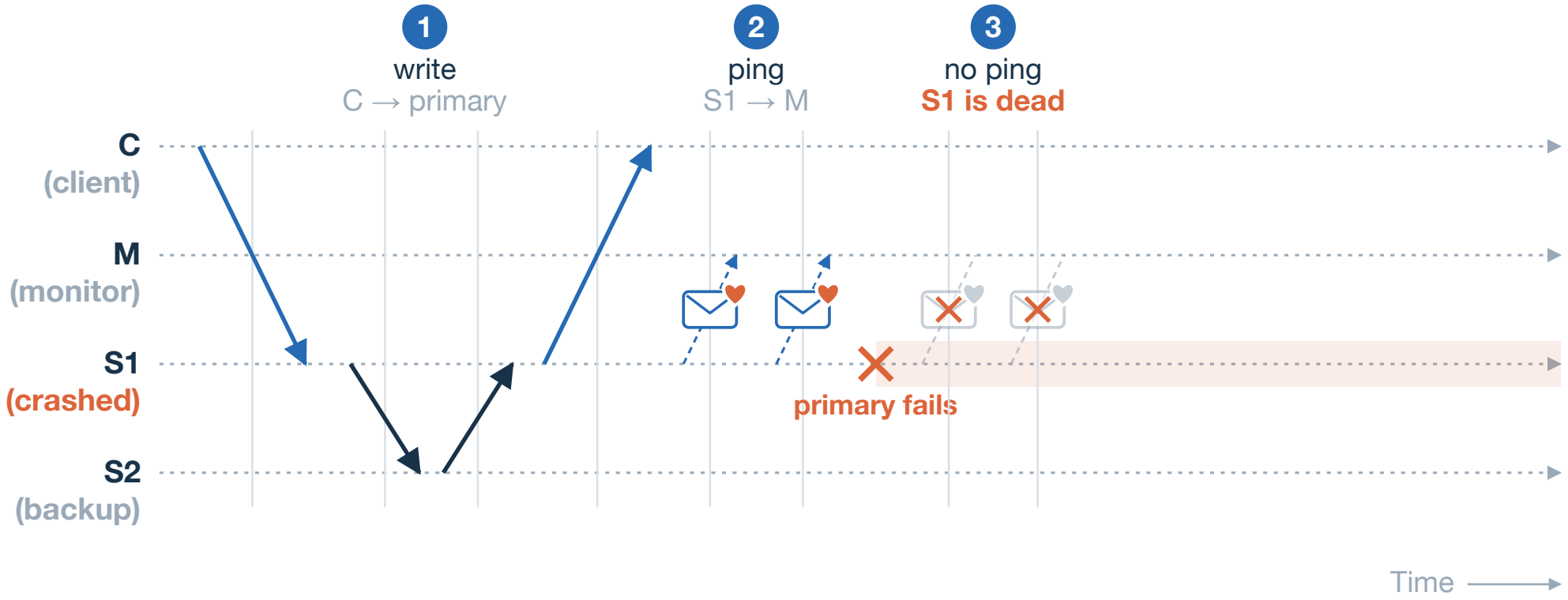
Primary-backup model with availability: the first try



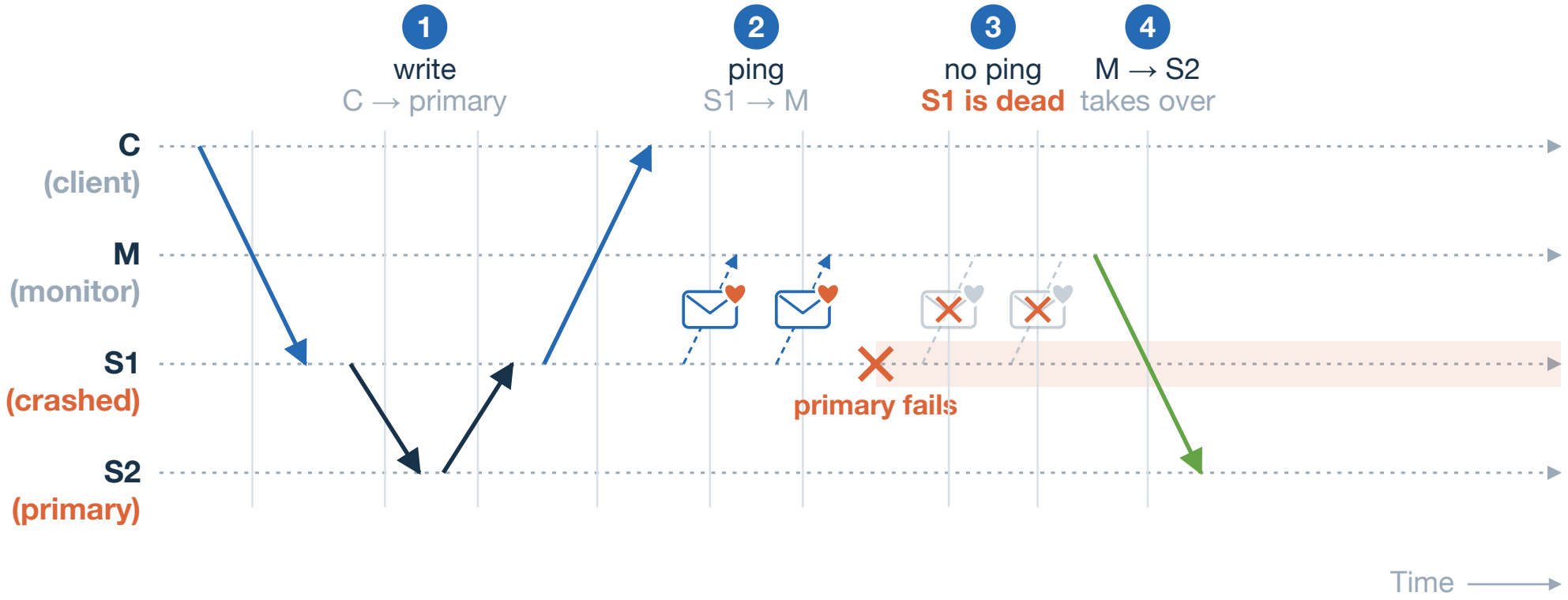
Primary-backup model with availability: the first try



Primary-backup model with availability: the first try

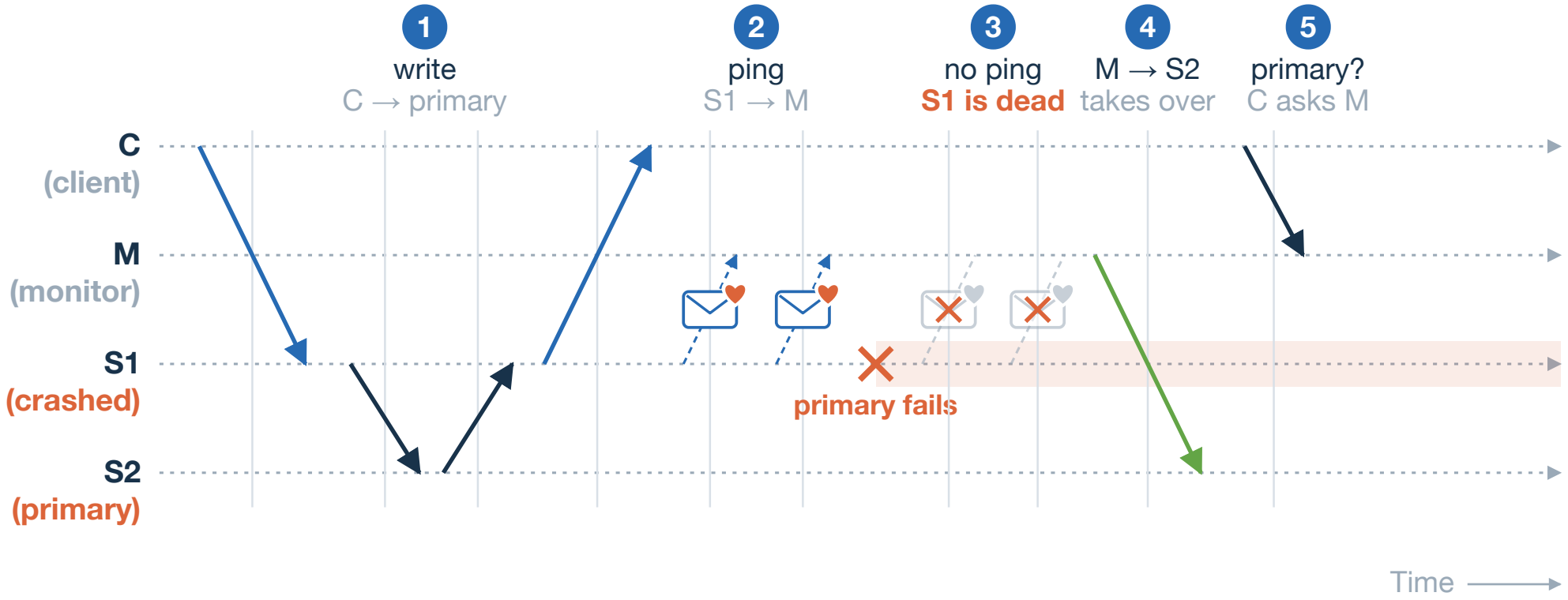


Primary-backup model with availability: the first try

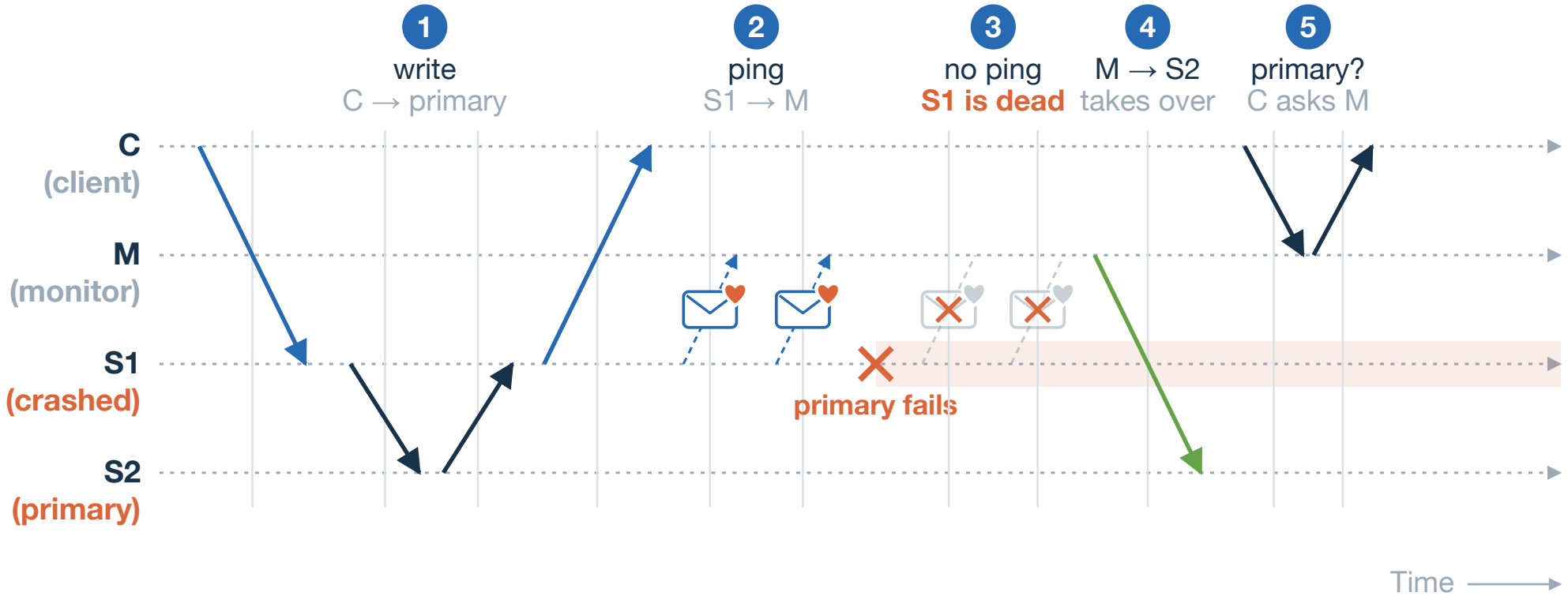


attempt: M watches the primary with heartbeats and appoints the next one — it is not on the write path

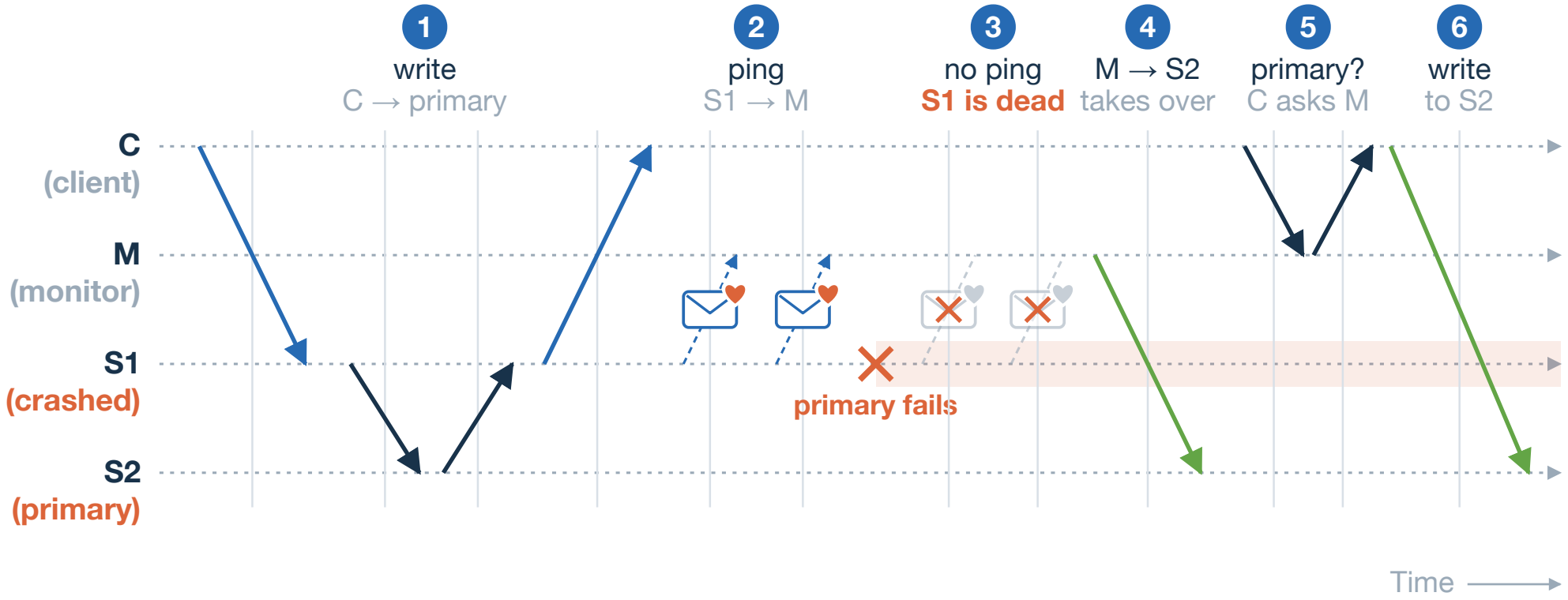
Primary-backup model with availability: the first try



Primary-backup model with availability: the first try

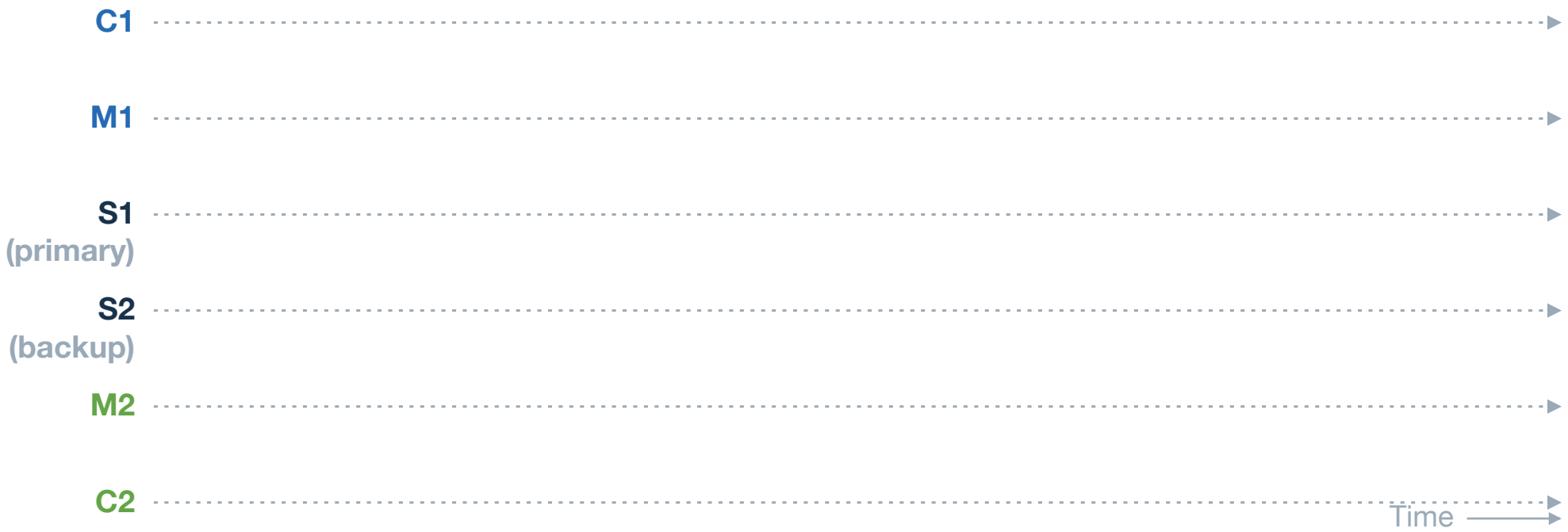


Primary-backup model with availability: the first try

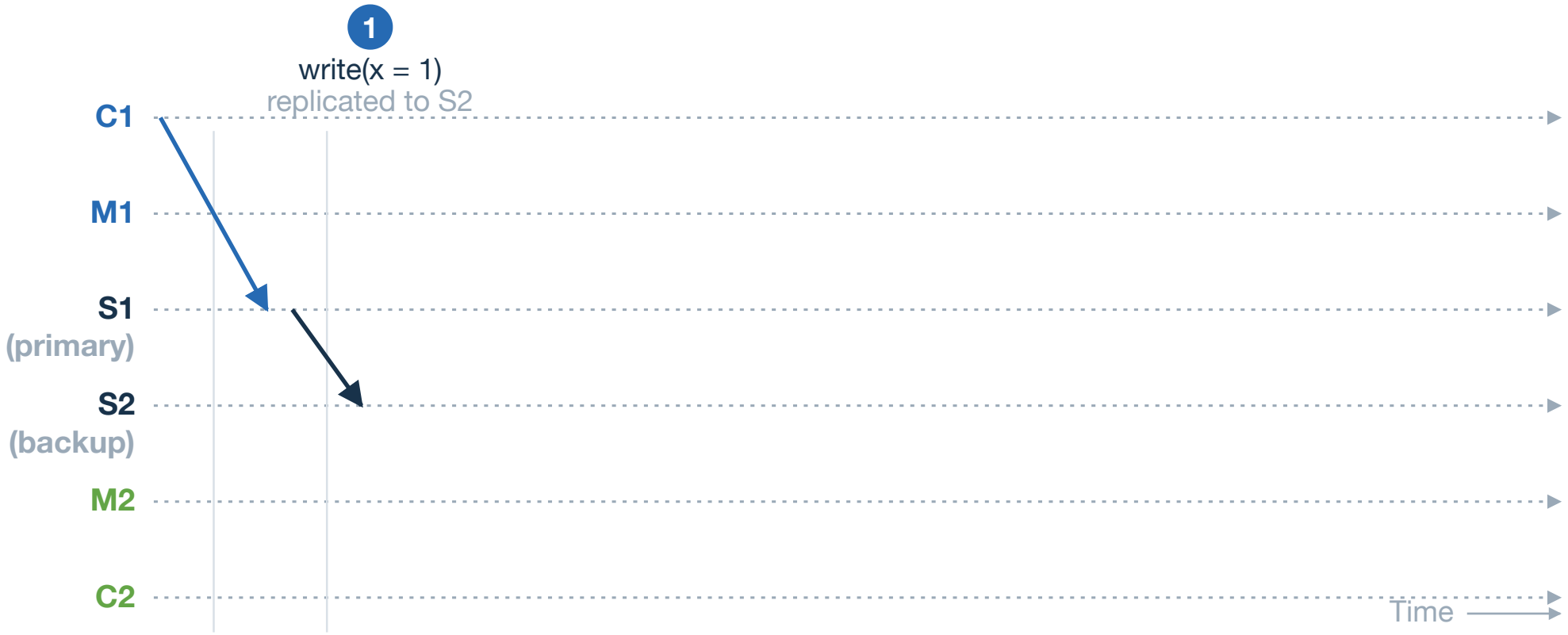


attempt: M watches the primary with heartbeats
and appoints the next one — it is not on normal path execution

Multiple monitors under a network partition



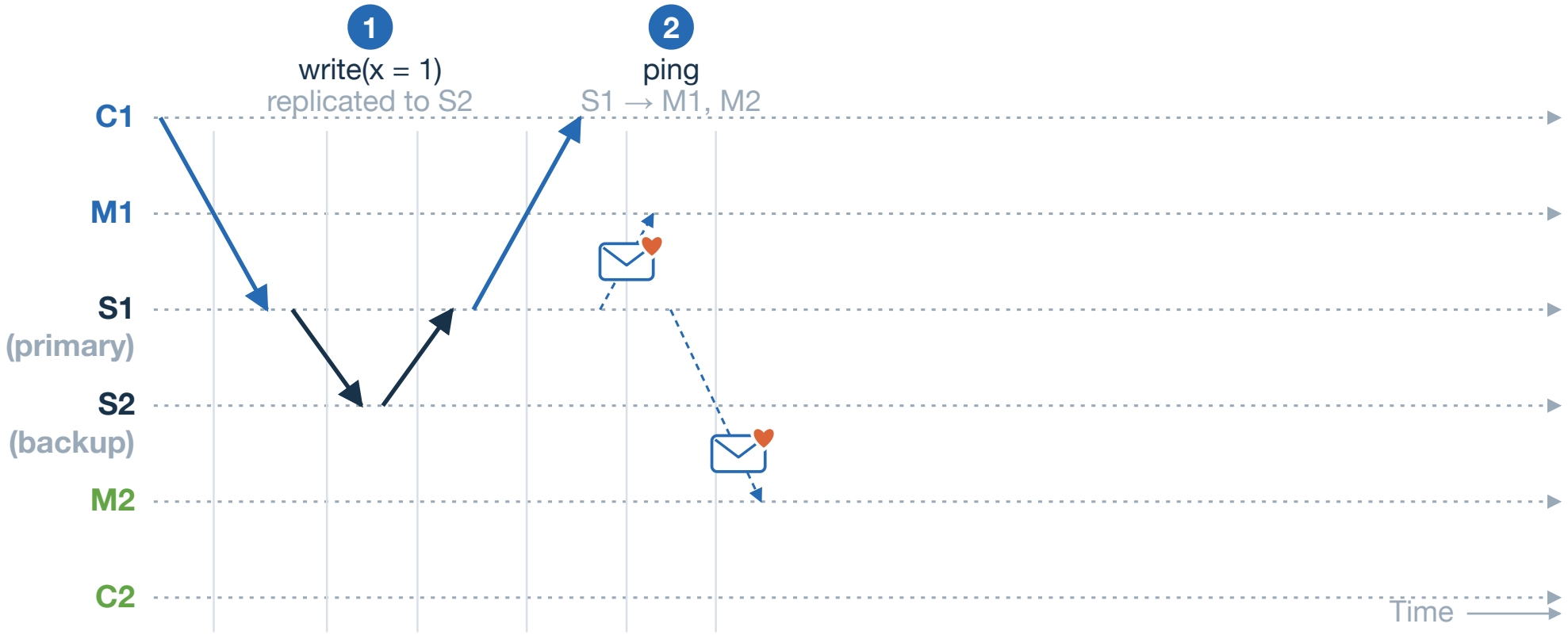
Multiple monitors under a network partition



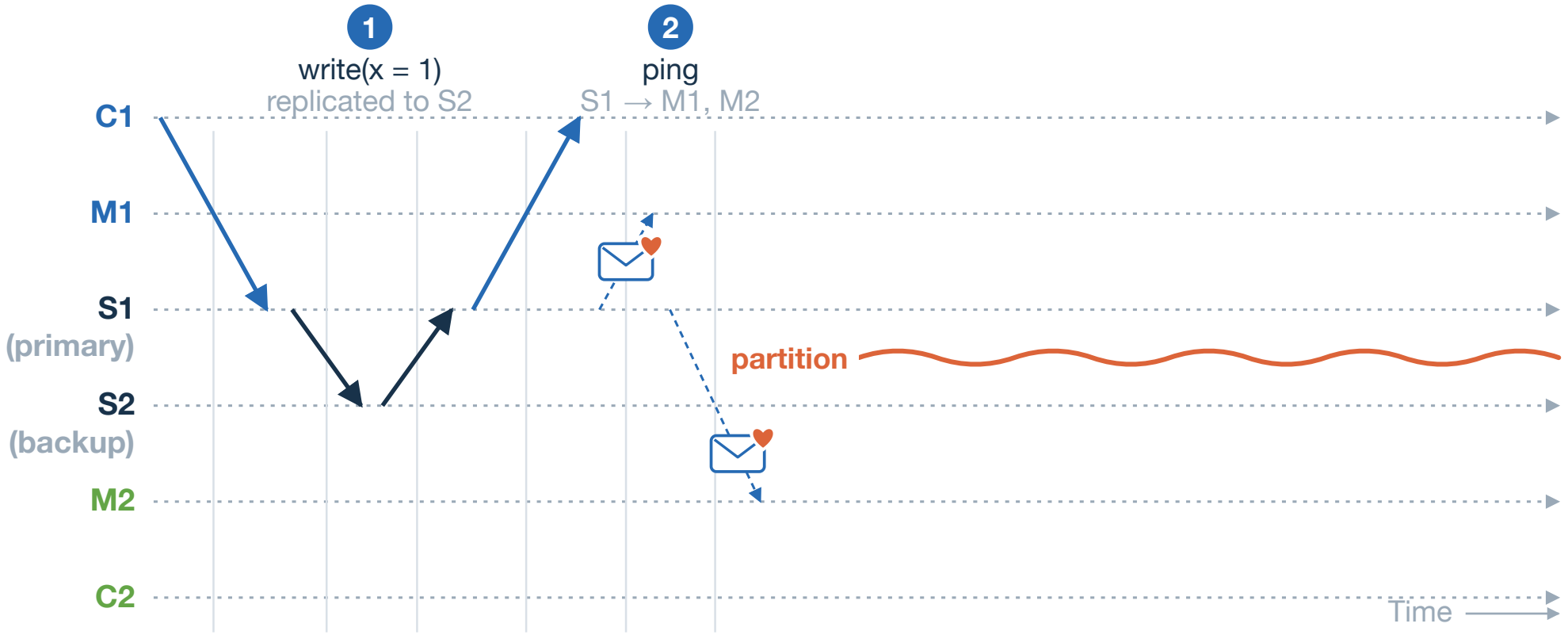
Multiple monitors under a network partition



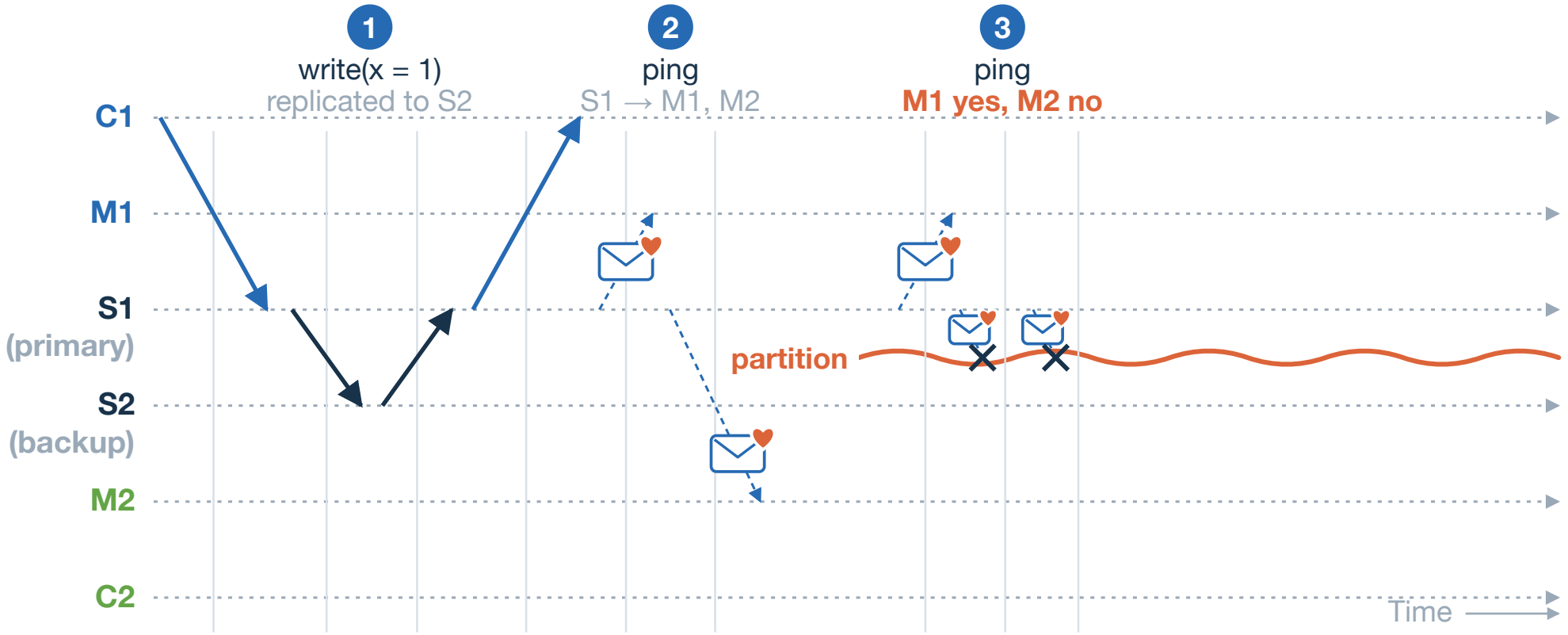
Multiple monitors under a network partition



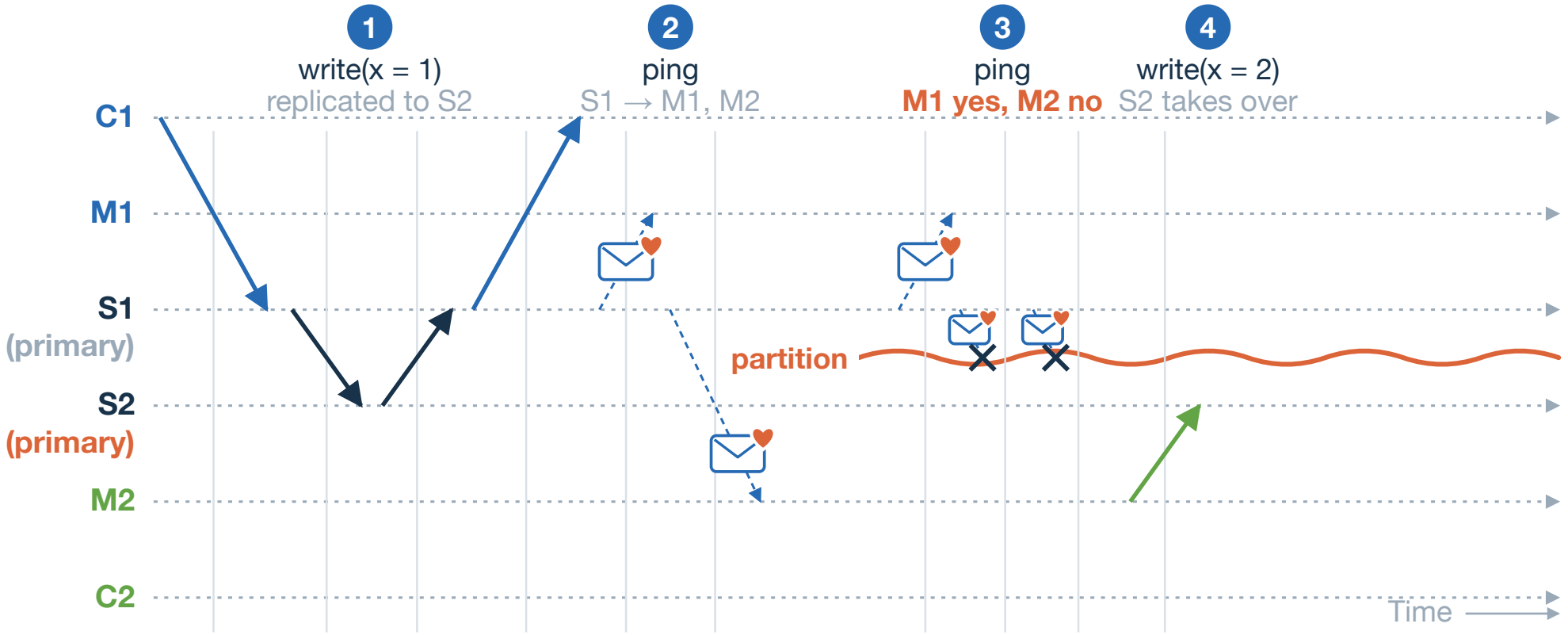
Multiple monitors under a network partition



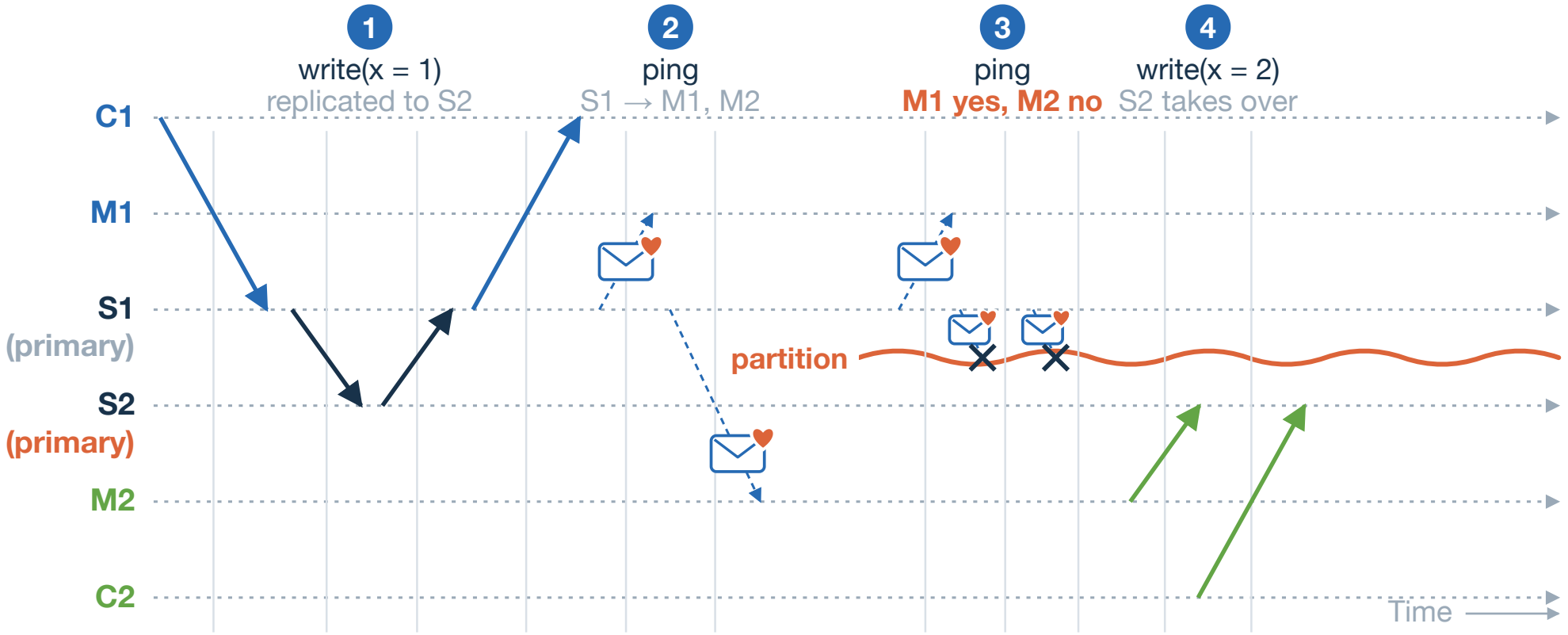
Multiple monitors under a network partition



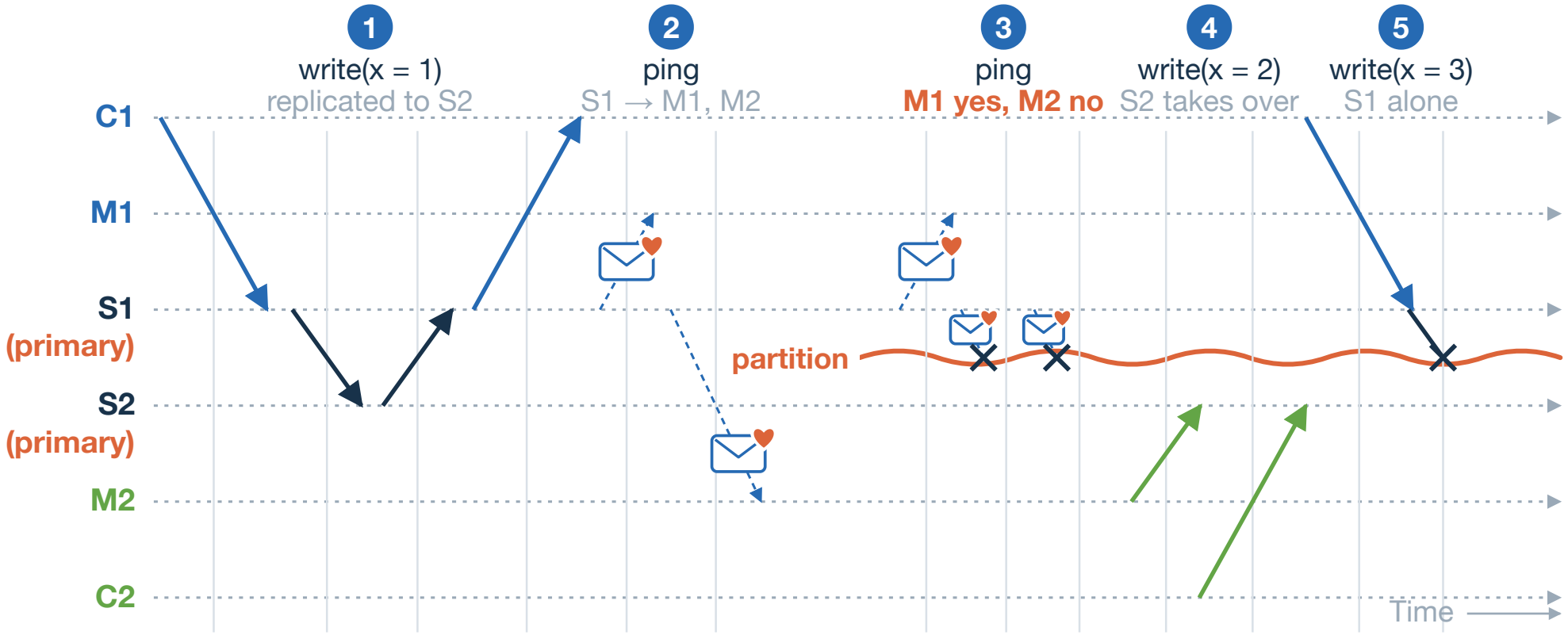
Multiple monitors under a network partition



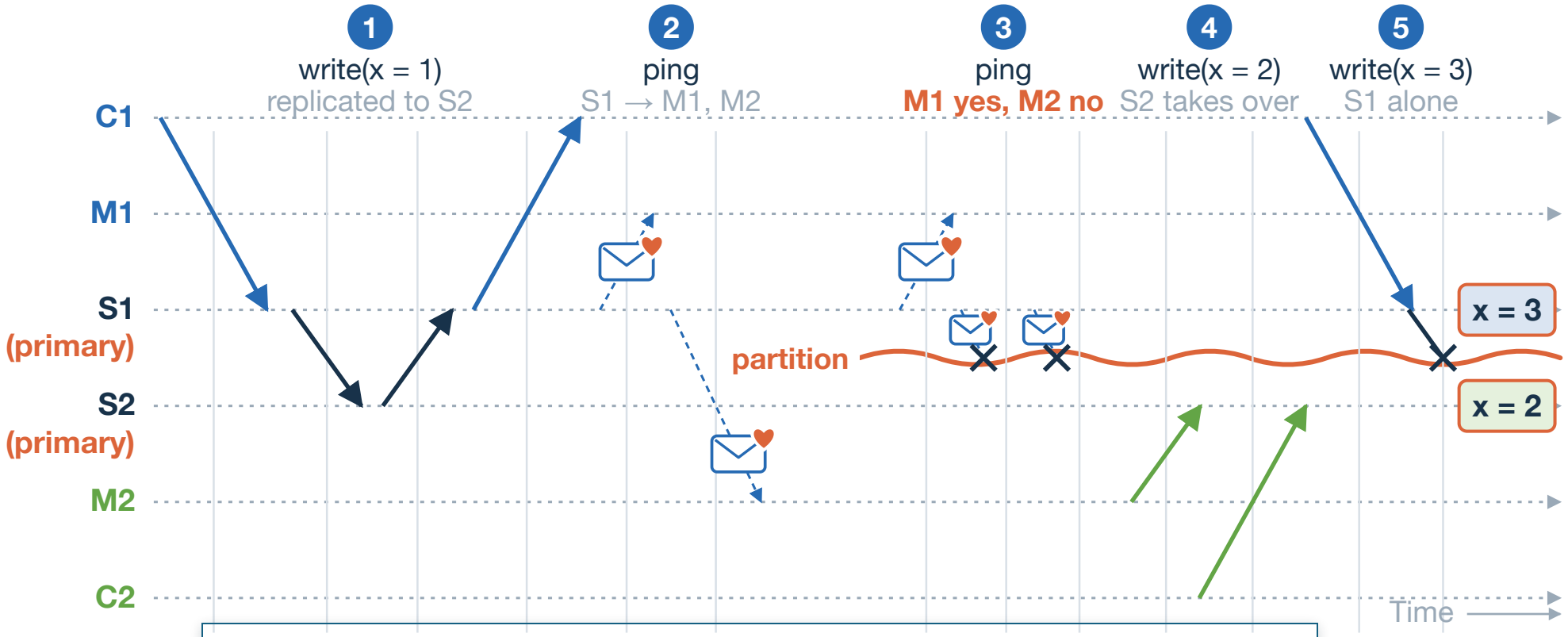
Multiple monitors under a network partition



Multiple monitors under a network partition



Multiple monitors under a network partition



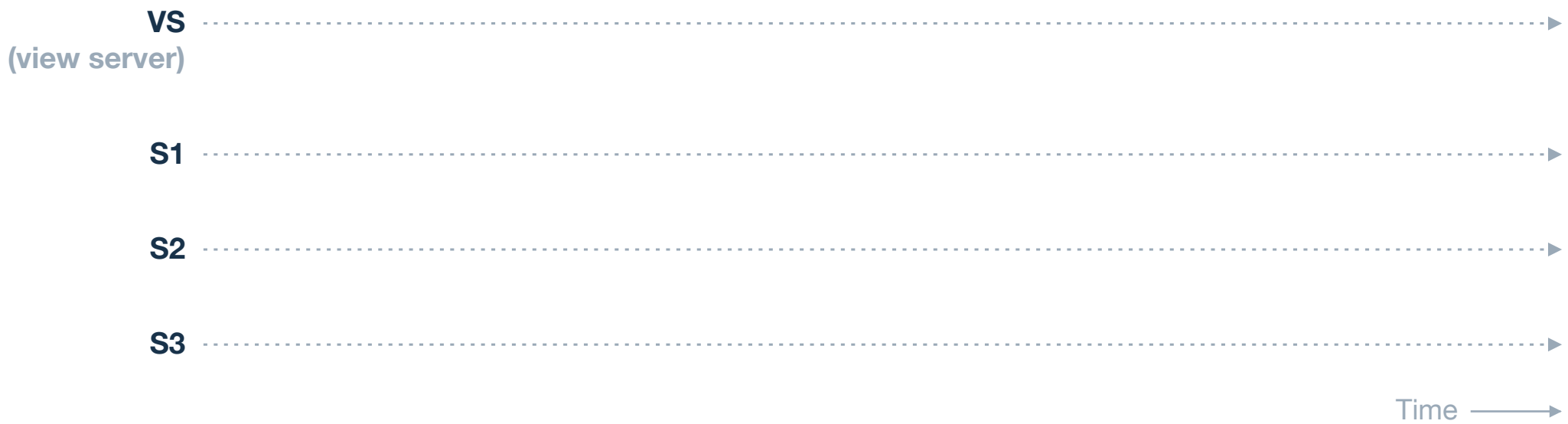
M1 and M2 appointed different primaries;
S1 and S2 are no longer consistent

View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary



View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary



View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary

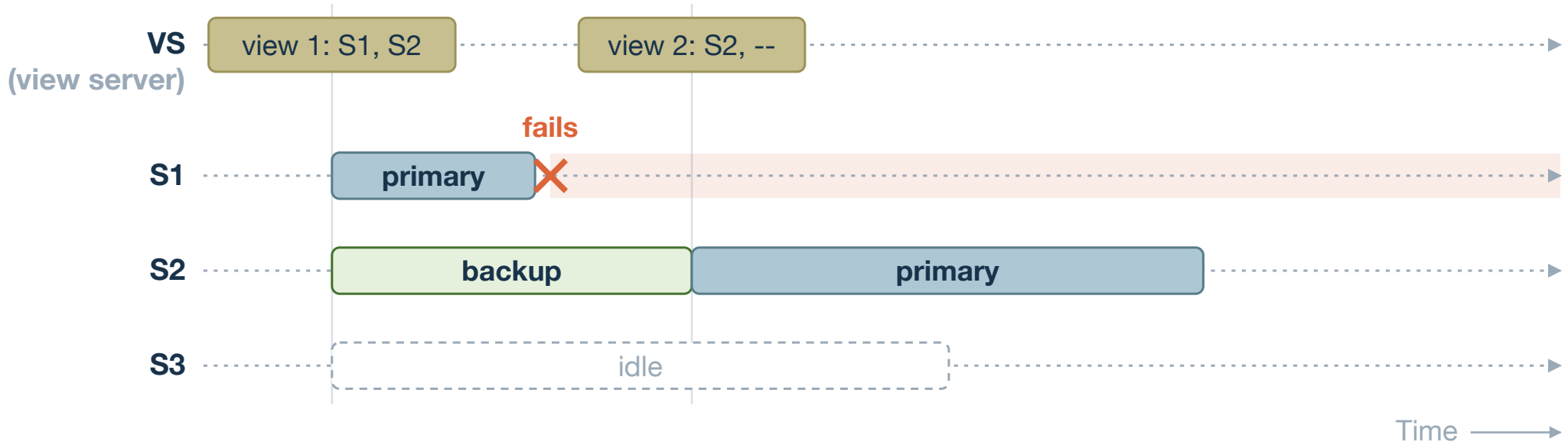


View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary

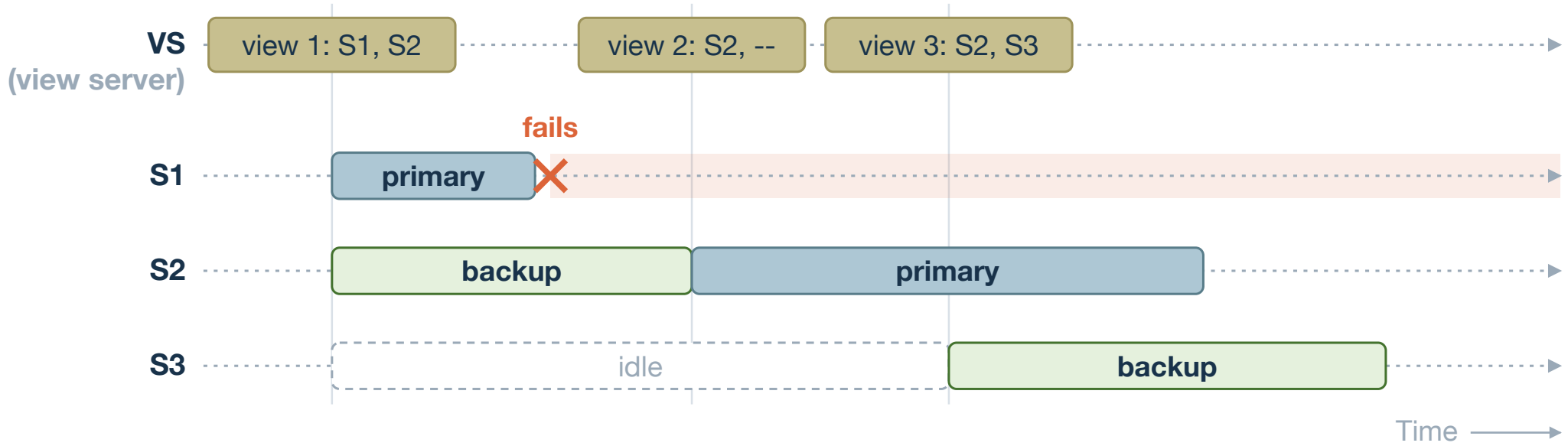


View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary

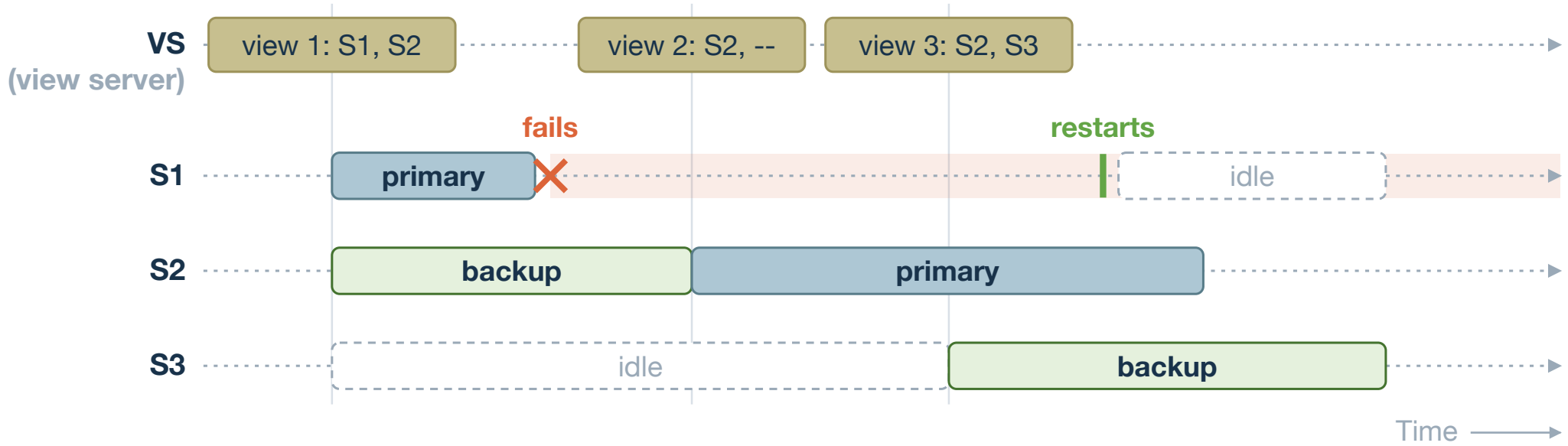


View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary

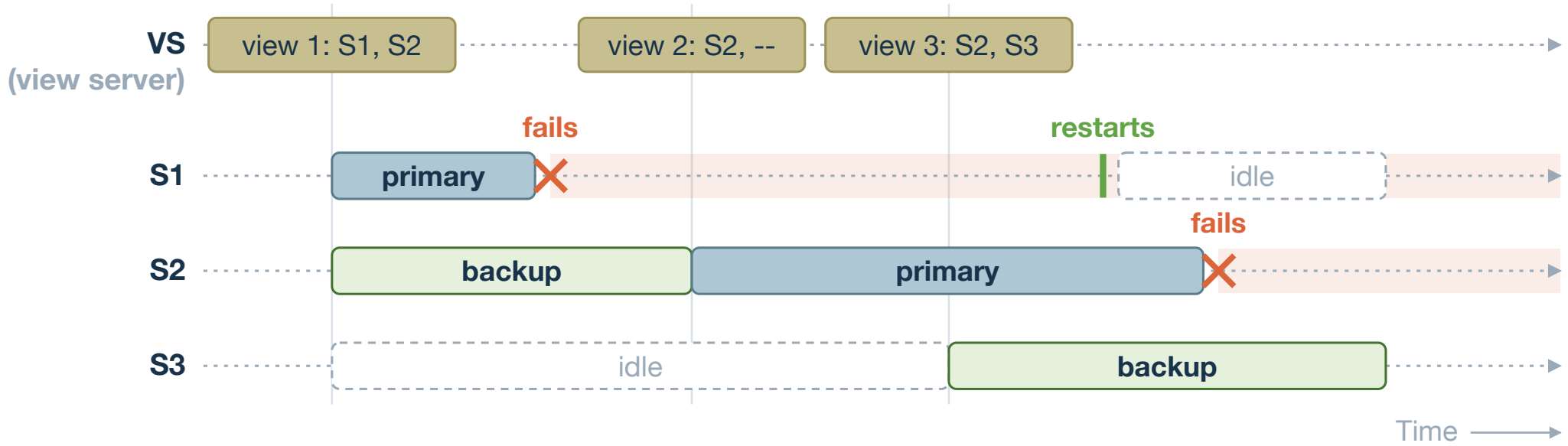


View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

Clients ask the view server who is primary, cache it, then contact the primary

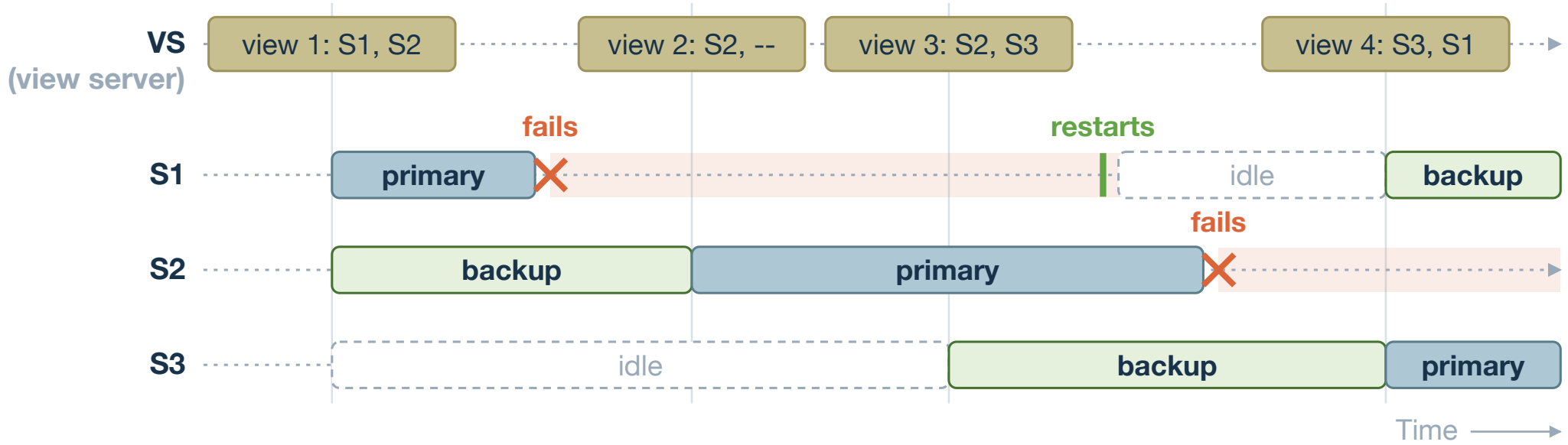


View Server: one server to decide who is the primary

The view server keeps a table that maintains a sequence of "view" (One-writer rule)

- Each view contains view number, primary server, and backup servers

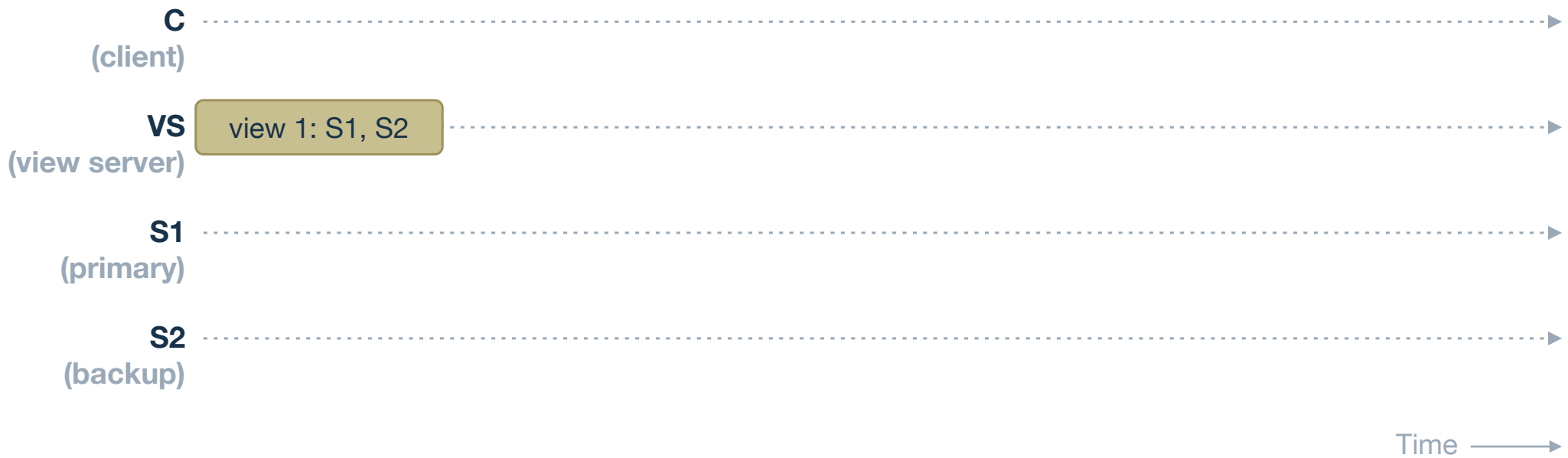
Clients ask the view server who is primary, cache it, then contact the primary



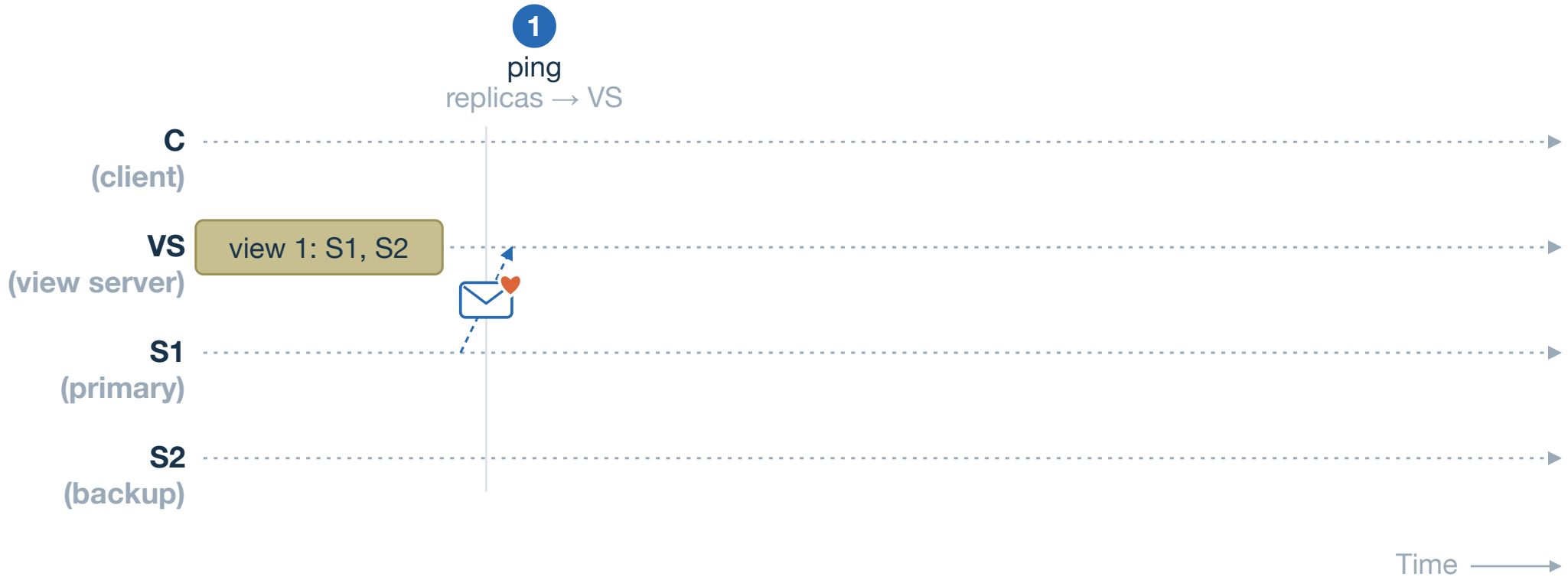
Only the view server can decide who is the primary now

Question: can the view server prevent two primaries from serving at the same time?

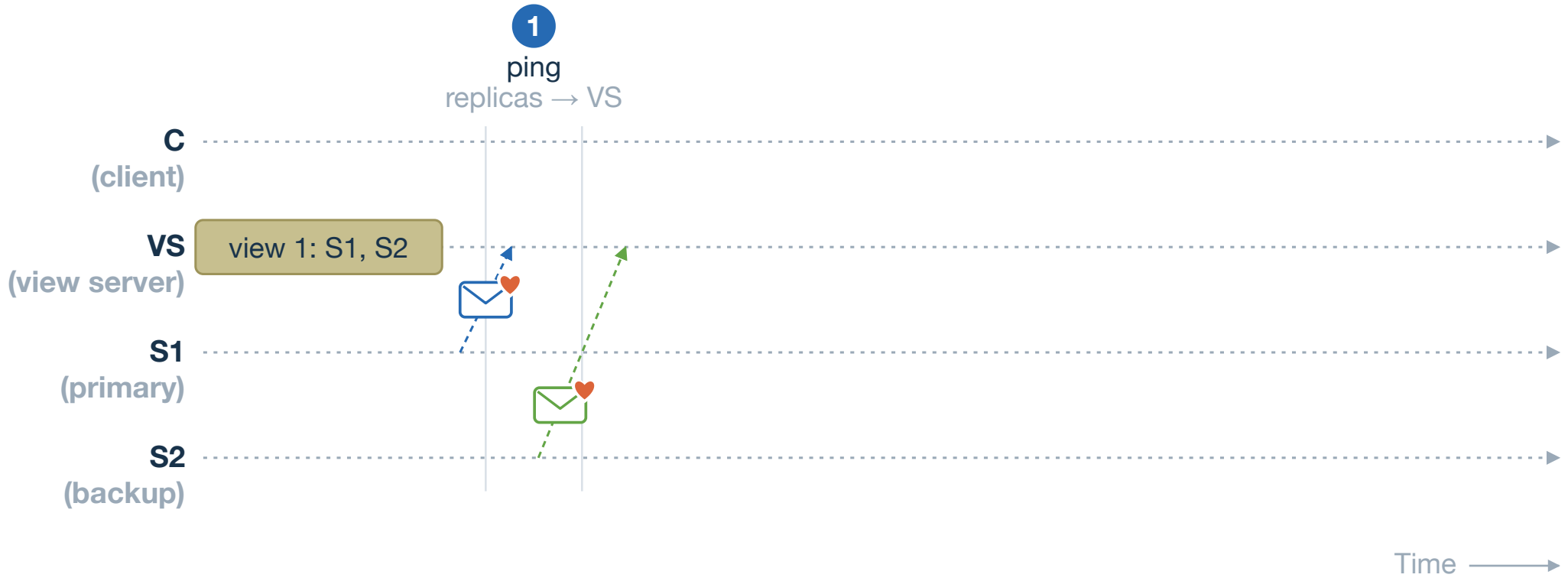
Question: can two servers simultaneously become primaries?



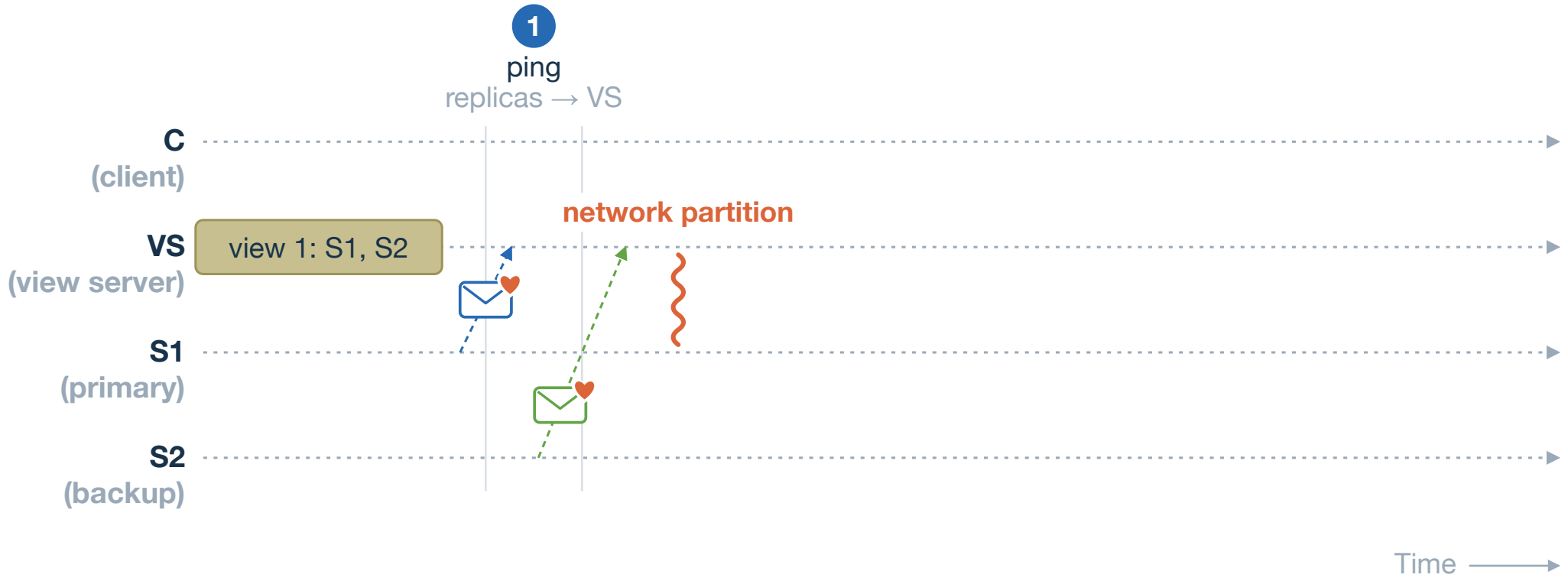
Question: can two servers simultaneously become primaries?



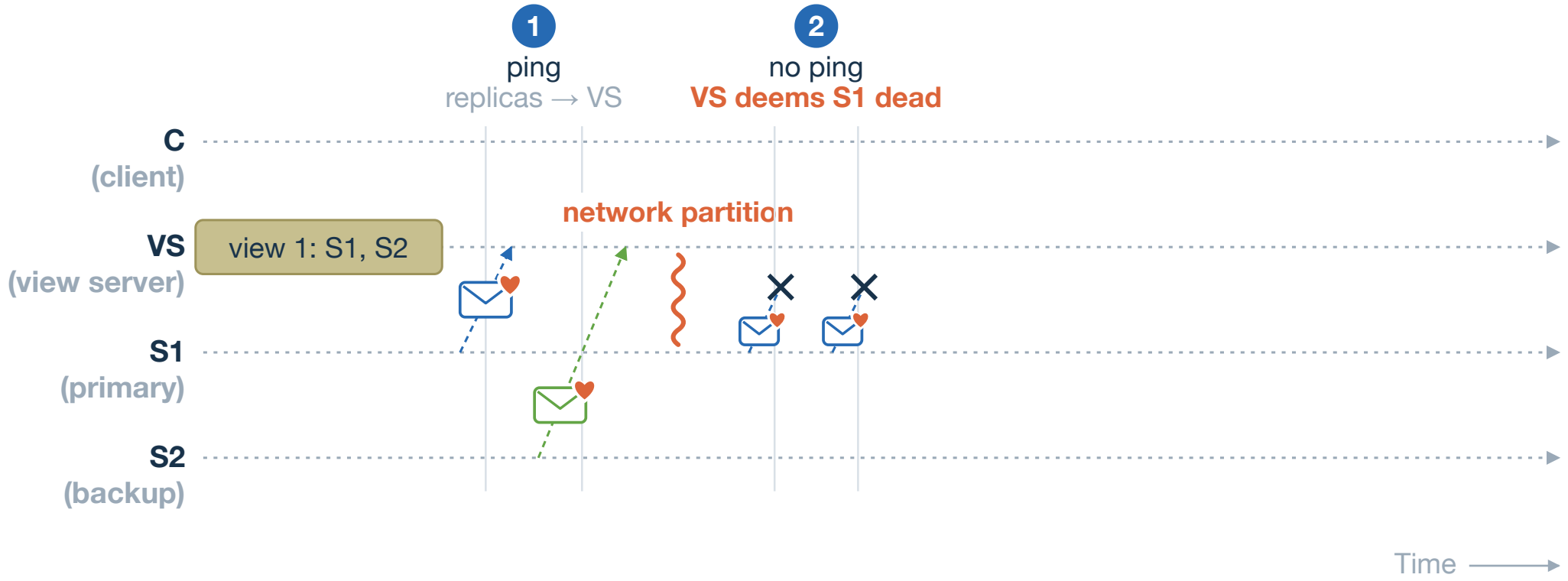
Question: can two servers simultaneously become primaries?



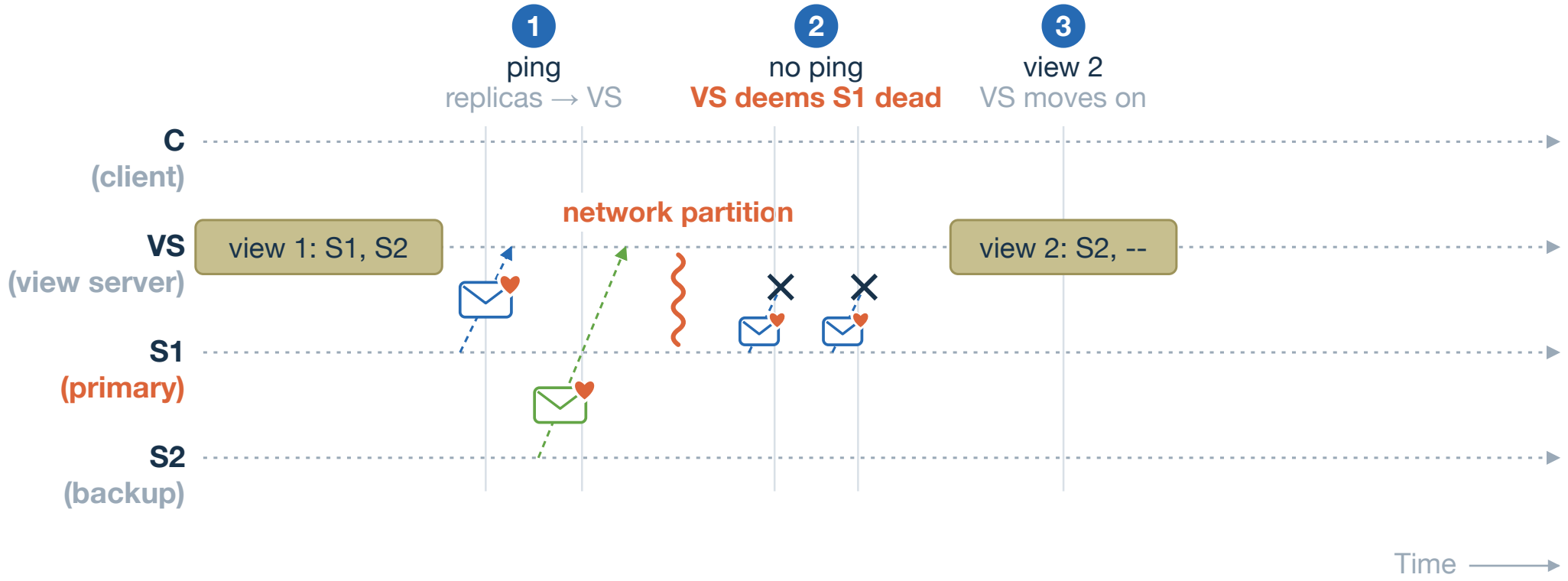
Question: can two servers simultaneously become primaries?



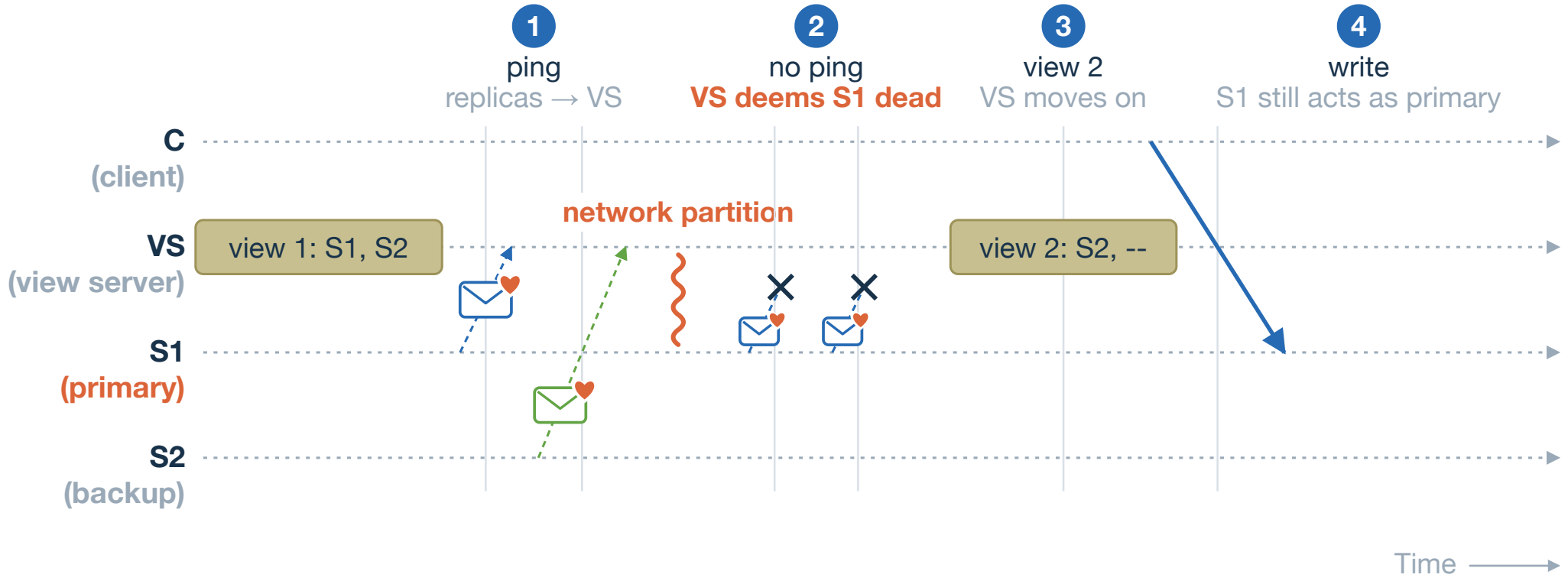
Question: can two servers simultaneously become primaries?



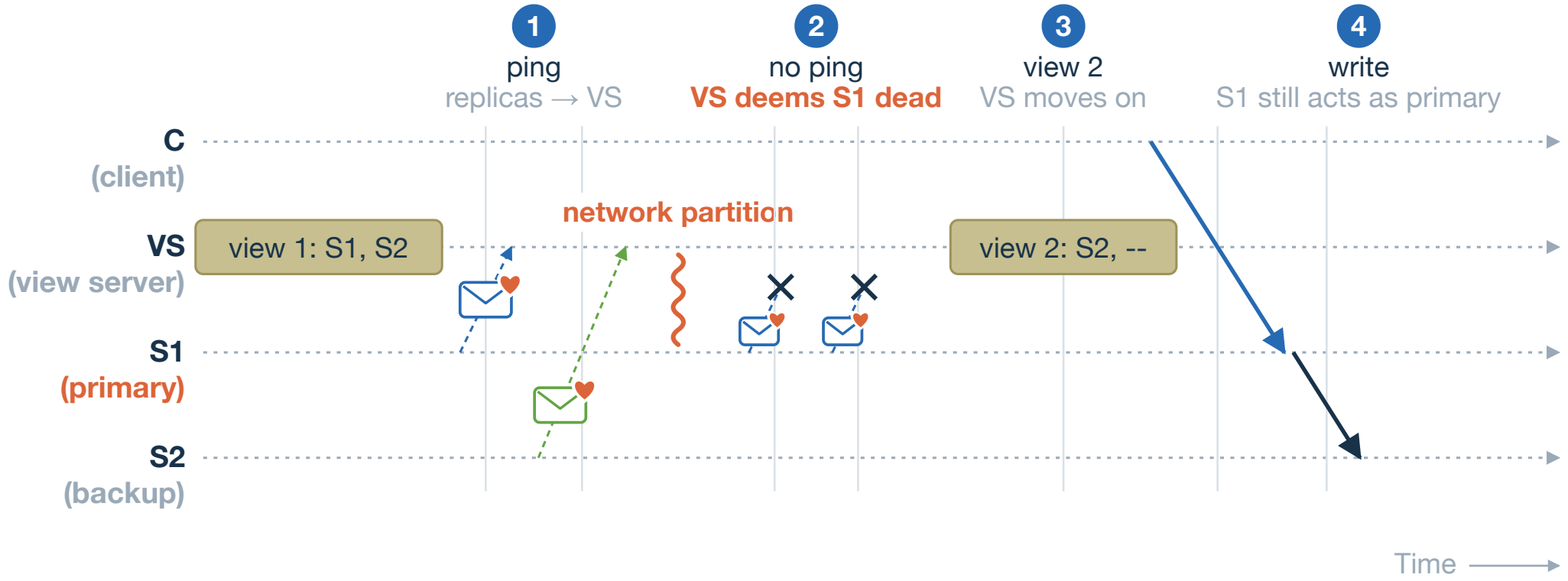
Question: can two servers simultaneously become primaries?



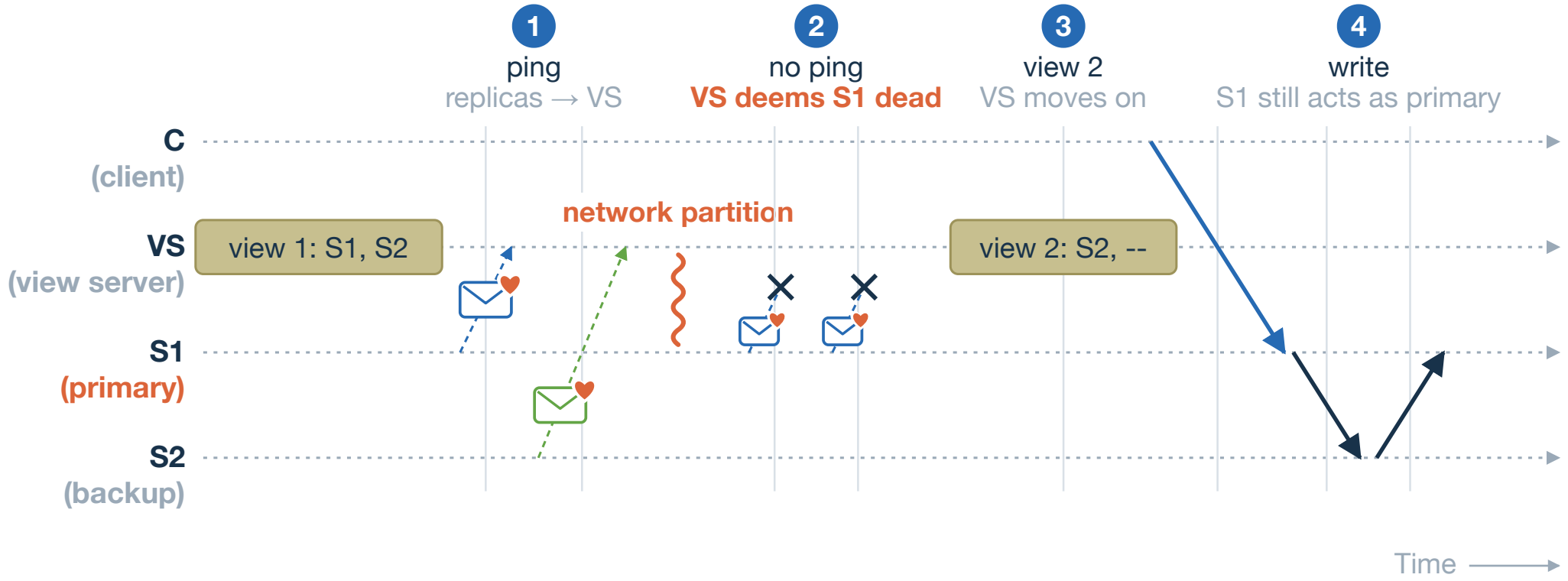
Question: can two servers simultaneously become primaries?



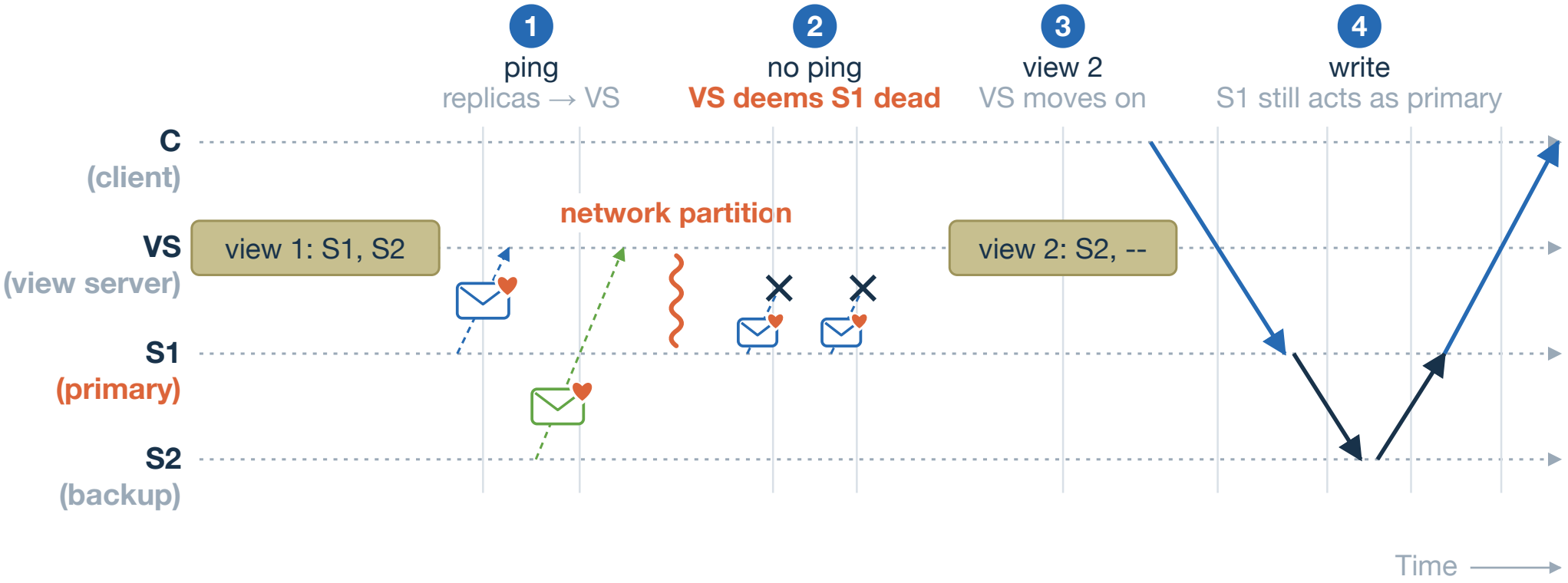
Question: can two servers simultaneously become primaries?



Question: can two servers simultaneously become primaries?



Question: can two servers simultaneously become primaries?



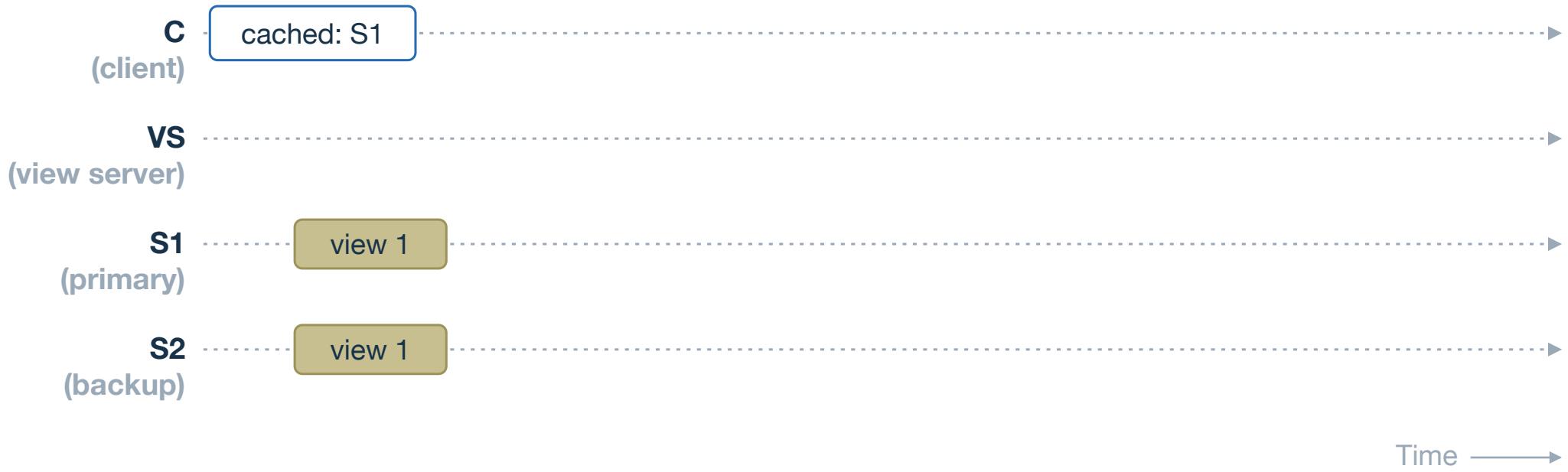
Yes, it can happen. So how to prevent split brain?

Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

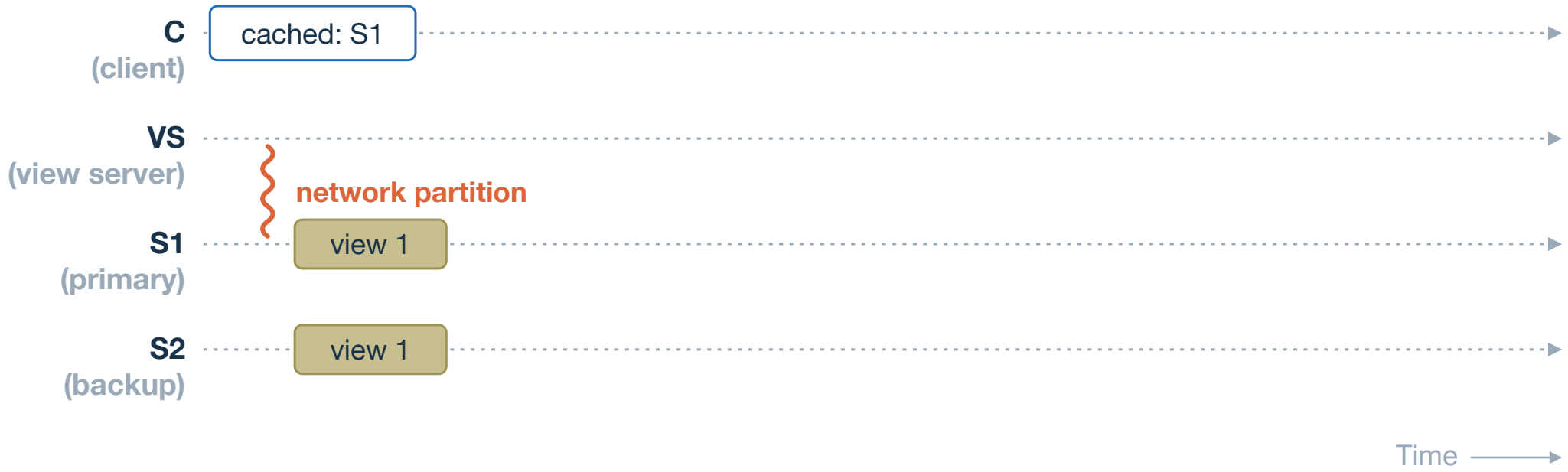


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests



Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

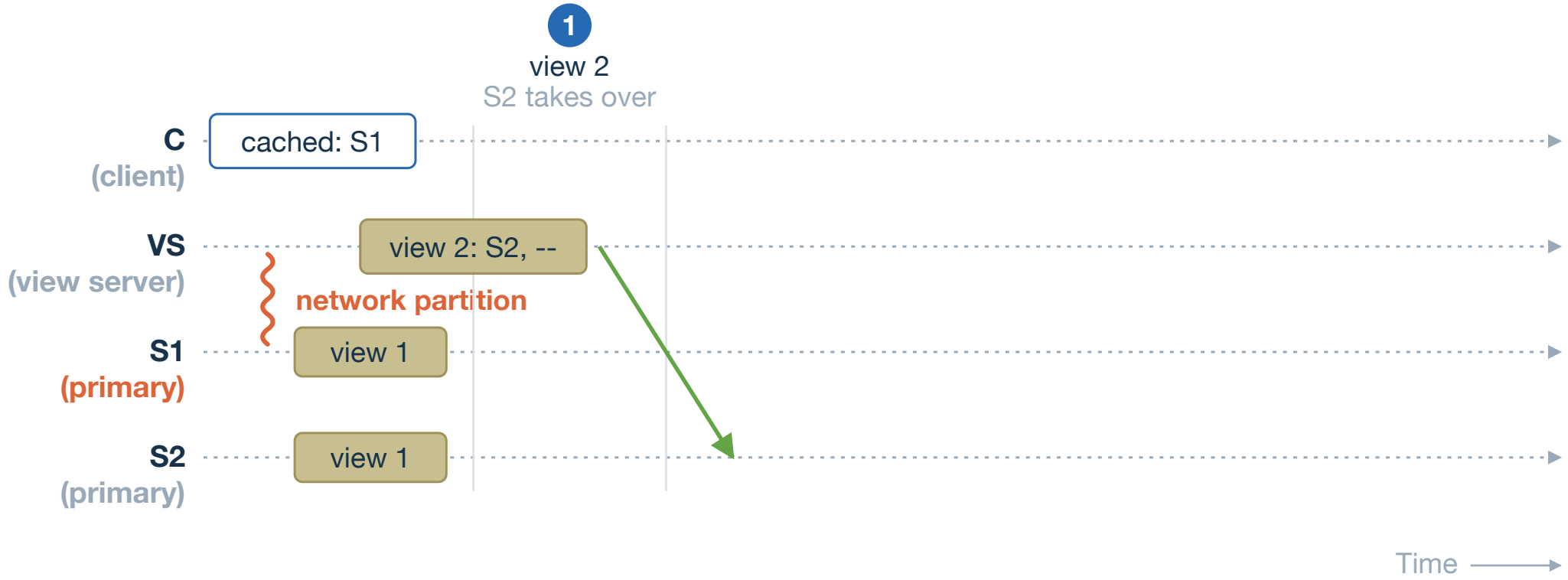


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

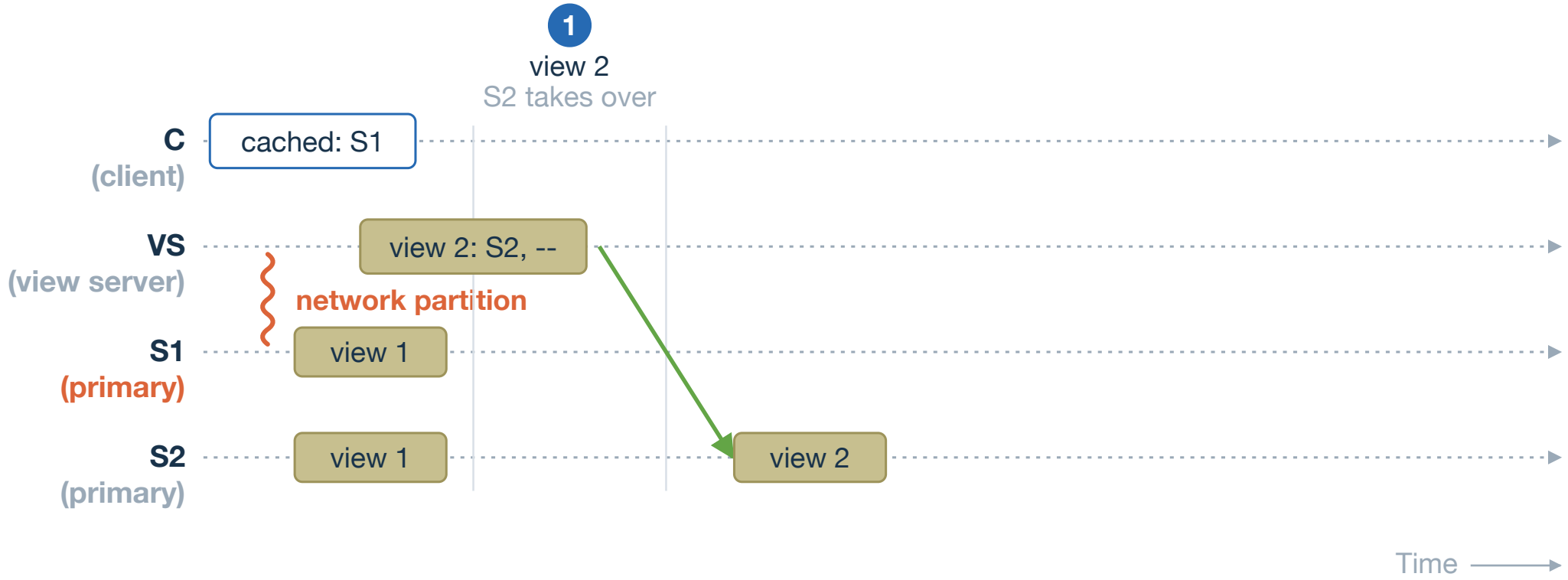


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

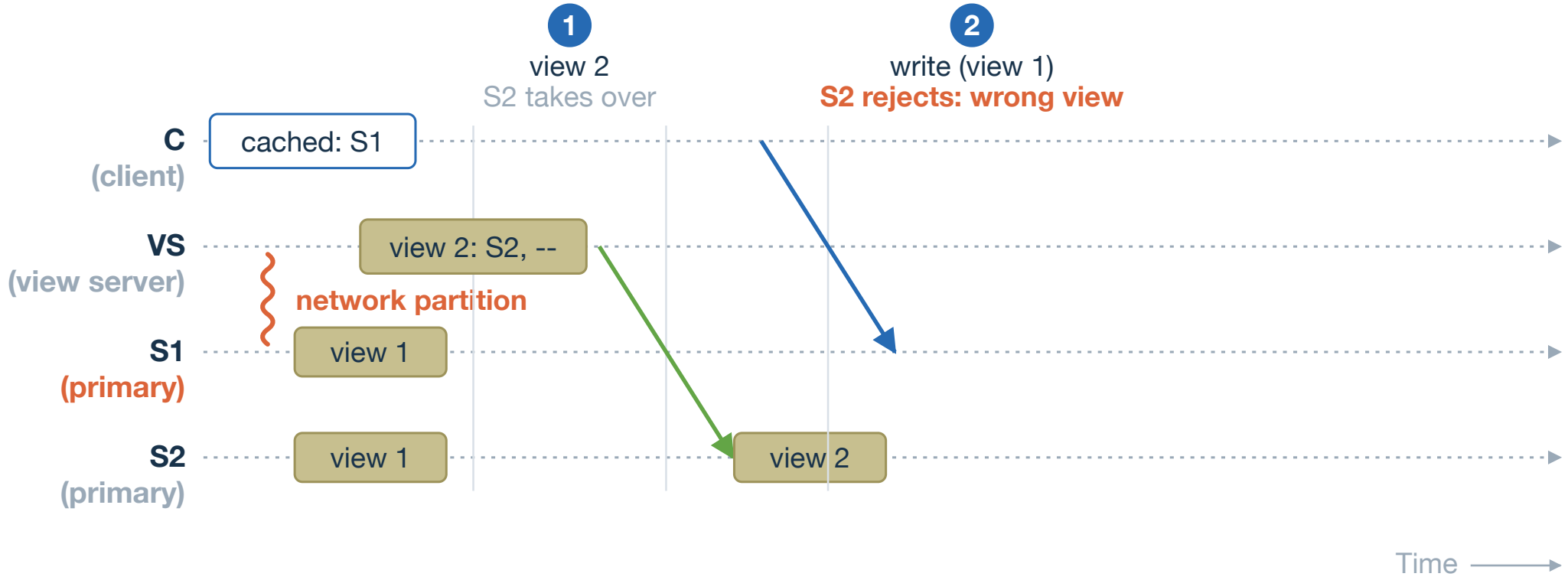


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

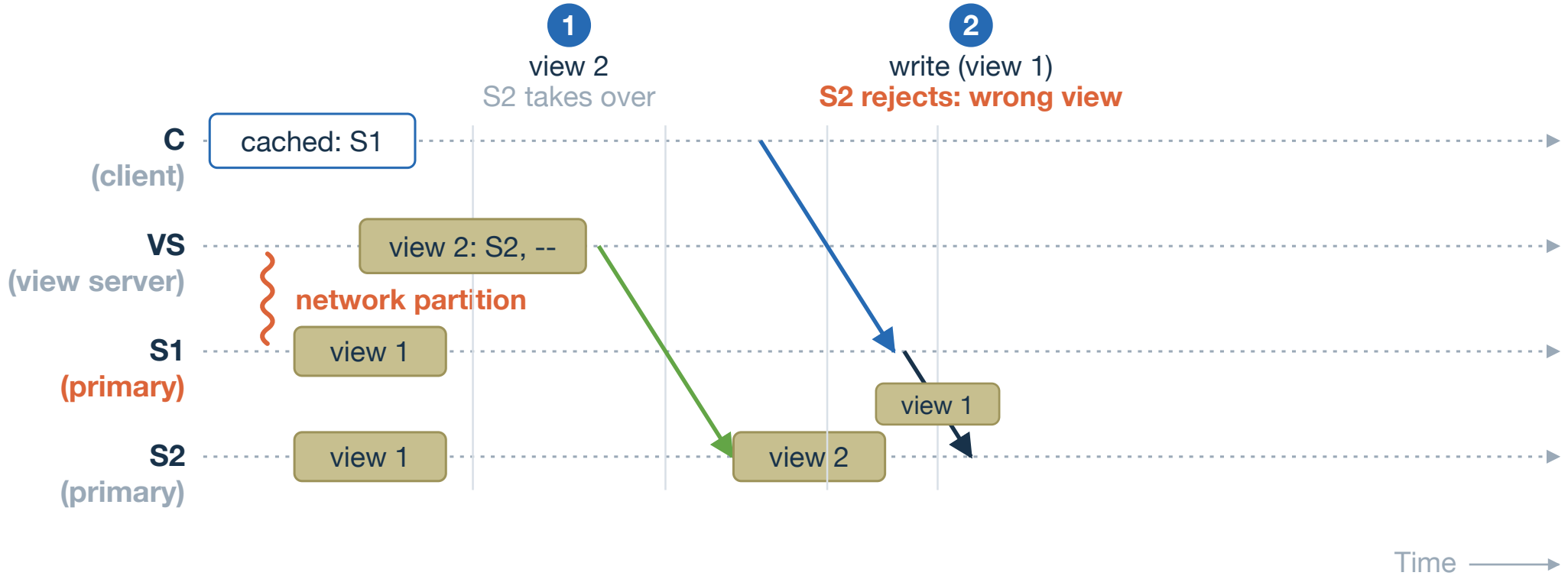


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

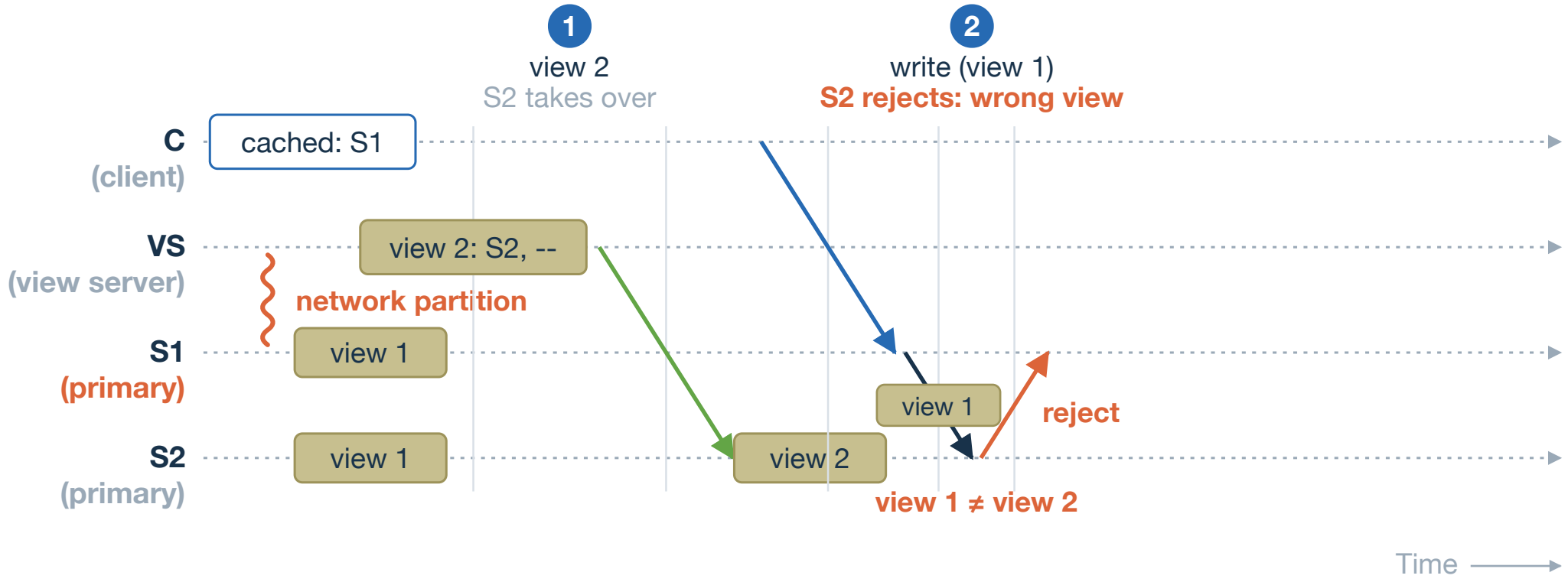


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

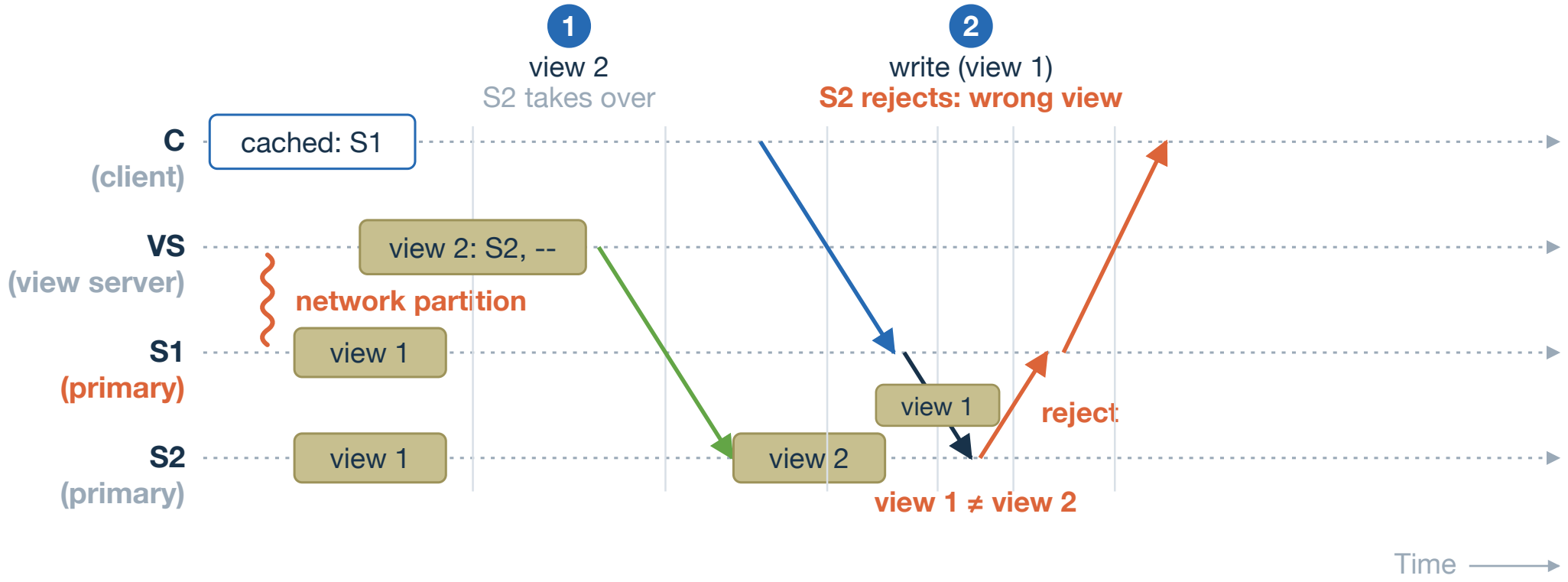


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

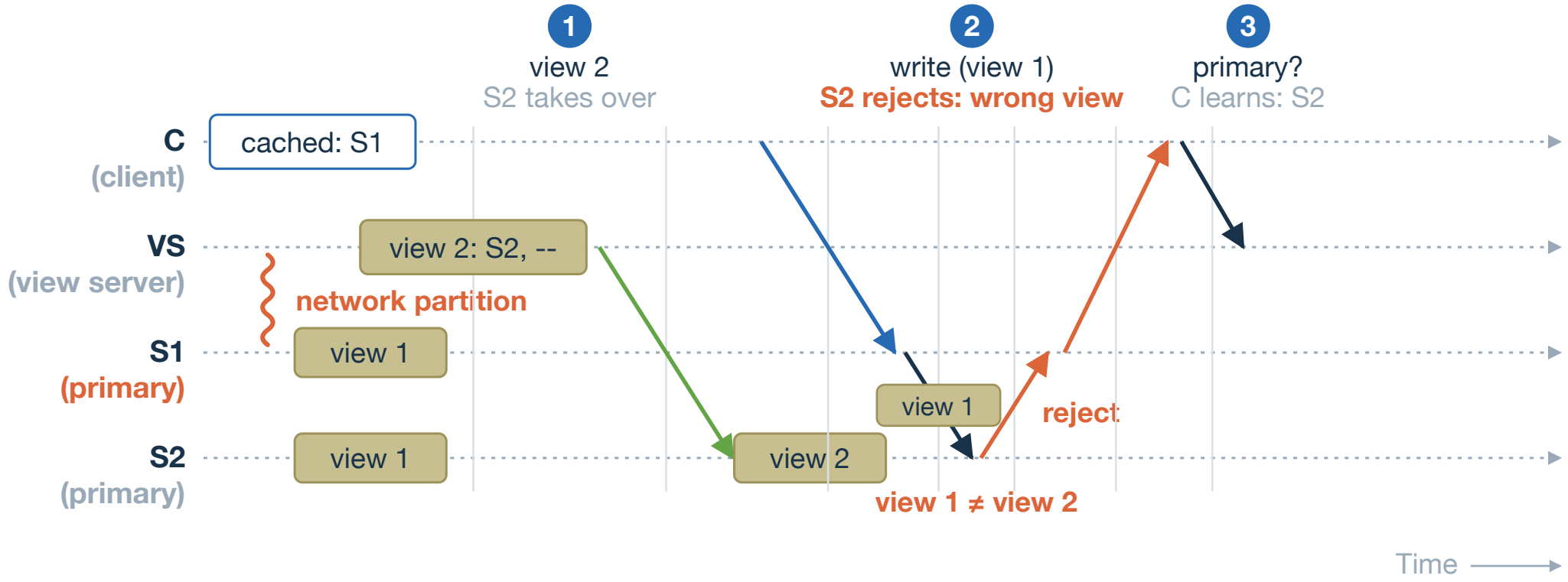


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

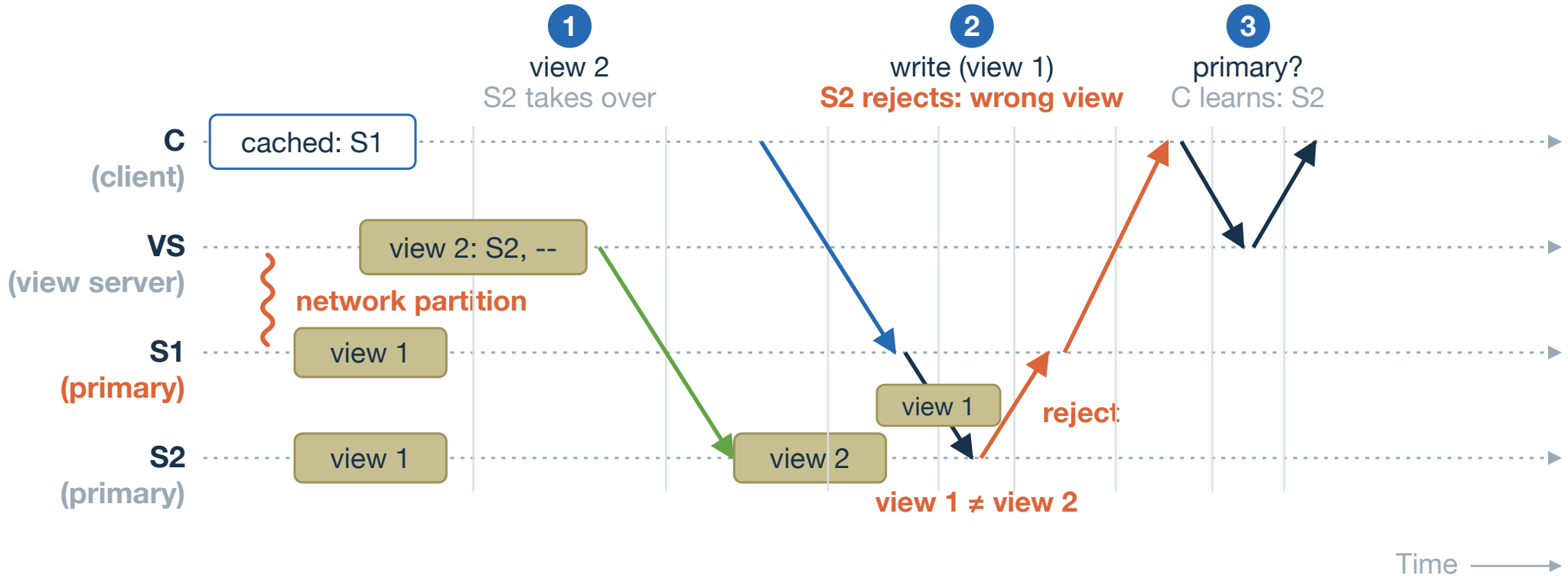


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

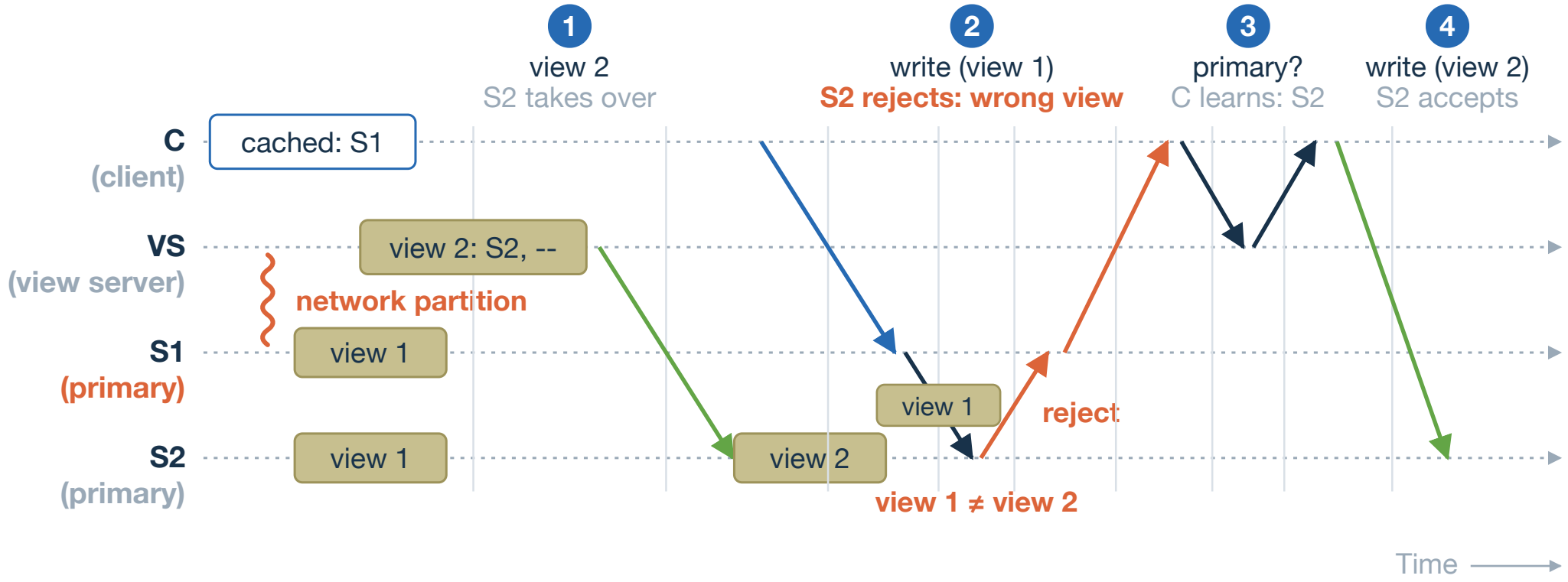


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests

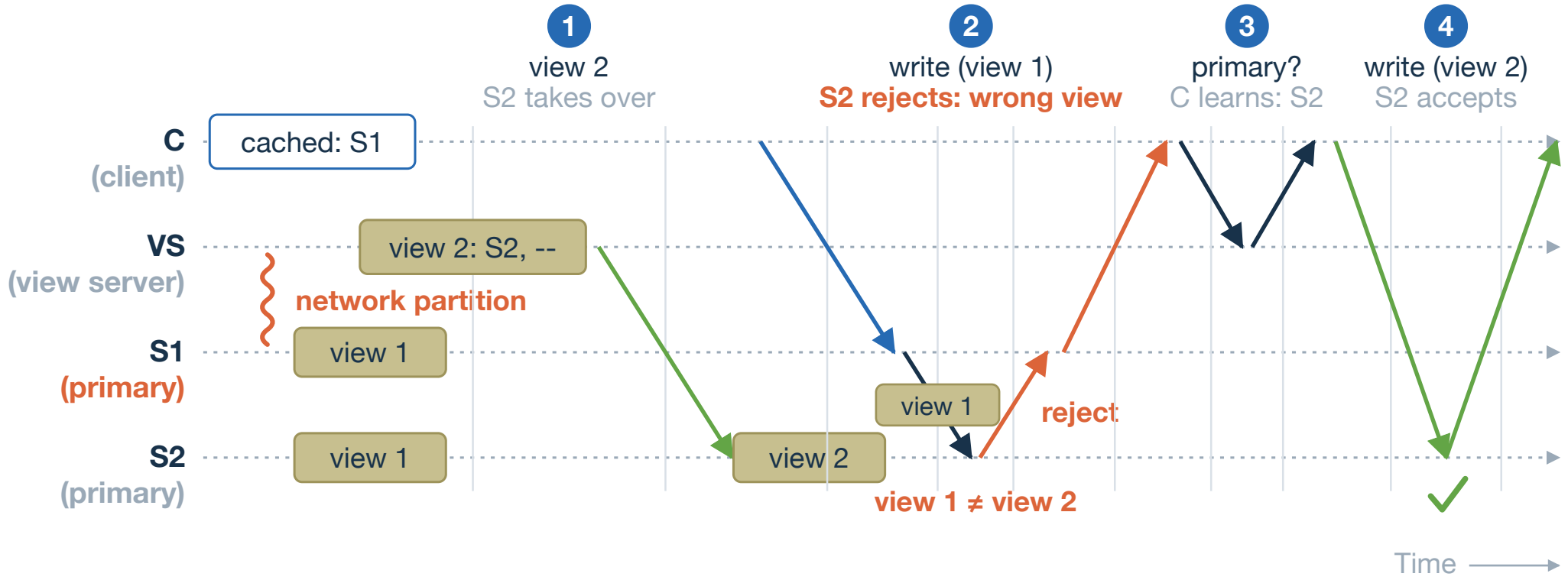


Rules for handling two live primaries

Primary must wait for backups to accept each request

- A backup accepts a request if it is in the same view of the primary

Non-primary must reject direct client requests



Consider S1 partitioned from the VS

Before S2 hears about view 2:

- S1 can process operations from clients, S2 will accept forwarded requests (because they are all on view 1)
- S2 will reject operations from clients who have heard about view 2

After S2 hears about view 2:

- If S1 receives client requests, it will forward. S2 will reject (not ACK), so S1 can no longer act as primary
- S1 will send an error to the client, the client will ask VS for the new view, learn about view 2, and re-send to S2

The commit point of switch-over:

- When S2 hears about view 2

Question: how many backups must ack before the primary replies?

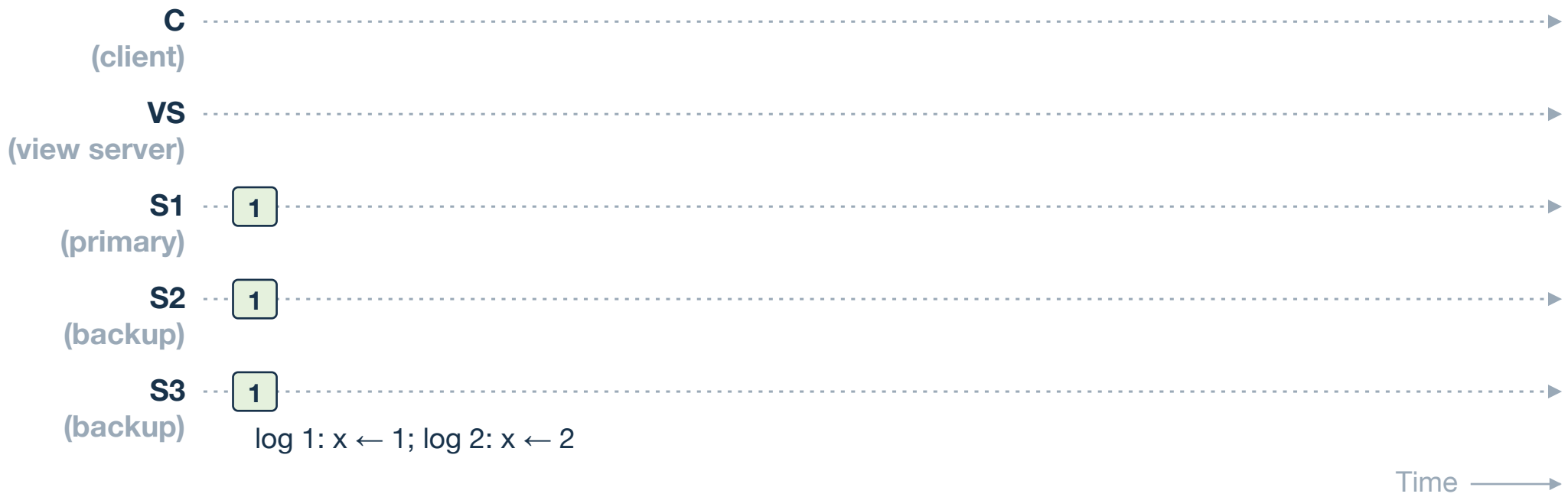
Waiting for all backups sounds bad

- Why? The more replicas there are, the more likely that some backup has failed

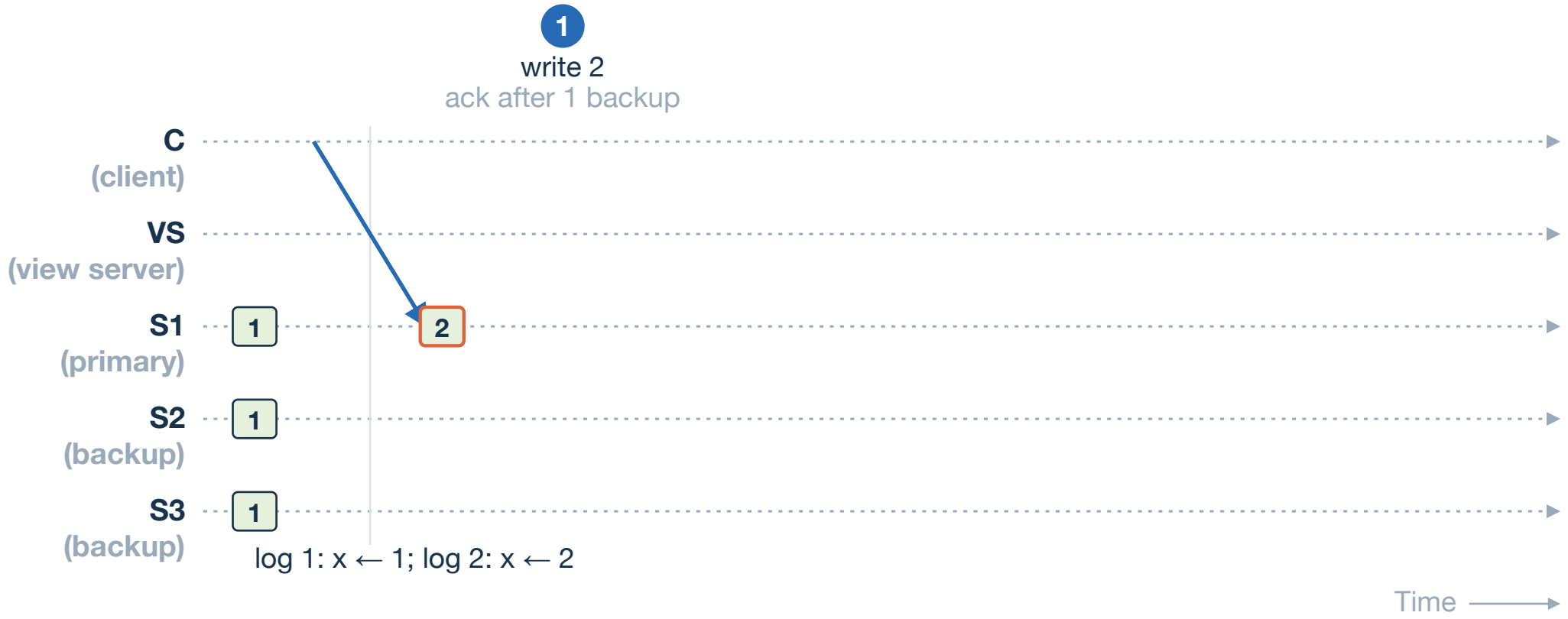
Goal: if the request is acked to the client, the request must persist after the primary & backup failures

To achieve so, we need to enforce some rules on which backup can become the primary

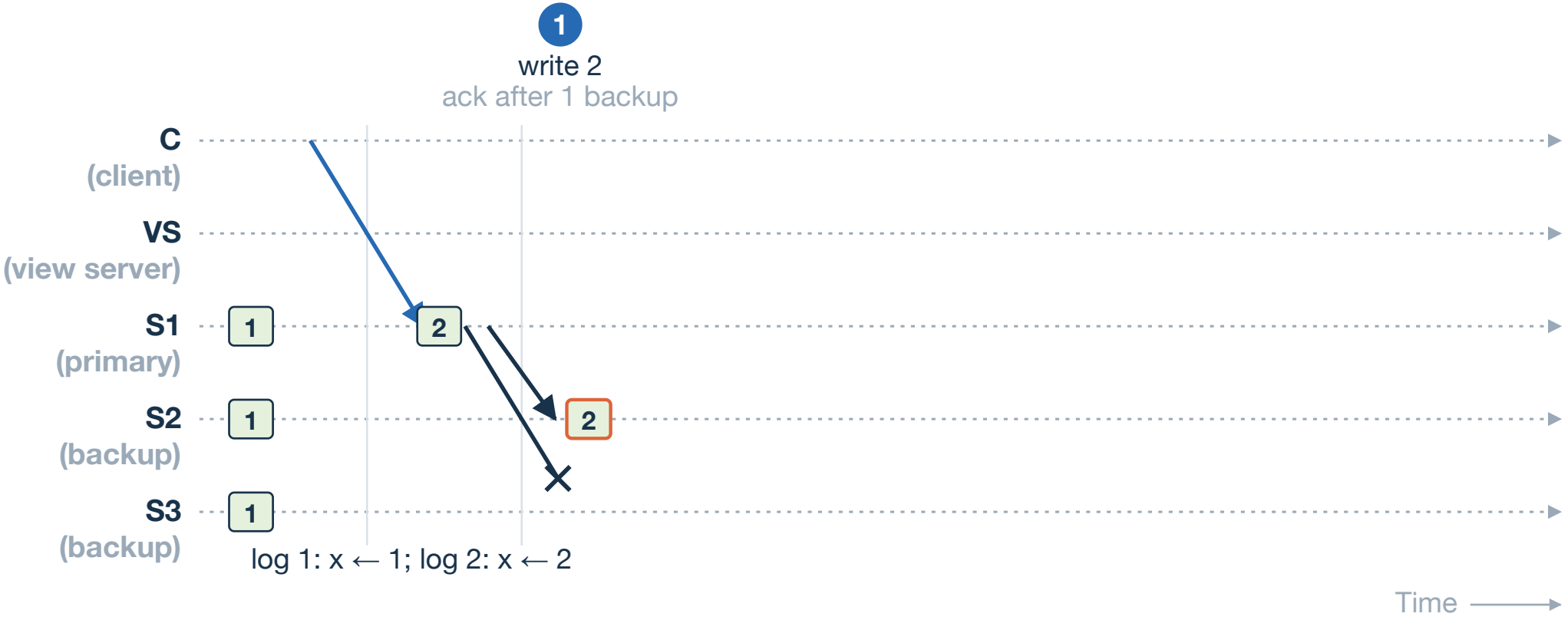
Question: which backup becomes the primary?



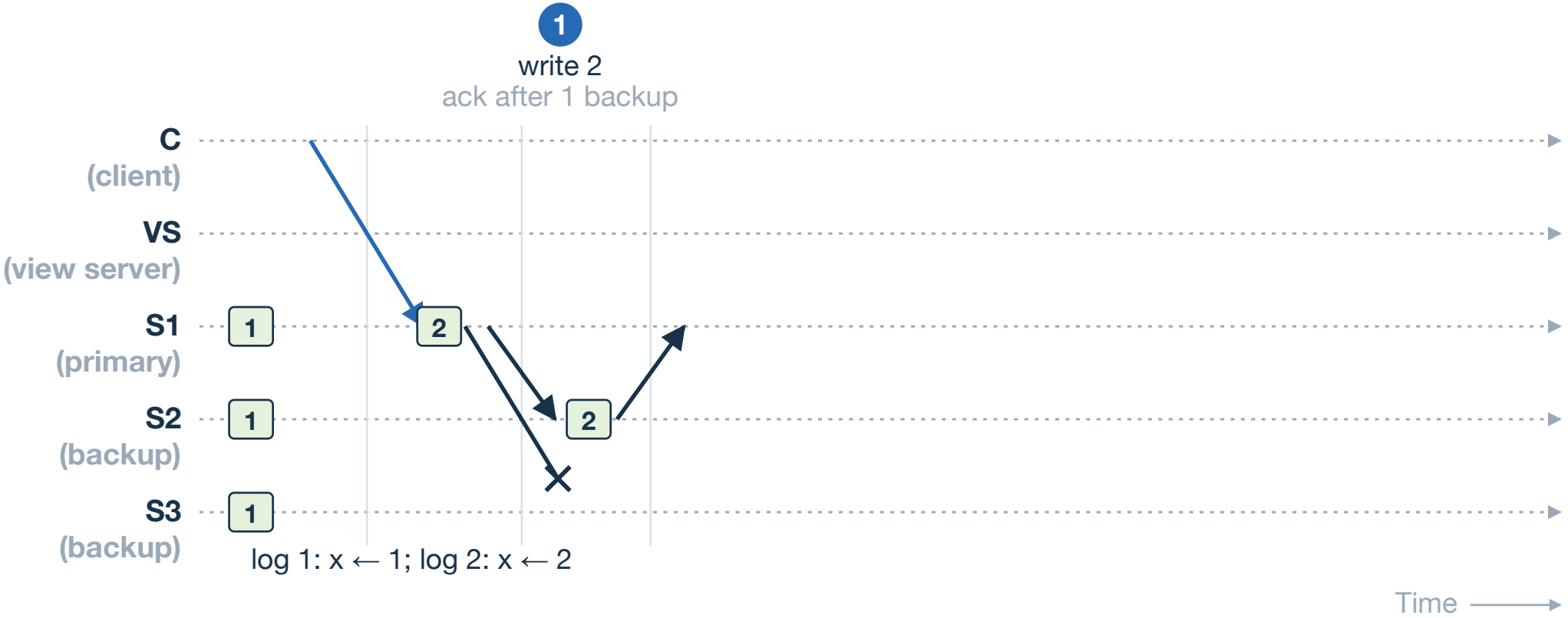
Question: which backup becomes the primary?



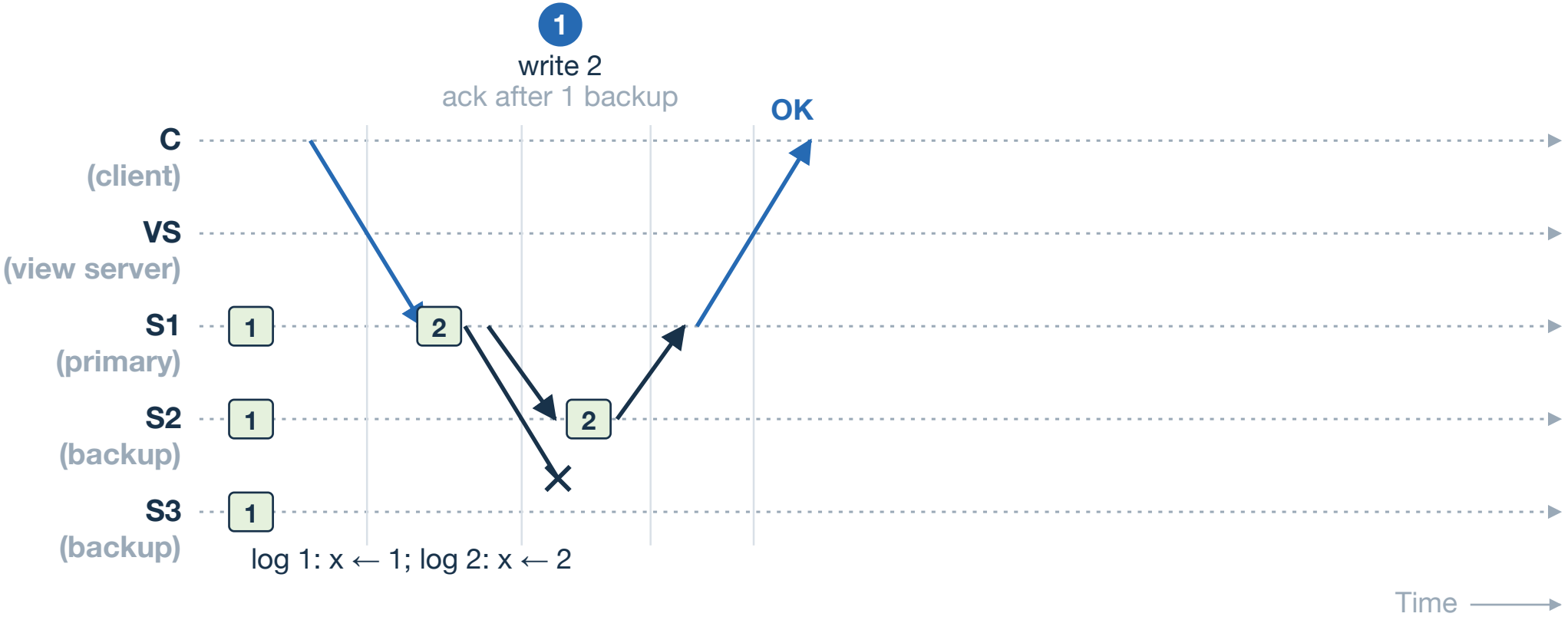
Question: which backup becomes the primary?



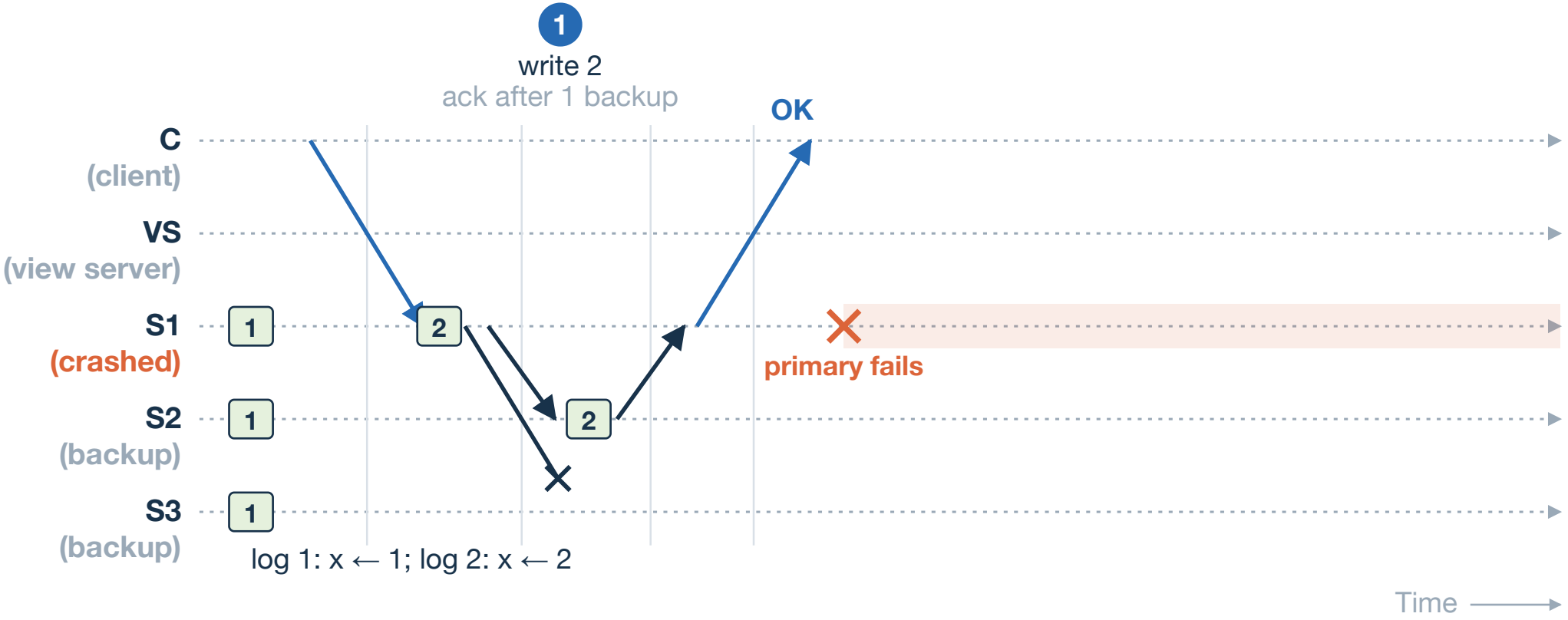
Question: which backup becomes the primary?



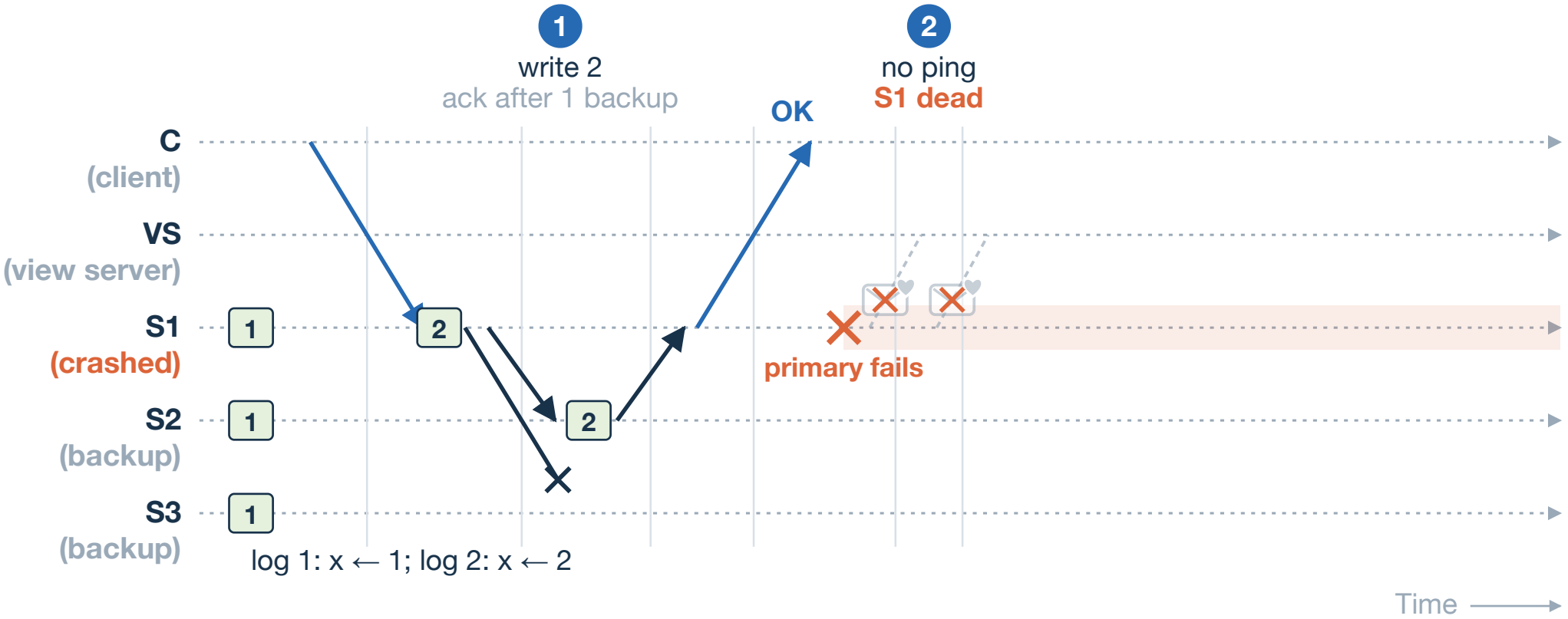
Question: which backup becomes the primary?



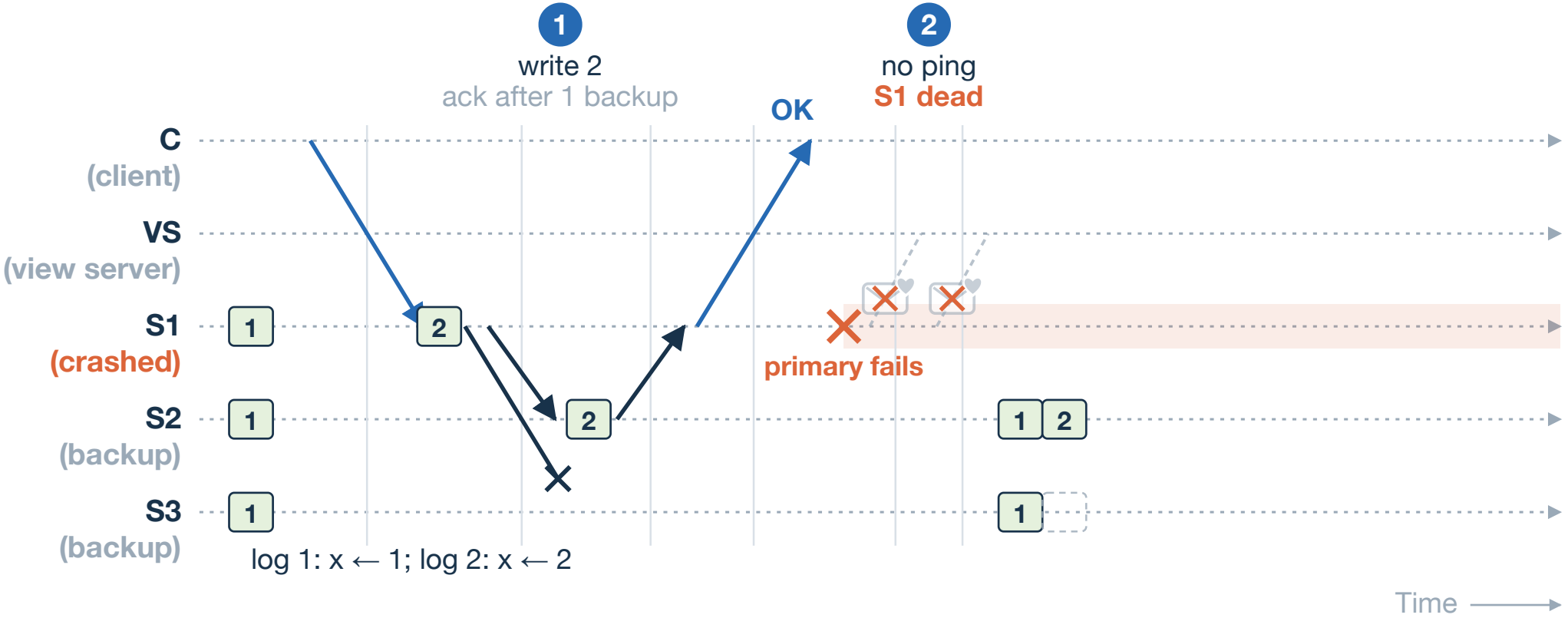
Question: which backup becomes the primary?



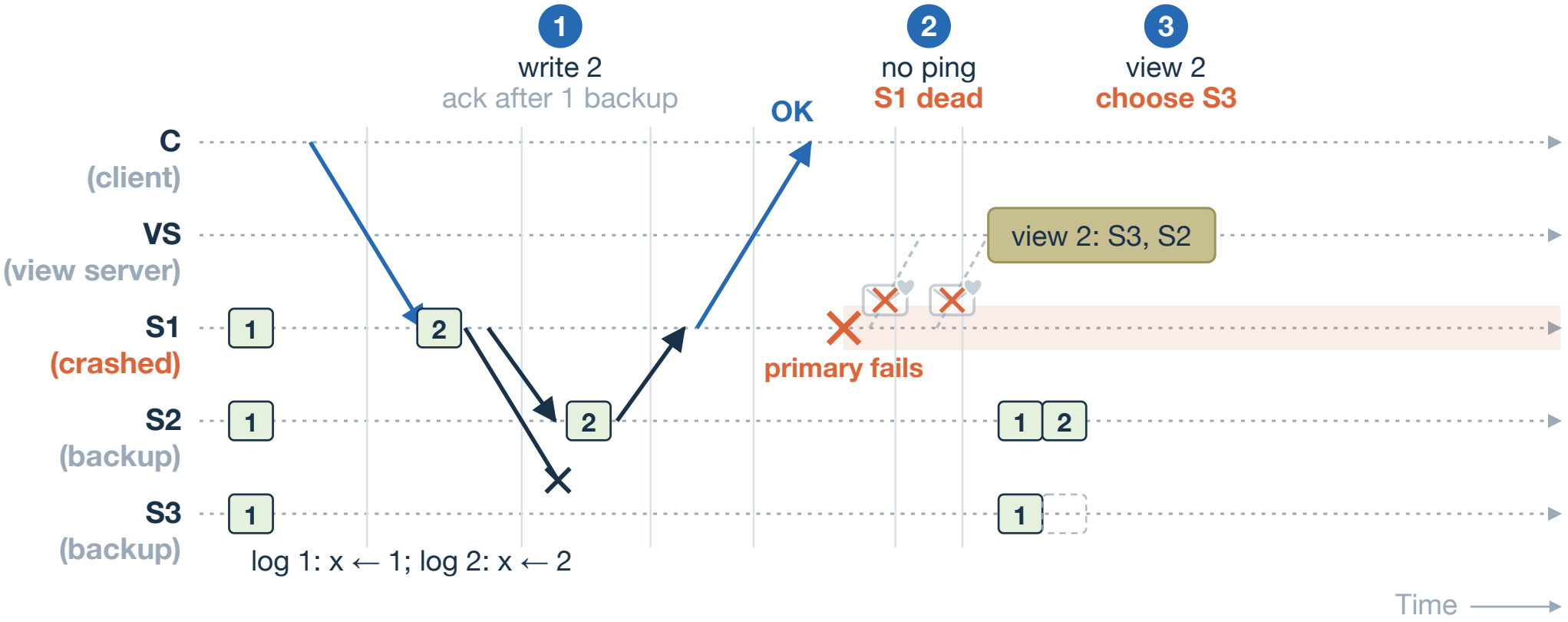
Question: which backup becomes the primary?



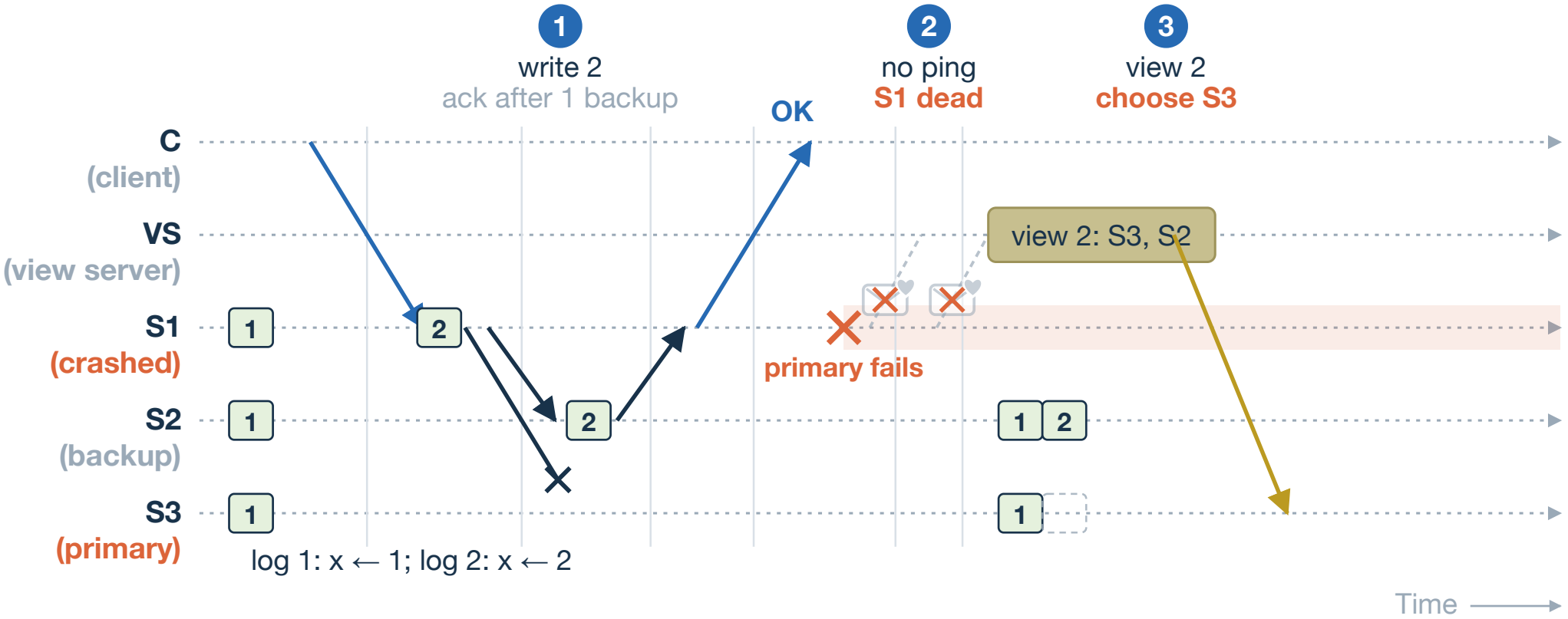
Question: which backup becomes the primary?



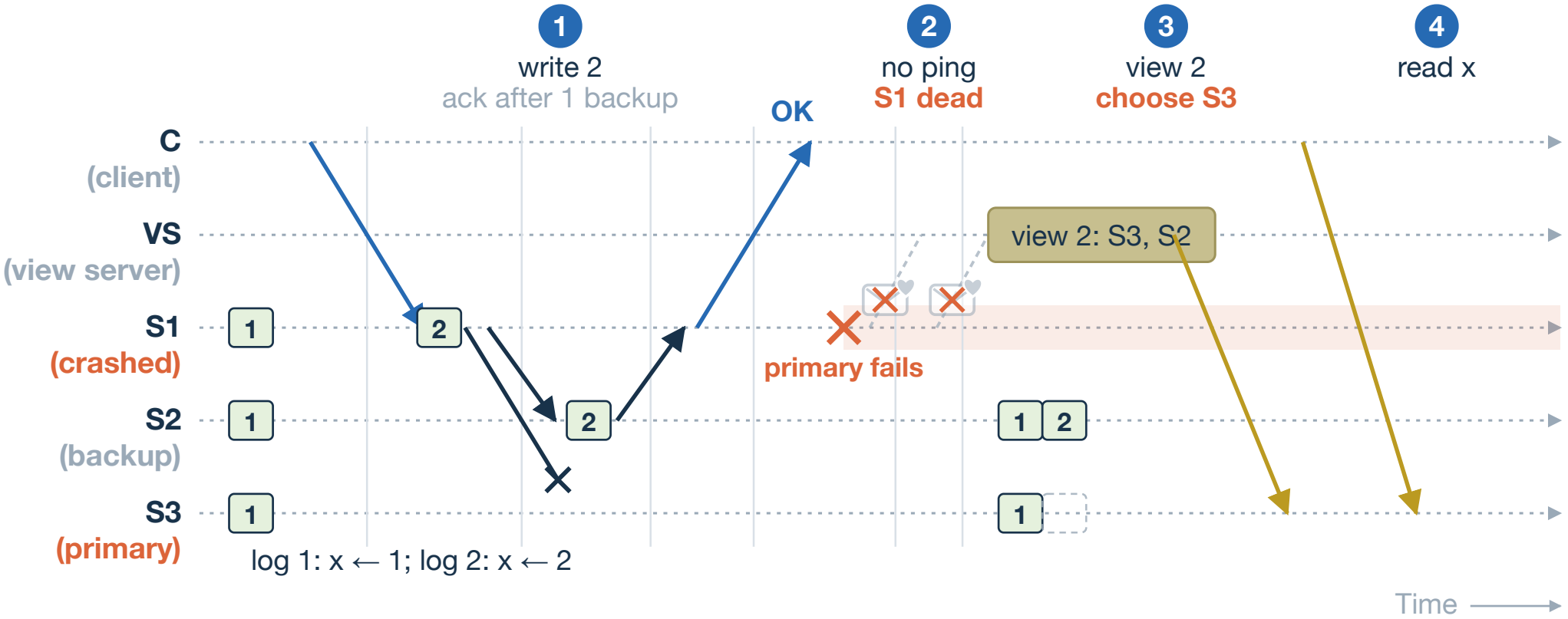
Question: which backup becomes the primary?



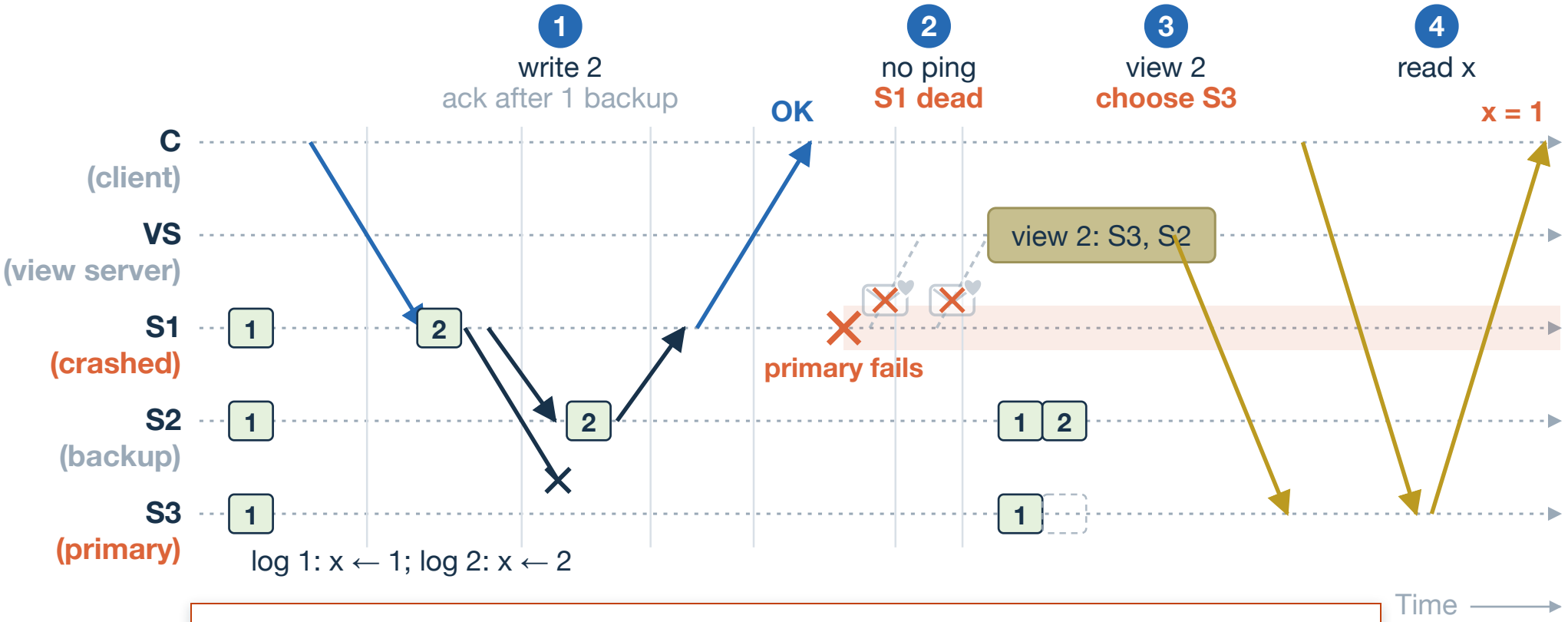
Question: which backup becomes the primary?



Question: which backup becomes the primary?

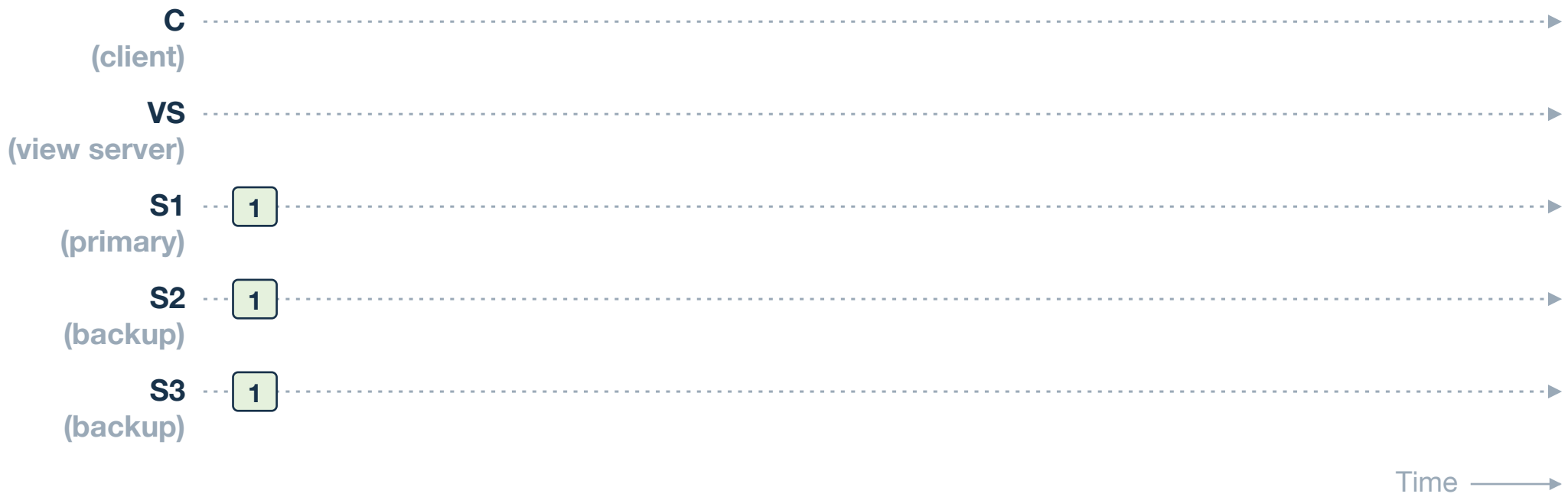


Question: which backup becomes the primary?

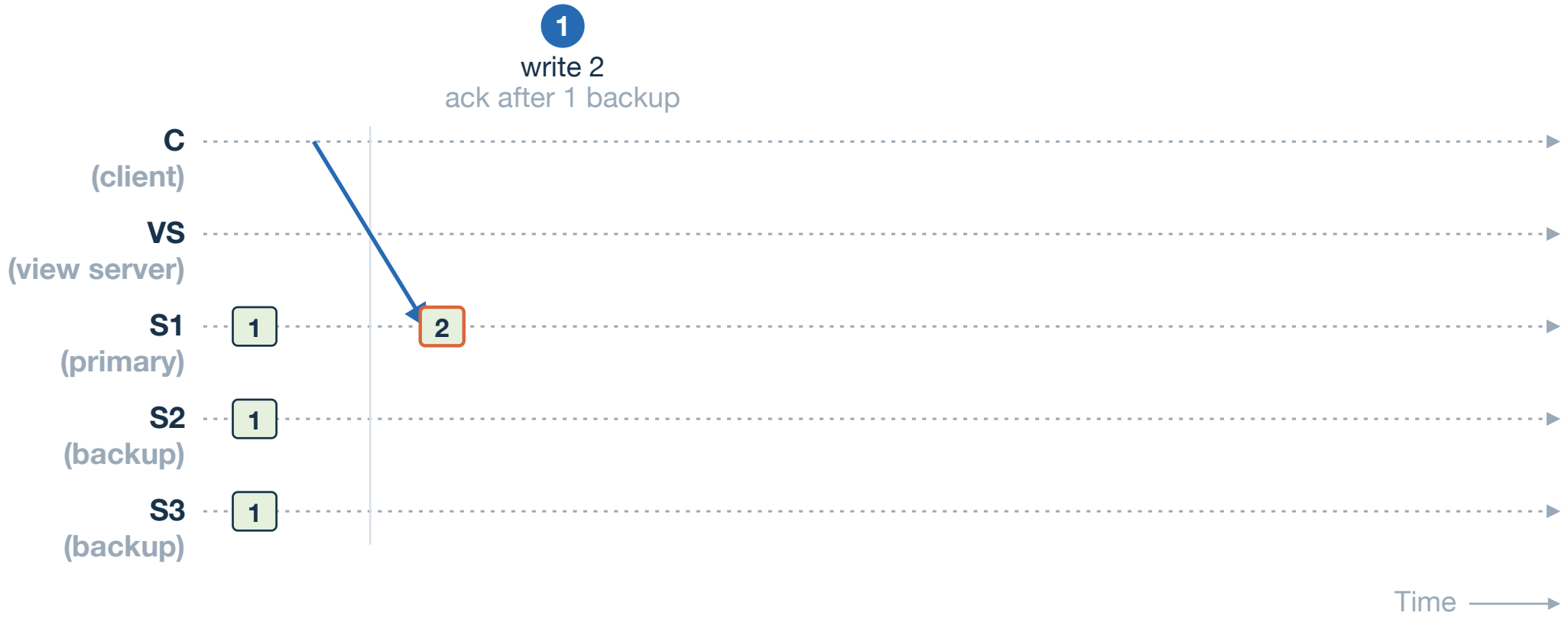


Partial ACKs + a stale primary:
an acknowledged write is missing after failover

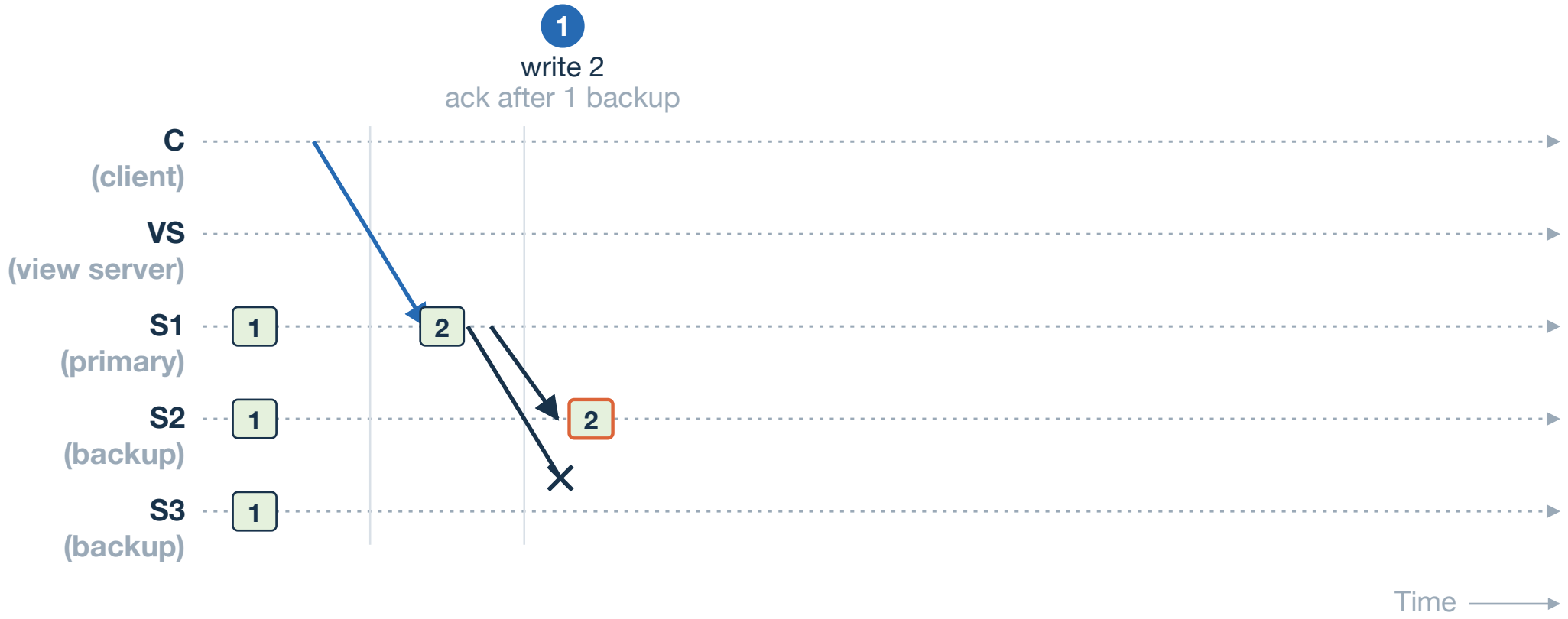
Question: when is a write reliable?



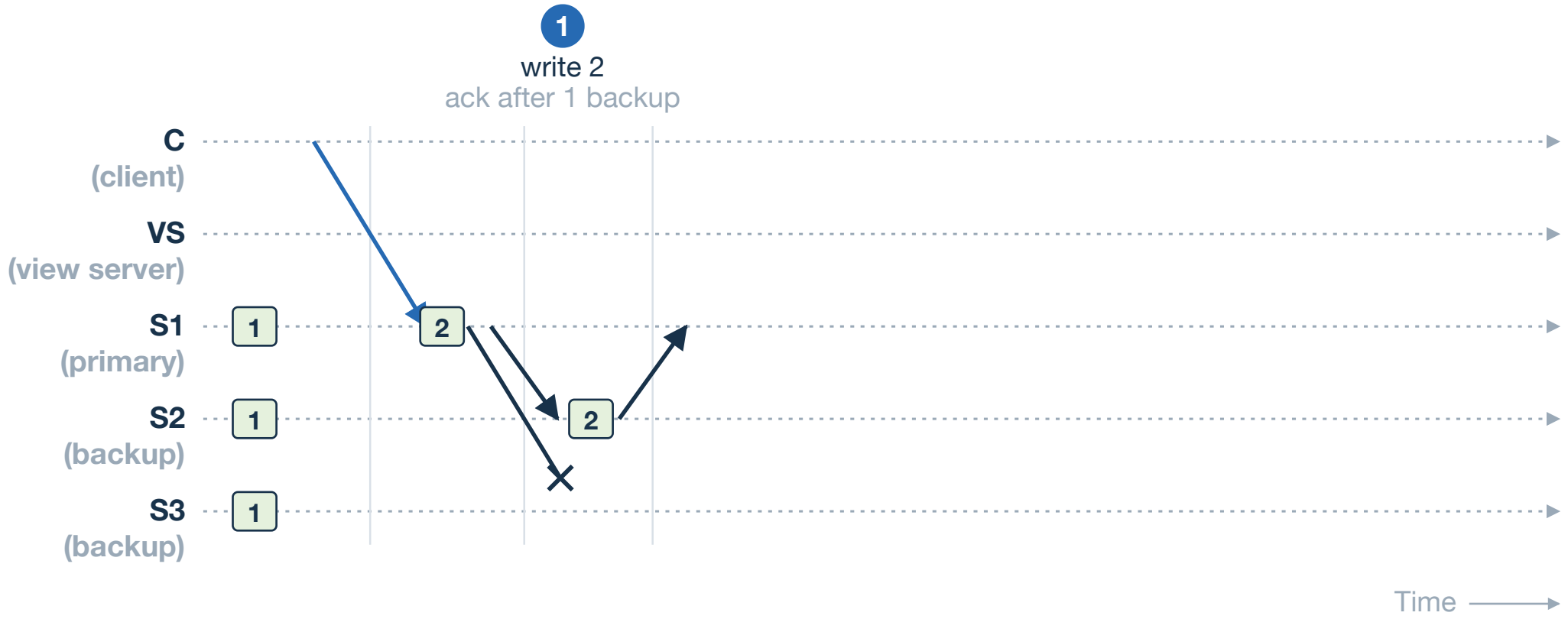
Question: when is a write reliable?



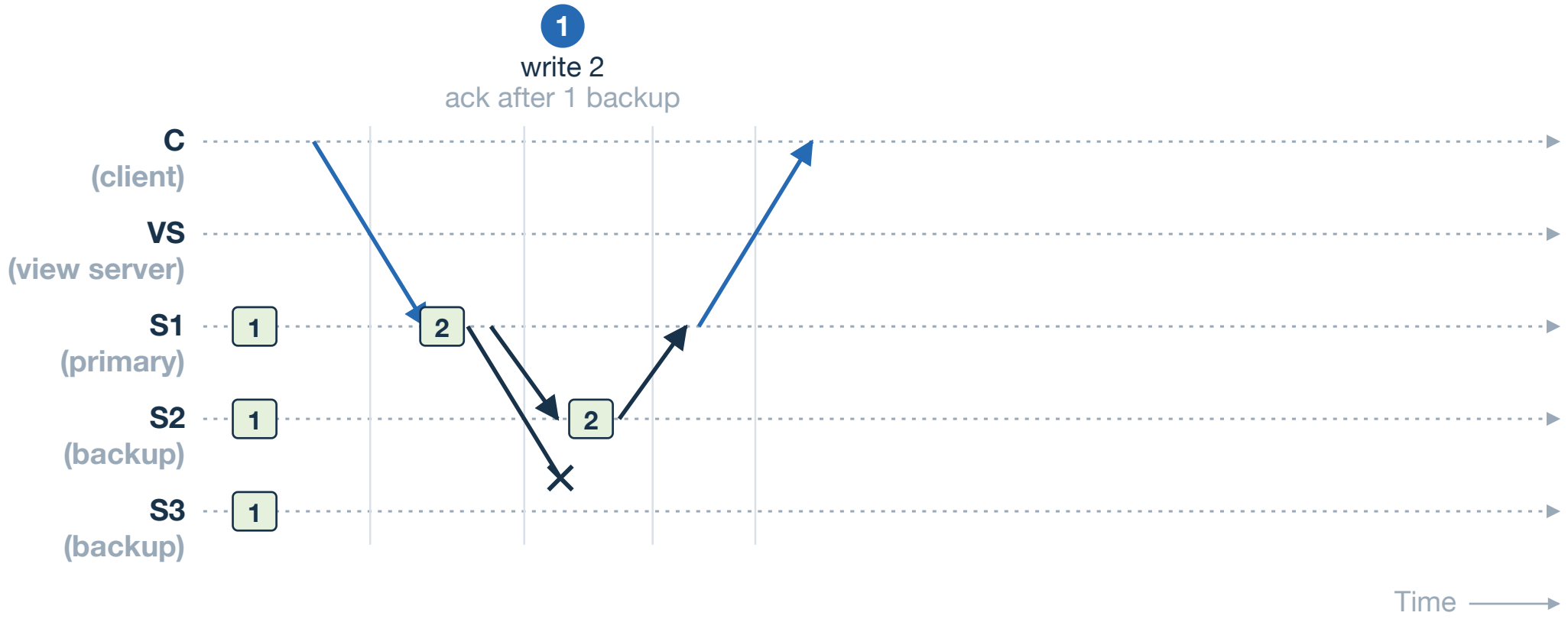
Question: when is a write reliable?



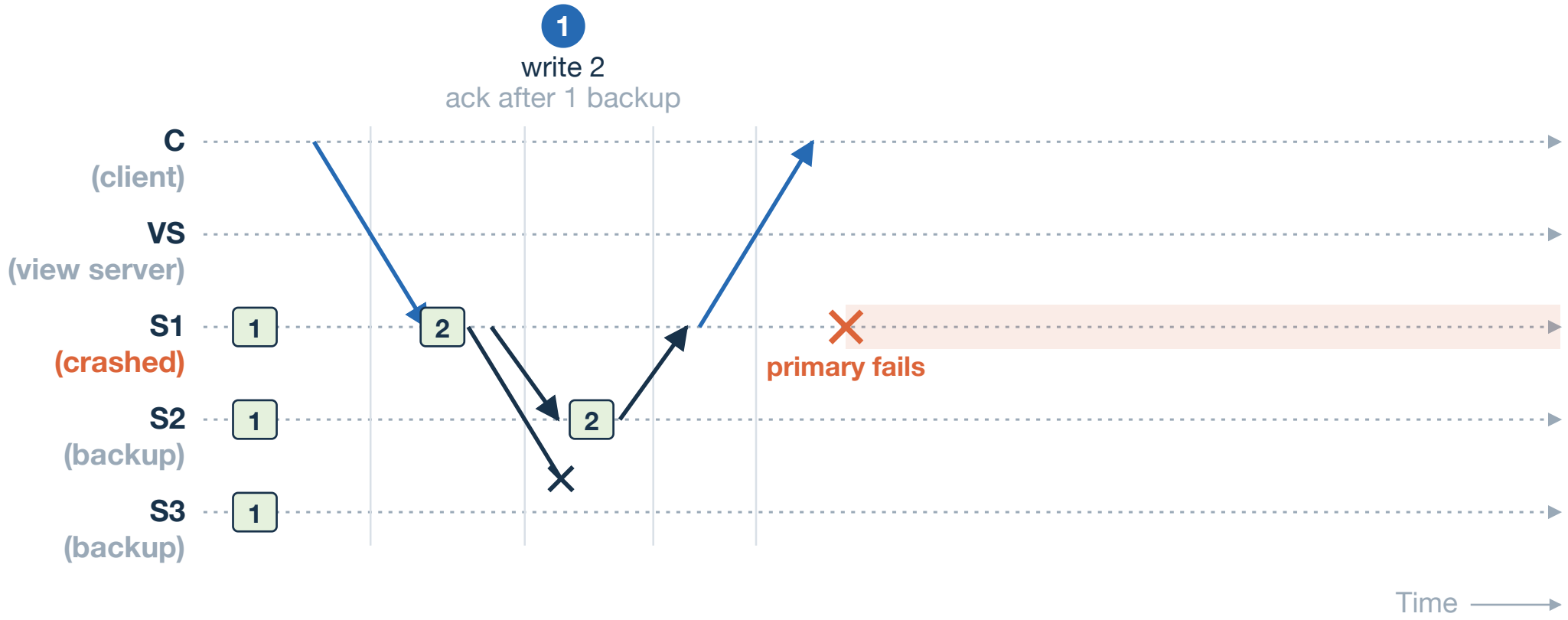
Question: when is a write reliable?



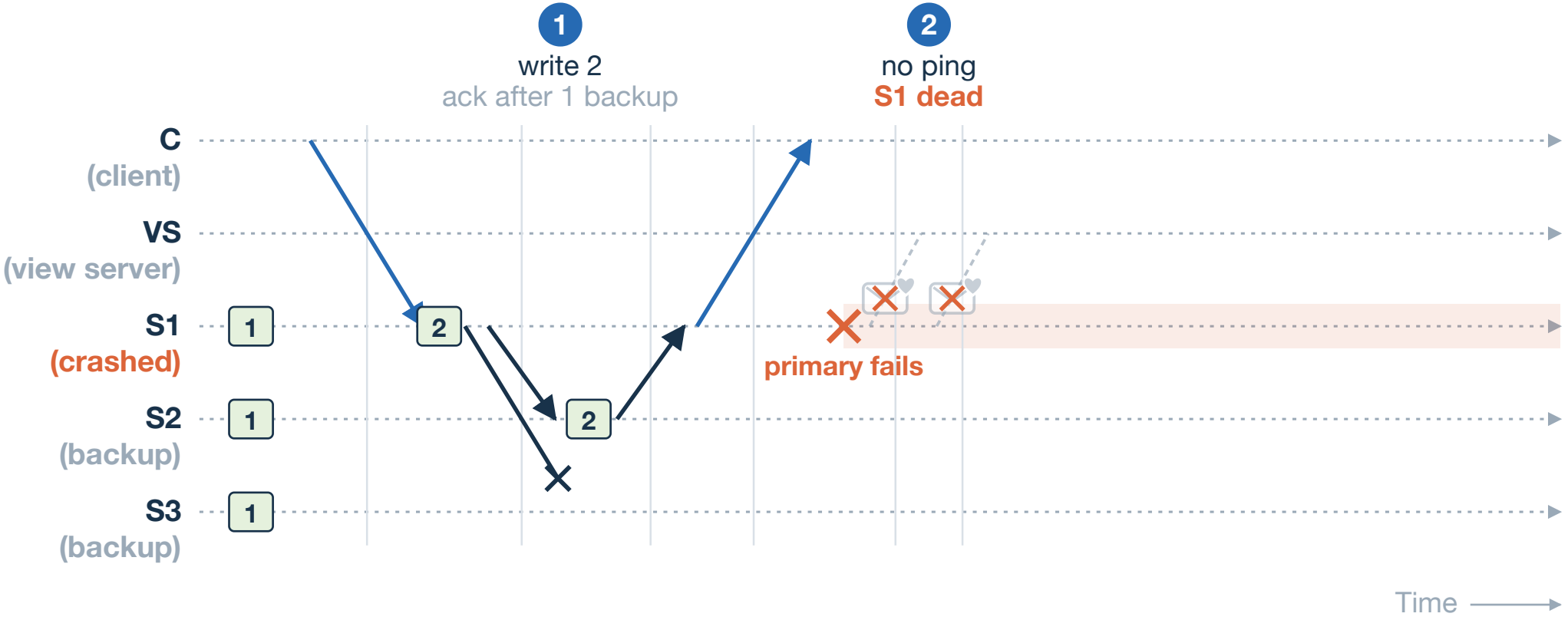
Question: when is a write reliable?



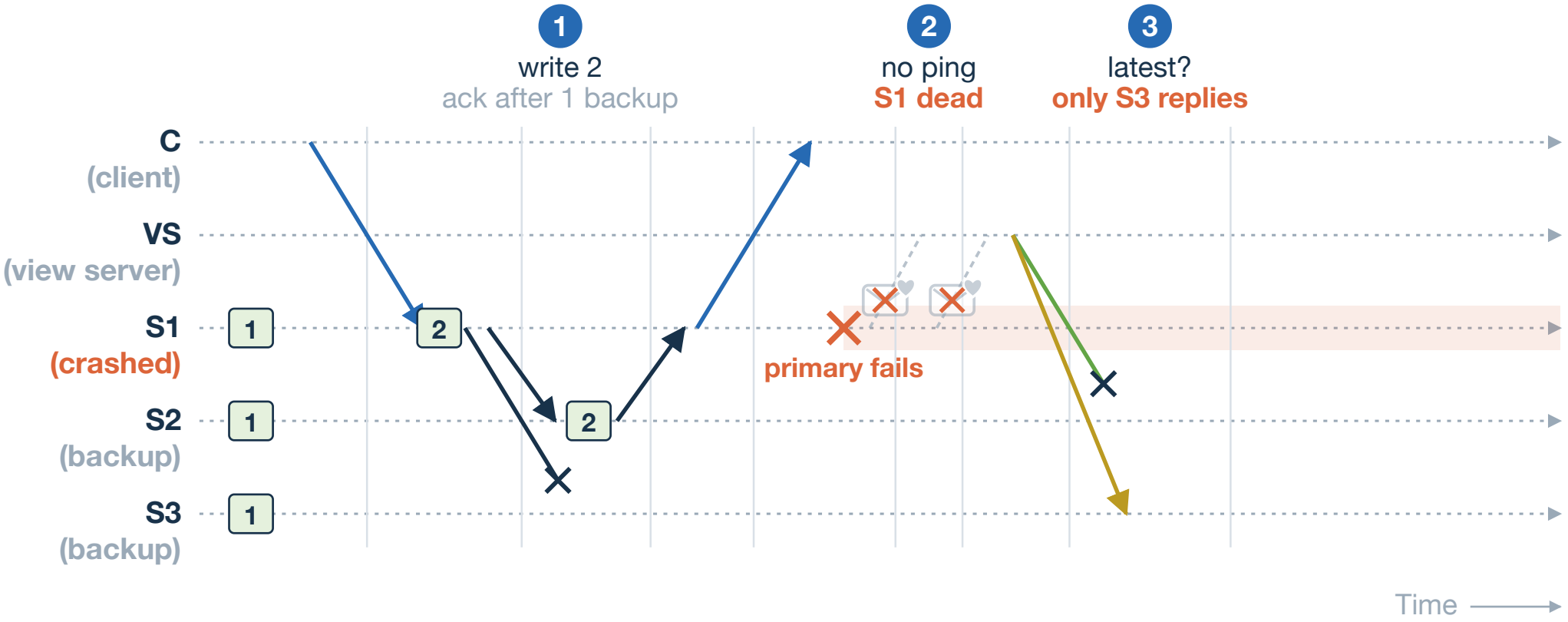
Question: when is a write reliable?



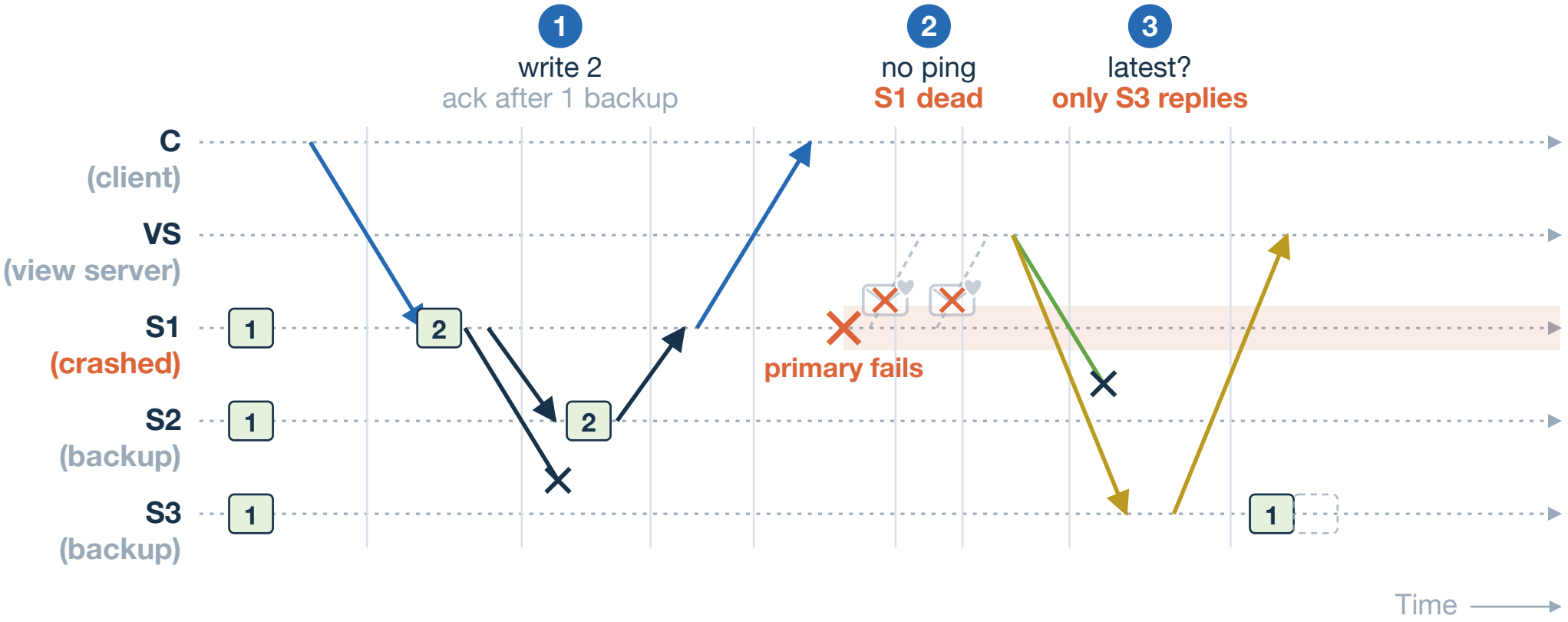
Question: when is a write reliable?



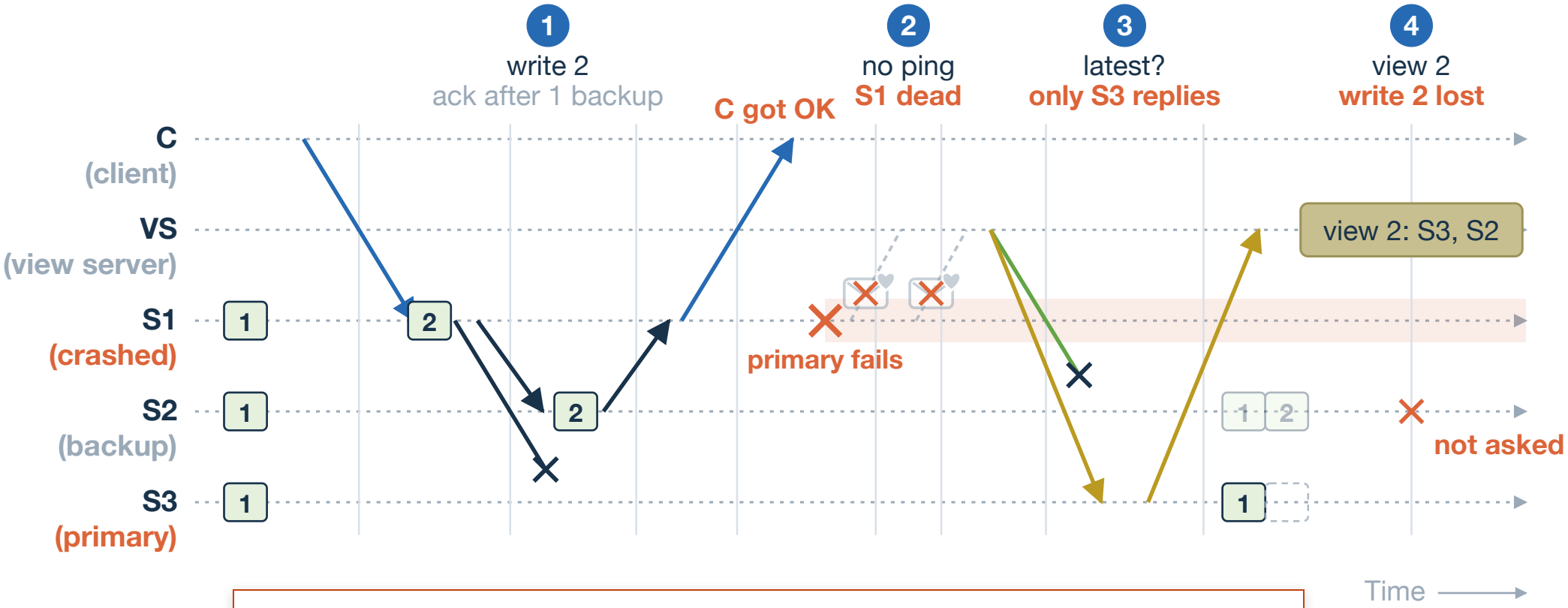
Question: when is a write reliable?



Question: when is a write reliable?



Question: when is a write reliable?



acked to C, yet lost: the backups written to and the backups asked do not overlap

Quorum

Simplified assumption: there is only one writer (e.g., primary)

- Goal: the reader reads the write if the writer returns OK

Define separate read & write quorums: Q_r & Q_w

- $Q_r + Q_w > N_{replicas}$ (Why?)
  - Confirm a write after writing to at least Q_w of replicas
  - Read from at least Q_r replicas, and take the newest version among them

Example

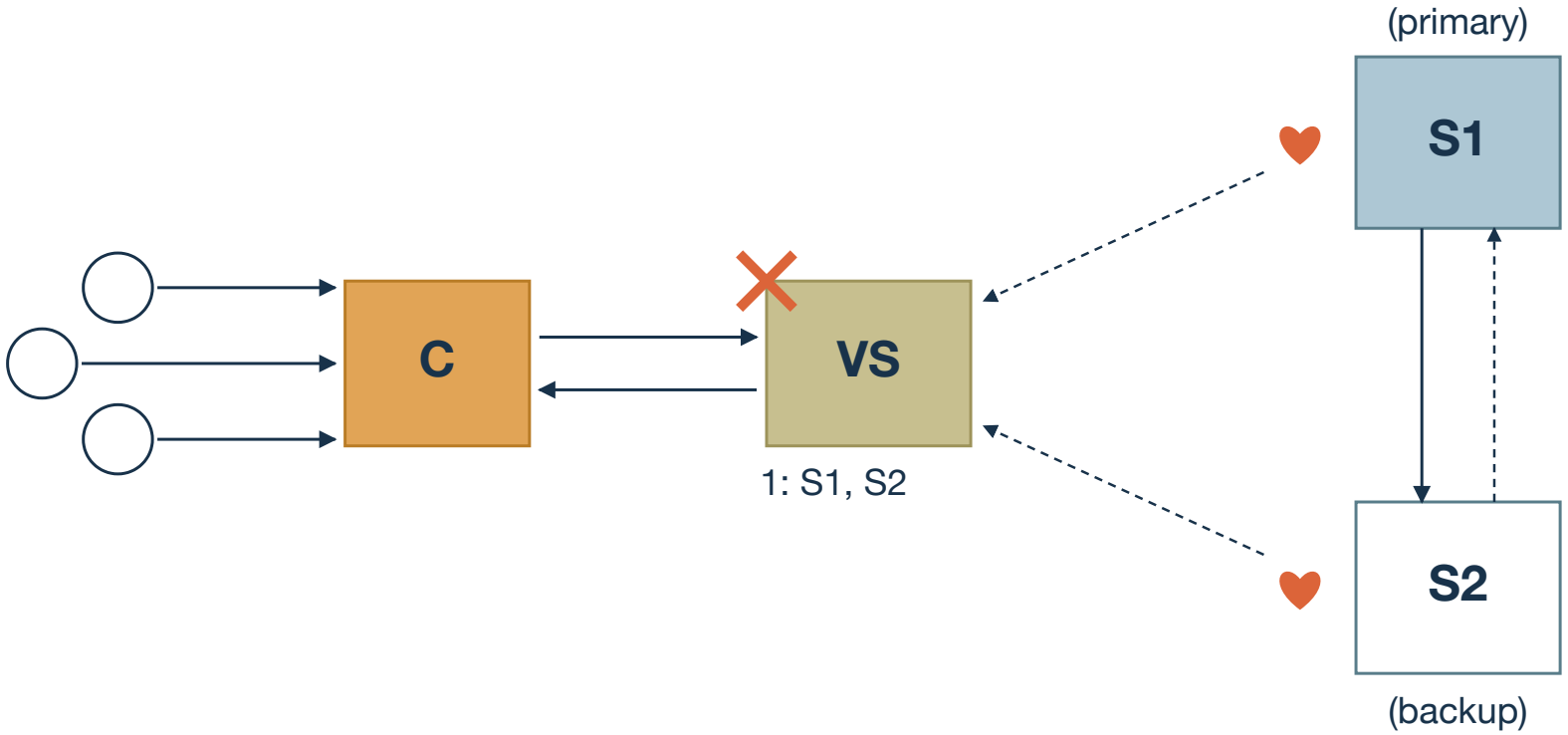
- In favor of reading: $N_{replicas} = 5, Q_w = 4, Q_r = 2$
- In favor of updating: $N_{replicas} = 5, Q_w = 2, Q_r = 4$
- Read from any single replica: $Q_w = N_{replicas}, Q_r = 1$

Put it all together: the complete data path of primary-backup

1. One server is responsible for ordering inputs received from clients
2. After receiving an input, sync to at least W backups before ack
3. If the primary fails, select R backups, and choose the one with the most up-to-date inputs, i.e., inputs with the largest sequence number
   - $R + W > N$ (Total number of replicas)
   - The primary must collect logs from these backups to form a complete log, and replicate it to the remaining backups (the recovery process)

Recovery typically completes within 30 ms

Final issue: what if view server fails?



Now we need Paxos or Raft — covered later

参考文献

[1] Notes on Data Base Operating Systems — Jim Gray, 1978. Operating Systems: An Advanced Course (LNCS 60) (P4)

[2] A Scalable, Commodity Data Center Network Architecture — Al-Fares, Mohammad and Loukissas, Alexander and Vahdat, Amin, 2008. SIGCOMM (P19)

[3] Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services — Gilbert, Seth and Lynch, Nancy, 2002. ACM SIGACT News (P20)

[4] CAP Twelve Years Later: How the “Rules” Have Changed — Brewer, Eric, 2012. IEEE Computer (P20)

[5] Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial — Fred B. Schneider, 1990. ACM Computing Surveys (P28)

[6] Weighted Voting for Replicated Data — David K. Gifford, 1979. SOSP (P40)