RSM & PAXOS

Consistency across replicas

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS & Jinyang Li@NYU & John Ousterhout

Review: Two-phase Commit for Multi-site Atomicity

Phase-1: preparation / voting

- Lower-layer transactions either aborts or *tentatively* committed
- Higher-layer transaction evaluate lower situation

Phase-2: commitment

- If top-layer commits or aborts, then lower-layer workers COMMIT or ABORT

Challenge

- Unreliable communications + participants

Review: Two-phase Commit for Multi-site Atomicity

Principles:

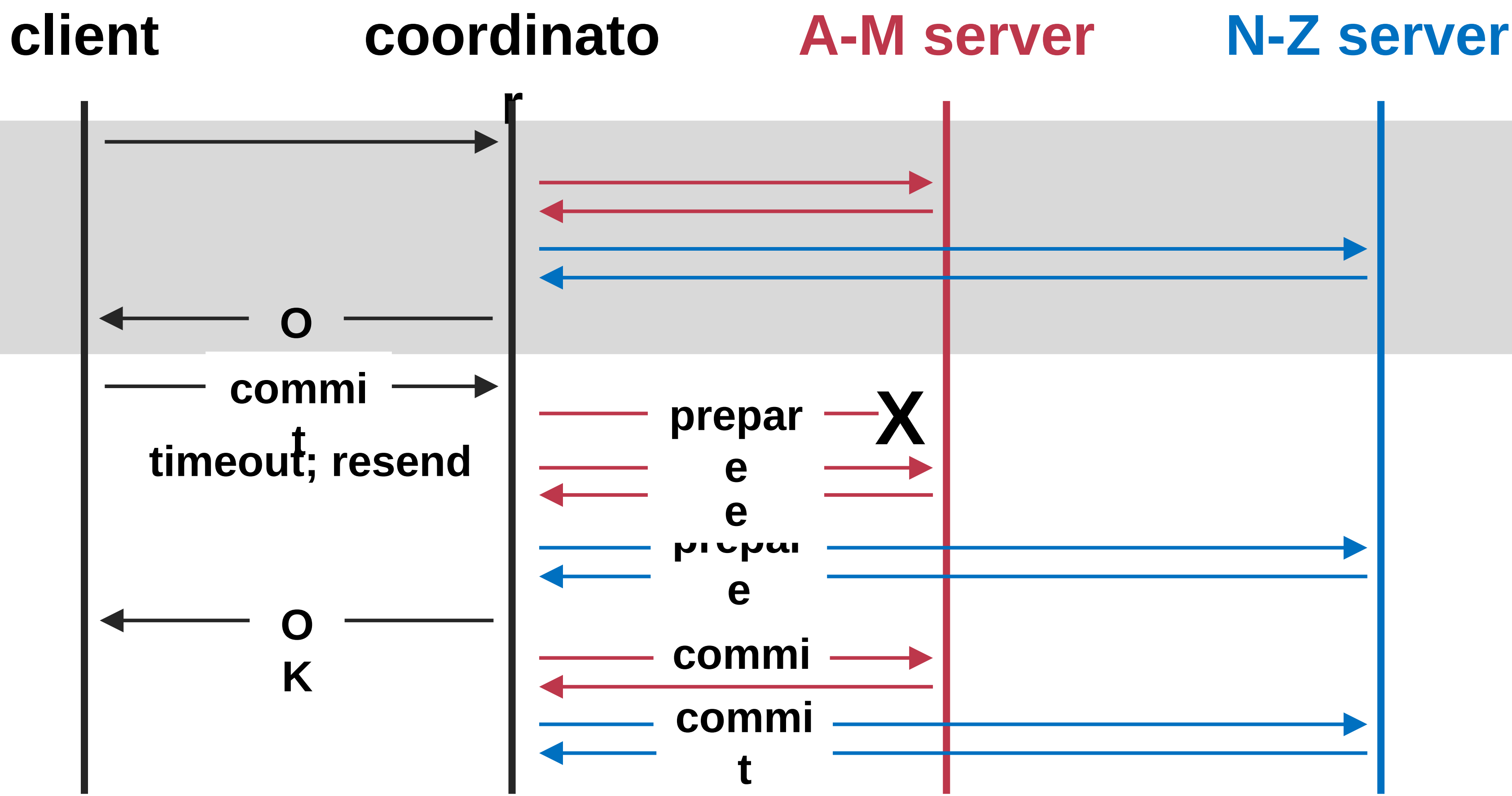
- Following the coordinator's decision
- Log sufficient state to tolerate failures (e.g., the coordinator's decision)

Coordinator (do the decision & maintain the state)

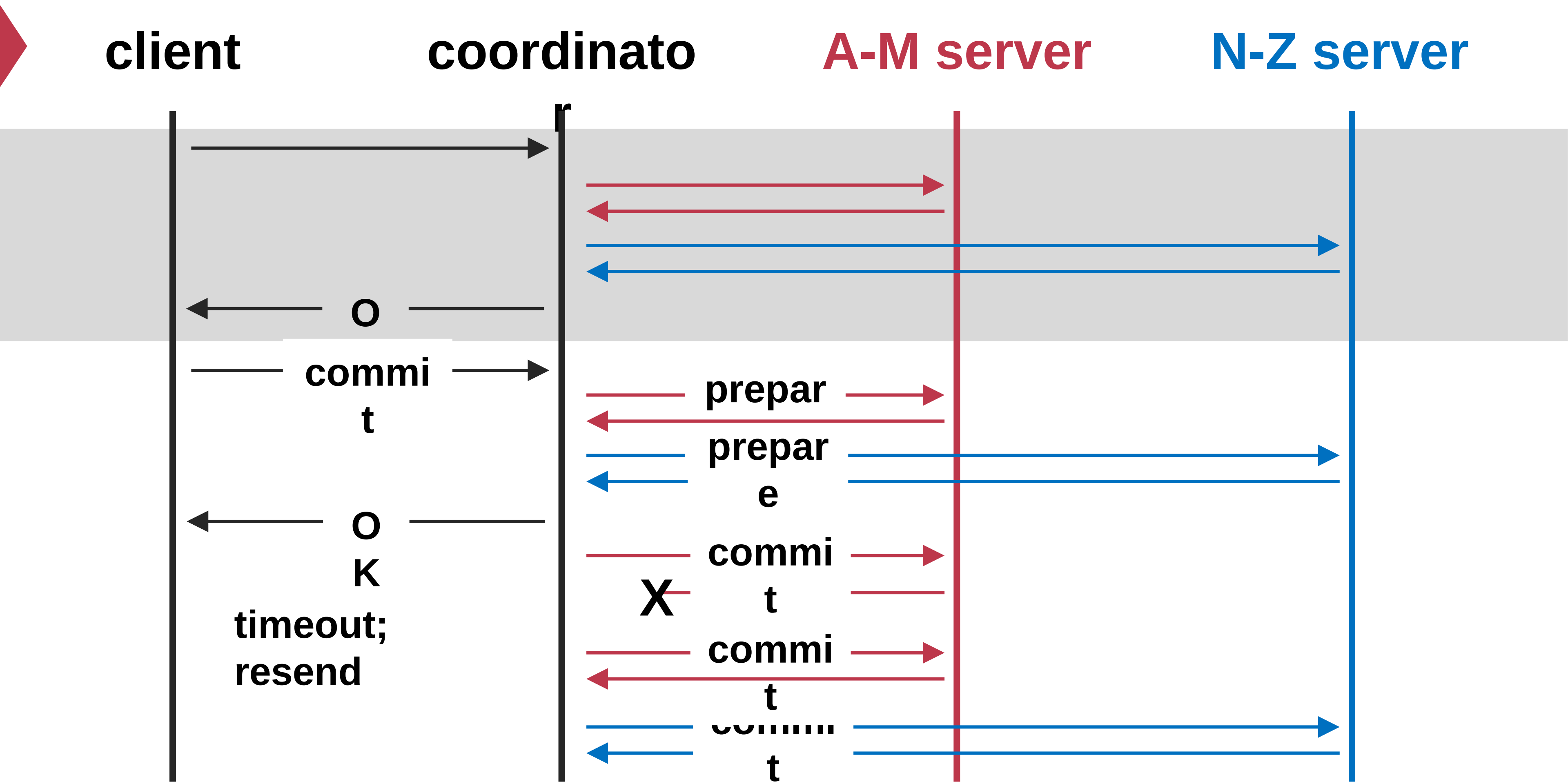
- Collect some **ABORT** or nothing: **ABORT** or retry
- Collect all **COMMIT**: then **COMMIT**

Worker passively react to the coordinator's actions

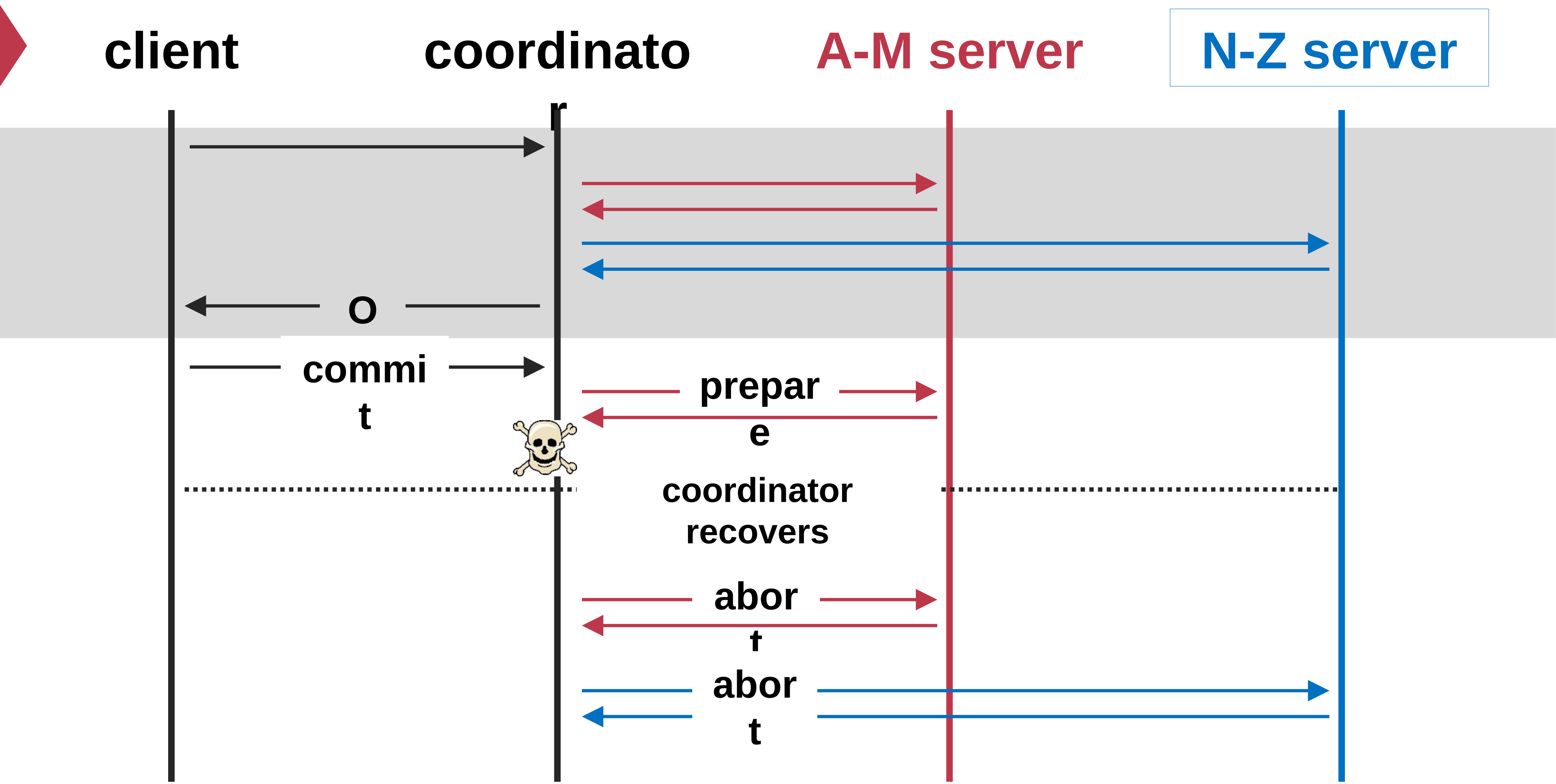
- If no message is sent from the coordinator, just wait
- When receive **COMMIT**: then **COMMIT**



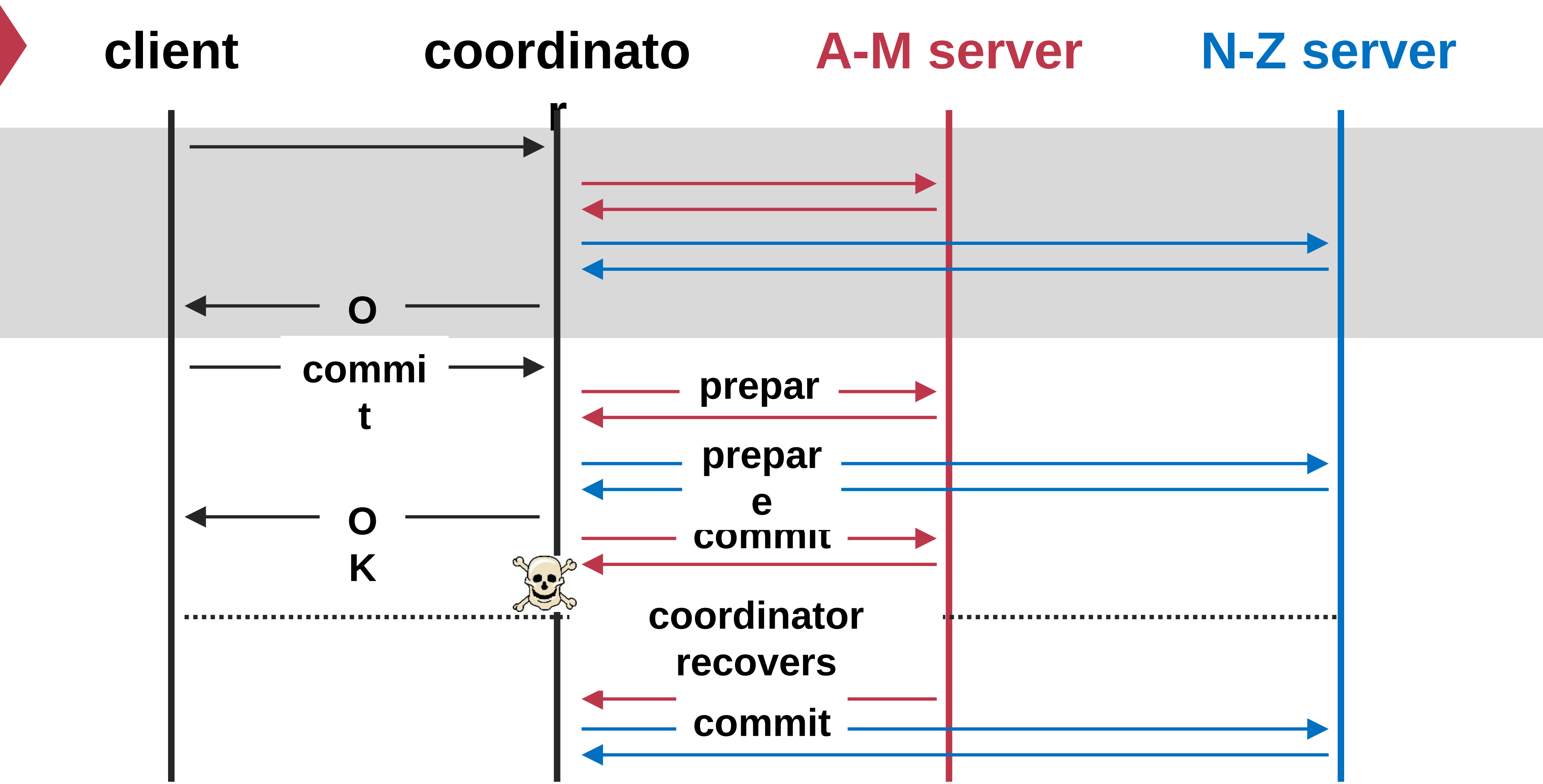
failure: lost prepare



failure: lost ACK of commit message



failure: coordinator failure during prepare



failure: **coordinator failure during commit**

Must log the decision before sending the commit messages

2PC and CAP

Review of The CAP theorem: 2 out of 3

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees

- **Consistency** (all nodes see the same data at the same time, e.g., linearizability)
- **Availability** (a guarantee that every request receives a response about whether it succeeded or failed)
- **Partition tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)

Review of The CAP theorem: 2 out of 3

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees

- **Consistency** (all nodes see the same data at the same time)
- **Availability** (a guarantee that every request receives a response about whether it succeeded or failed)
- **Partition tolerance** (the system continues to operate despite arbitrary messa

Question: which property does 2PC achieve?

2PC only guarantees consistency!

If the coordinator fails before sending the commit

- All other transaction must wait until it wakes up
- The coordinator logs its decisions to recovery & resume after the failure

If one site fails during the transaction's execution

- All TX requiring this site's involvement must wait until it wakes up

Only some corner cases can ensure availability or partition tolerant of a TX

- By aborting TXs. But it doesn't help much.

We need replication to achieve high availability!

Review: Two-phase Commit

Two-phase commit allows us to achieve multi-site atomicity: transaction remains atomic even when they require communication with multiple machines

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

Our remaining issue deals with availability and replication: we will replicate data across sites to improve availability, we may also replicate the coordinator, but must deal with keeping multiple copies of the data consistent

We focus on replicating the data. Replicating the coordinator is similar

Replication is everywhere

For performance

– Higher throughput: replicas can serve concurrently

– Lower latency: cache is also a form of replication

For fault tolerance

– Maintain availability even if some replicas fail

Replication Consistency

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., linearizability)

- Ensure strong consistency between replicas
- Needed when out-of-sync replicas can cause serious problems

Pessimistic replication

Pessimistic Replication

Some applications may prefer not to tolerate inconsistency

- E.g., a replicated lock server, or replicated coordinator for 2PC
    - Better not give out the same lock twice
- E.g., Better have a consistent decision about whether transaction commits

Trade-off: stronger consistency with pessimistic replication means:

- Lower availability than what you might get with optimistic replication
- Performance overhead for waiting syncing w/ other replicas

Goal: Single-copy Consistency (Linearizability)

Problem of optimistic way: replicas get out of sync

- One replica writes data, another doesn't see the changes
- This behavior was impossible with a single server

Ideal goal: single-copy consistency

- Property of the externally-visible behavior of a replicated system
- Operations appear to execute as if there's only a single copy of the data
    - Internally, there may be failures or disagreement, which we have to mask
- Similar to how we defined serializability goal ("as if executed serially")

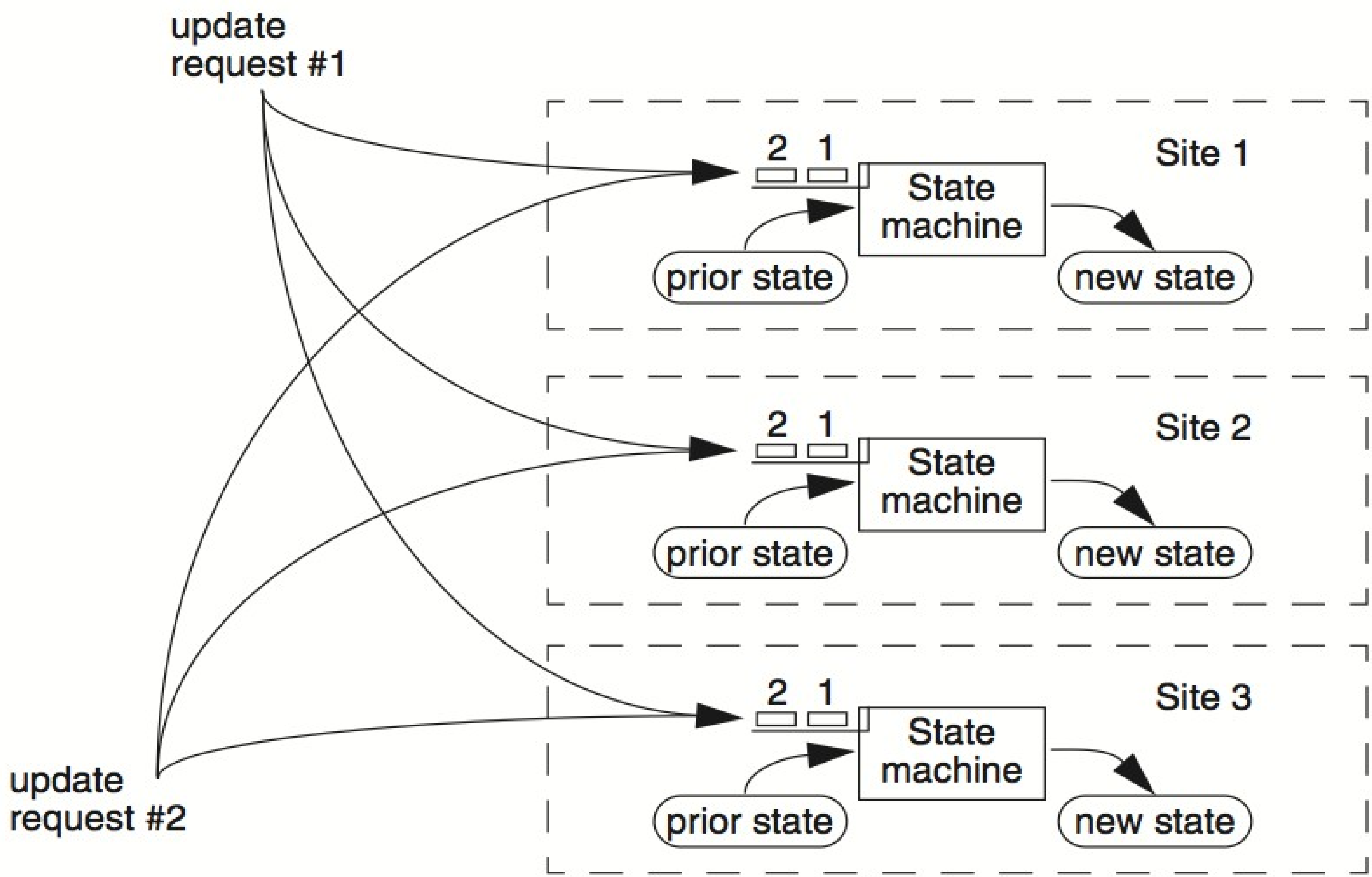
Replicated State Machines (RSM)

RSM: Replicated State Machines

A general approach to making consistent replicas of a server:

- Start with the **same initial state** on each server
- Provide each replica with the **same input** operations, in **same order**
- Ensure all operations are **deterministic**
  - E.g., no randomness, no reading of current time, etc.

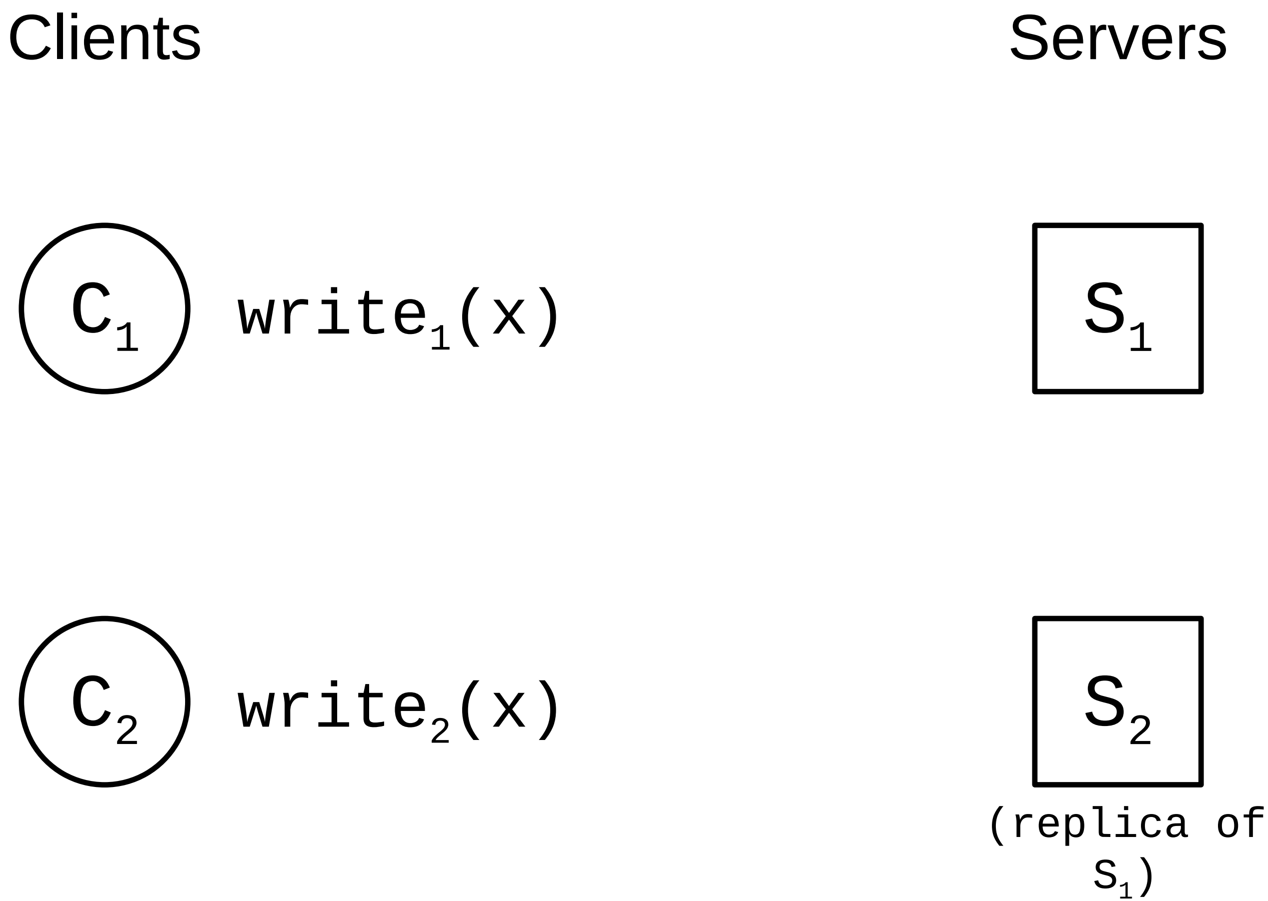
These rules ensure each server will end up in the same final state

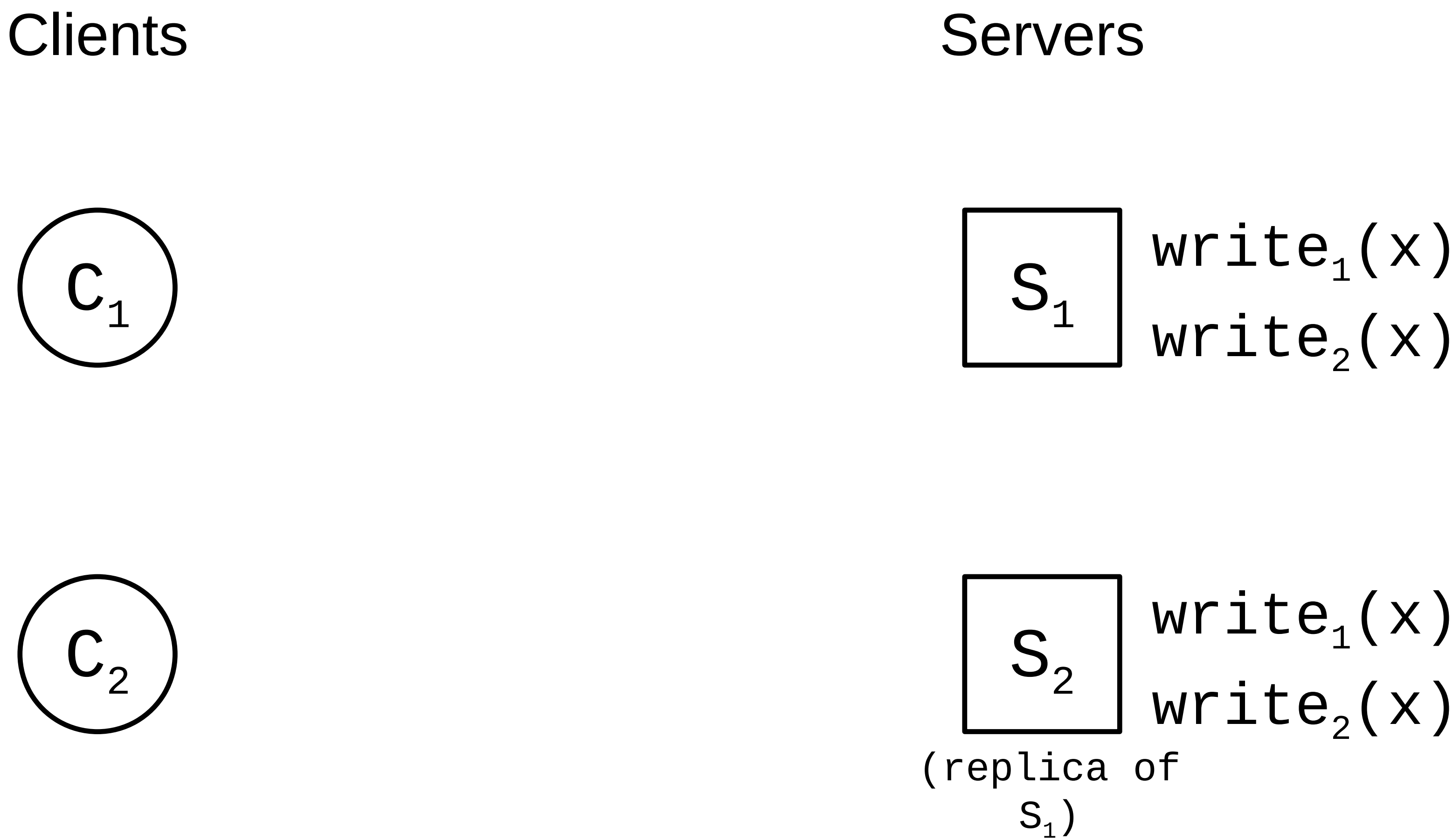

update
request #1
Site 1
2 1
State
machine
prior state
new state
Site 2
2 1
State
machine
prior state
new state
Site 3
2 1
State
machine
prior state
new state
update
request #2

Inconsistency of Replicas

Problem: replicas can become inconsistent

- Issue: clients' requests to different servers can arrive in different order
- How do we ensure the servers remain consistent?
  - Unlike optimistic replication (e.g., eventual consistency), we cannot re-order events later, we must order it right now


Clients
Servers
C_1
$write_1(x)$
S_1
C_2
$write_2(x)$
S_2
(replica of S_1)



problem: How to ensure the order of operations?

Implementing RSM w/ Primary Backup model

RSMs provide single-copy consistency

- Operations complete as if there is a single copy of the data
- Though internally there are replicas

RSMs can use a primary-backup mechanism for replication

- Ensure only **one server** for ordering inputs received from clients
- It can also recruit new backups after servers fail

Primary does important stuff

- Ensures that it sends all inputs to the backup before ACKing the coordinator
- 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()`)

What if Primary Fails?

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

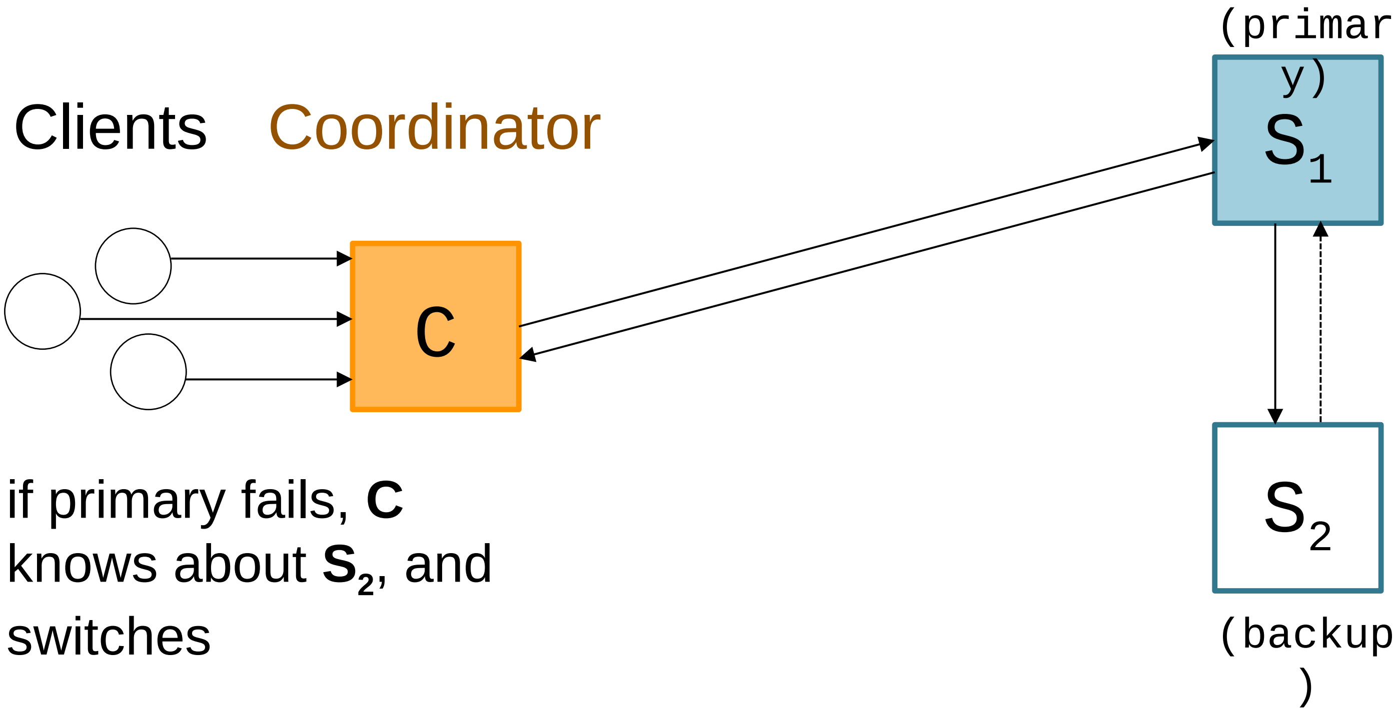
- Not unreasonable for small web services

Idea 2: Coordinator knows about both primary and backup, and decides which to use

- Won't work if using multiple coordinators: the "split brain" syndrome
- Multiple coordinators come to independent, and different, conclusions about who is primary when there are network partitions

Idea 3: we have some black magic to select the primary after a previous primary fails ⏪ (Will talk about in the later part of the lecture)

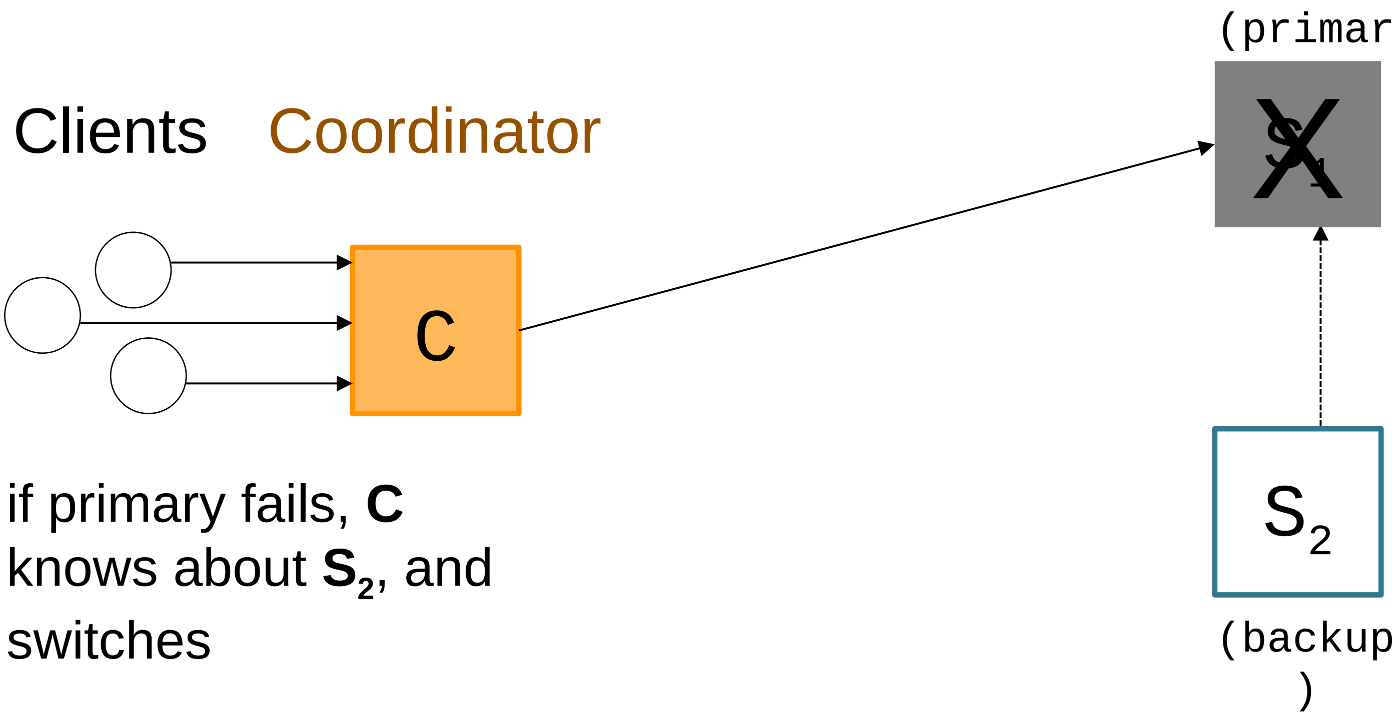
Primary/Backup Model



if primary fails, **C** knows about **S_2**, and switches

attempt: coordinators communicate with primary servers, who communicate with backup servers

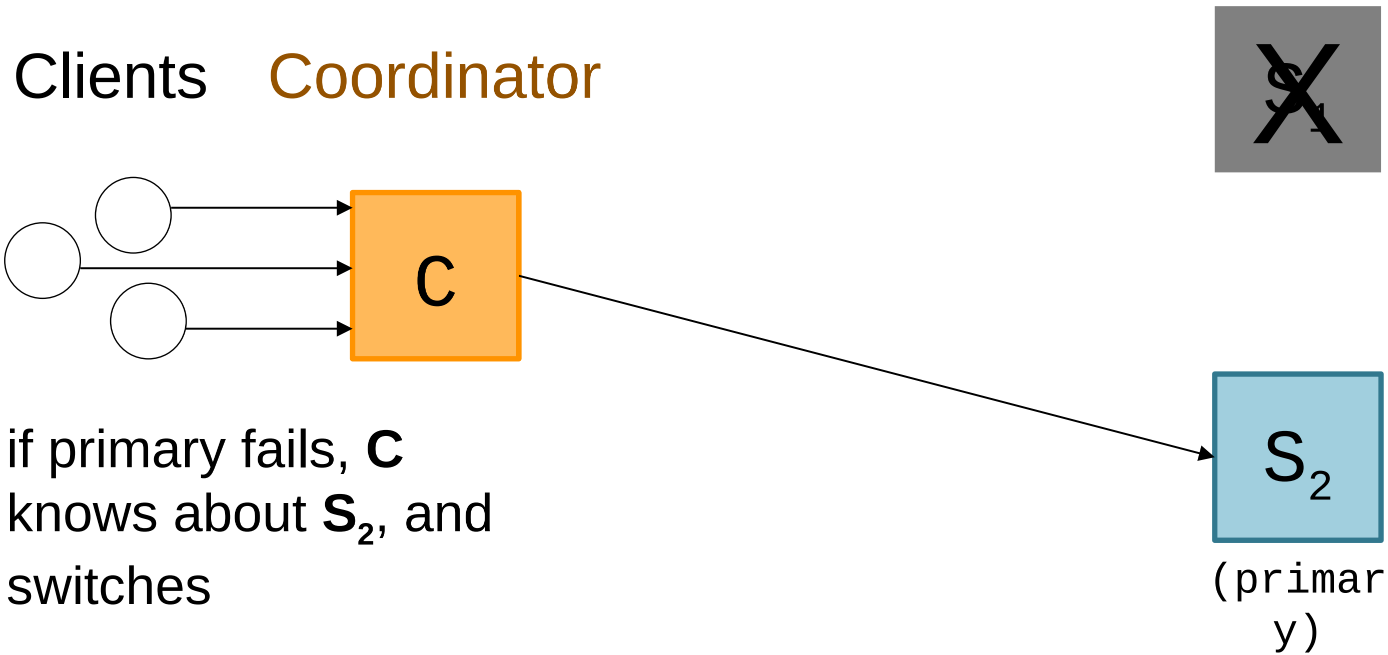
Primary/Backup Model



if primary fails, **C** knows about **S_2**, and switches

attempt: coordinators communicate with primary servers, who communicate with backup servers

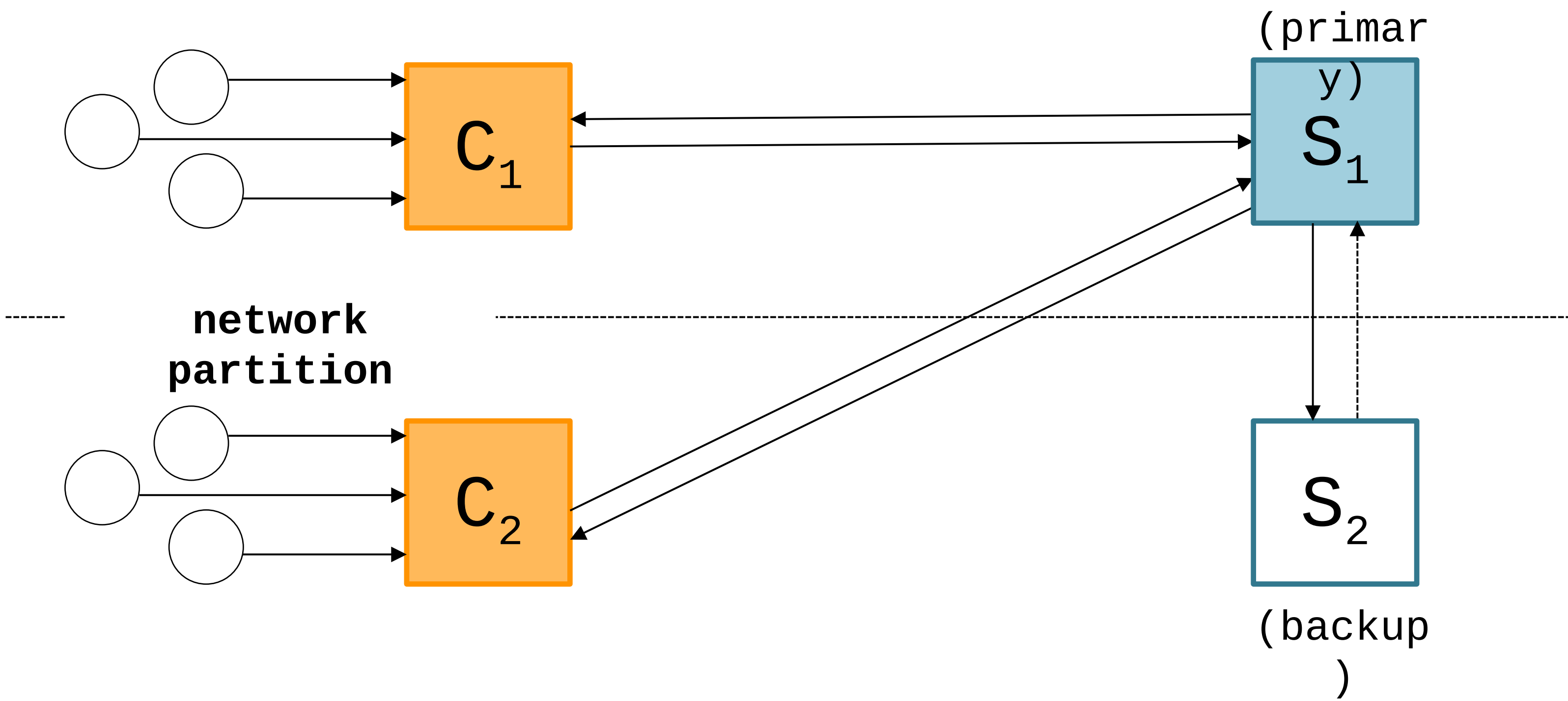
Multiple Coordinators + the Network = ?



if primary fails, **C** knows about $\mathbf{S_2}$, and switches

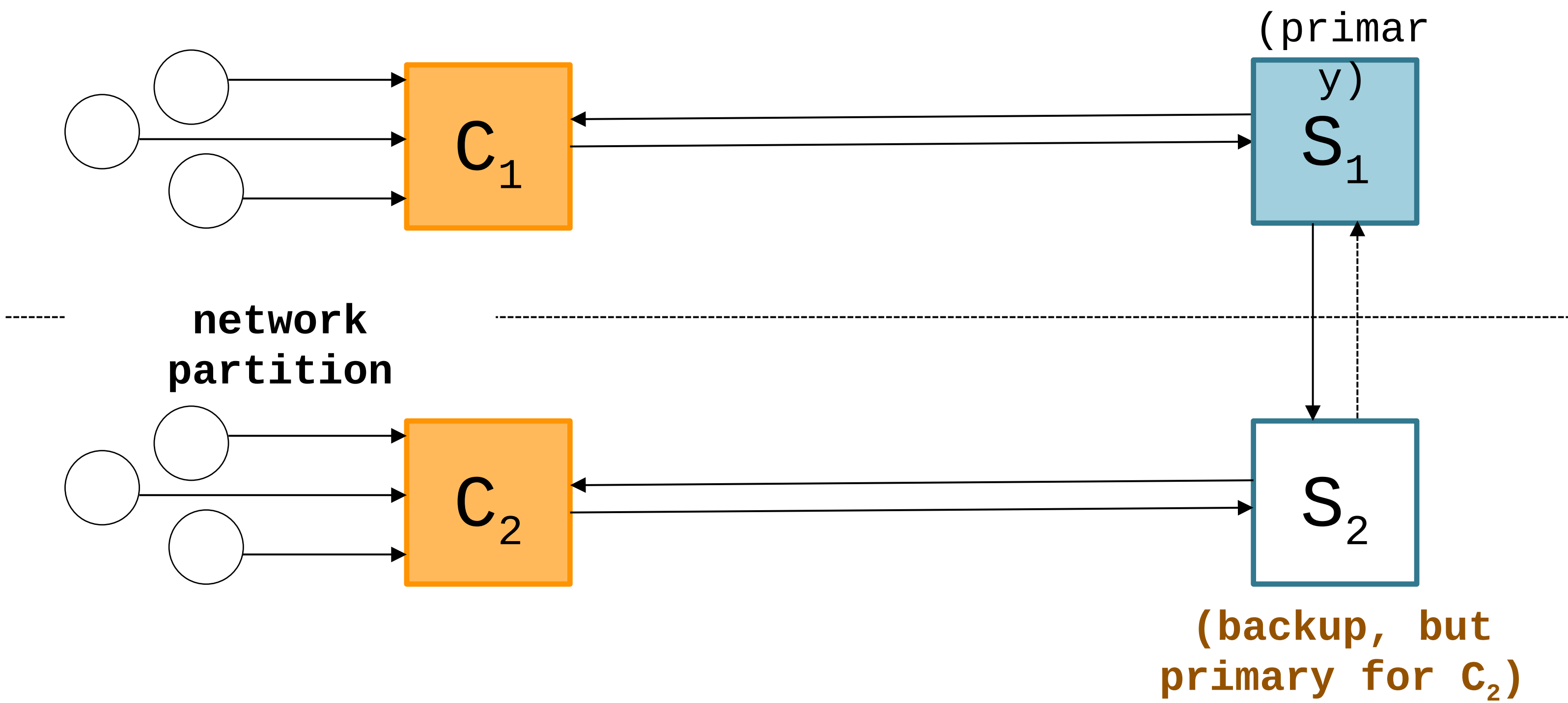
attempt: coordinators communicate with primary servers, who communicate with backup servers

Multiple Coordinators + the Network = Problems



attempt: coordinators communicate with primary servers, who communicate with backup servers

Multiple Coordinators + the Network = Problems



C_1 and **C_2** are using different primaries;

S_1 and **S_2** are no longer **consistent**

View Server

The view server keeps a table that maintains a sequence of "view"

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

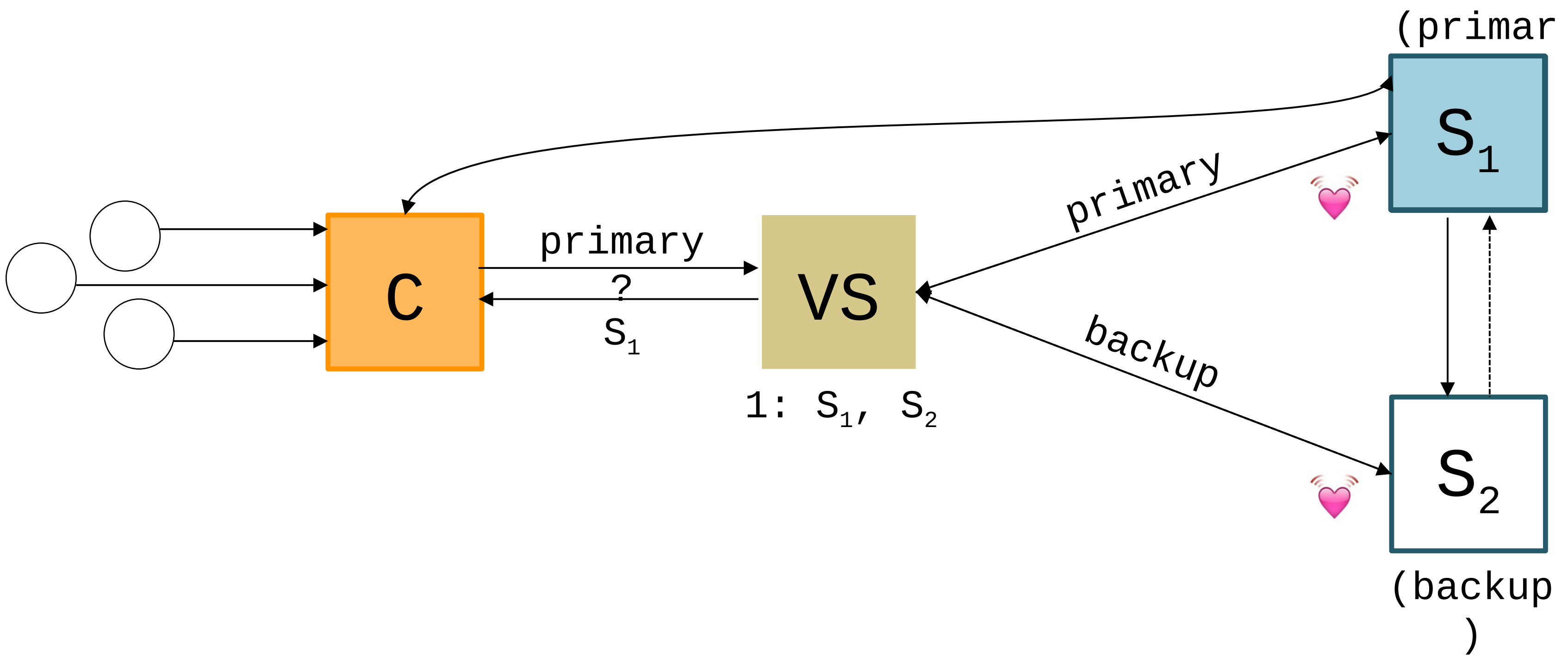
The view server alerts each server as to whether it's the primary or the backup

Upon receiving any updates, the primary will receive an ACK from the backup before responding to the view server (just as before)

Coordinators make requests to the view server asking who is primary

- Coordinators then contact the primary

View Server



Use a **view server**, which determines which replica is the primary

View Server

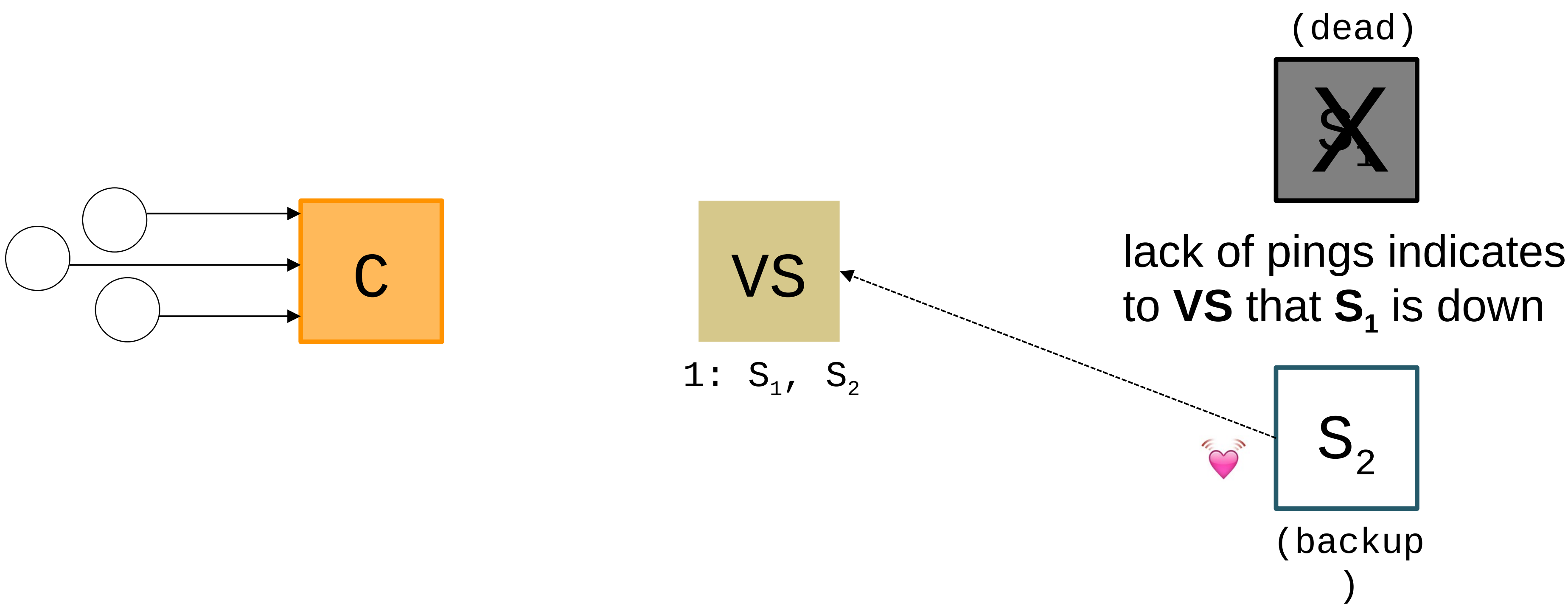
To discover failures

- Replicas ping to the view server
- If view server misses N pings in a row, it deems a server to be dead

Basic failure (actual worker crash):

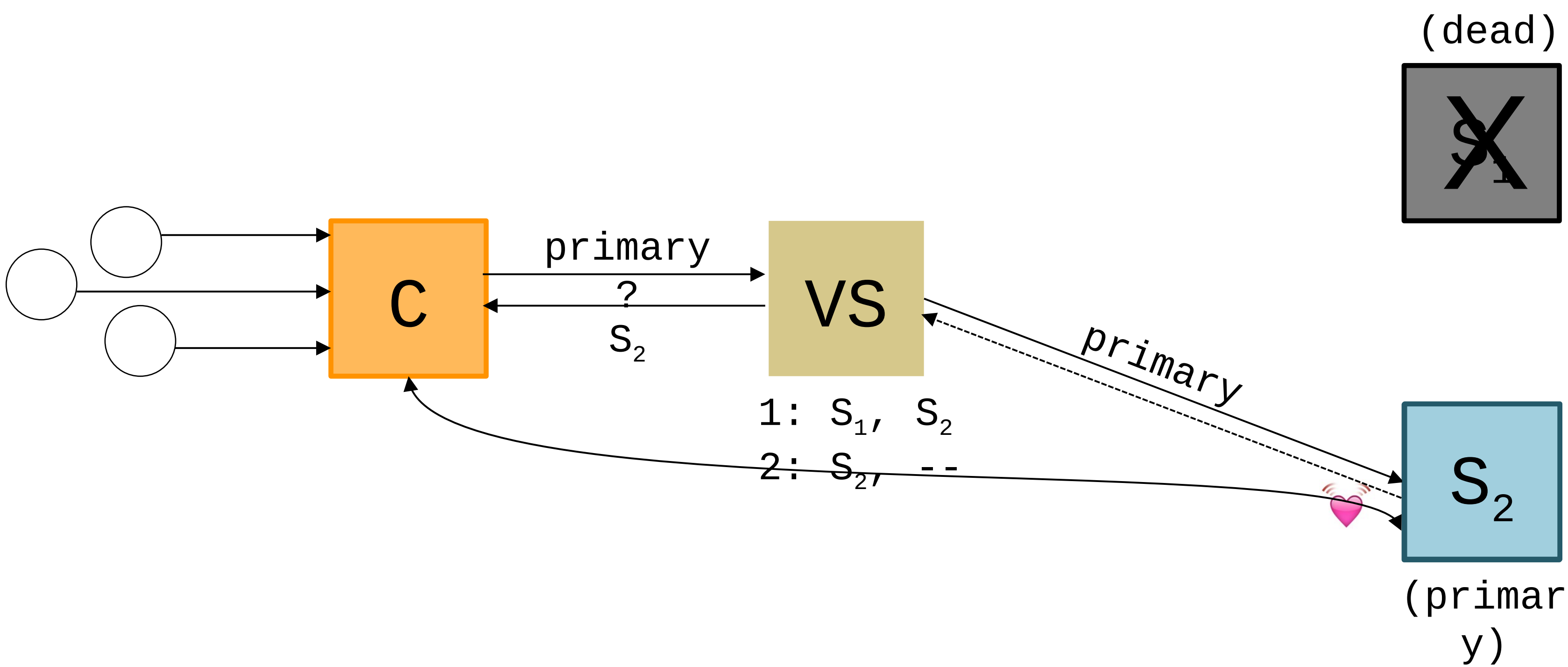
- 1. Primary fails; pings cease
- 2. View server lets S2 know it's primary, and it handles any client requests
    - Before S2 knowing it's the primary, it will simply reject requests that come directly from the coordinator
- 3. View server will eventually recruit a new idle server to act as backup

Failure of Primary



Use a **view server**, which determines which replica is the primary

Failure of Primary



before S_2 knows it's primary, it will reject any requests from clients

Rules when Facing Network Partitions

Primary must wait for backup to accept each request

Non-primary must reject direct coordinator requests

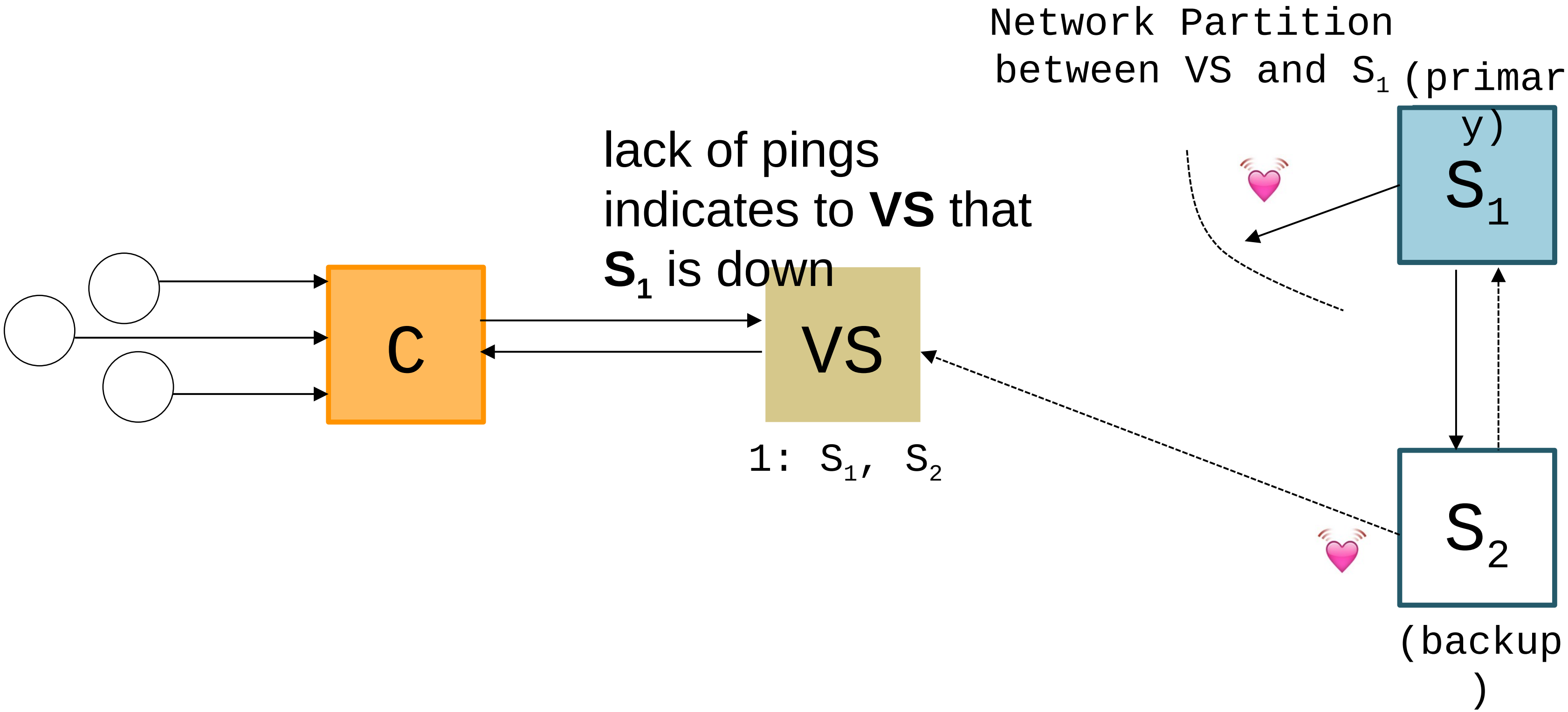
- That's what happened in the earlier failure, in the interim between the failure and S2 hearing that it was primary

Primary must reject forwarded requests

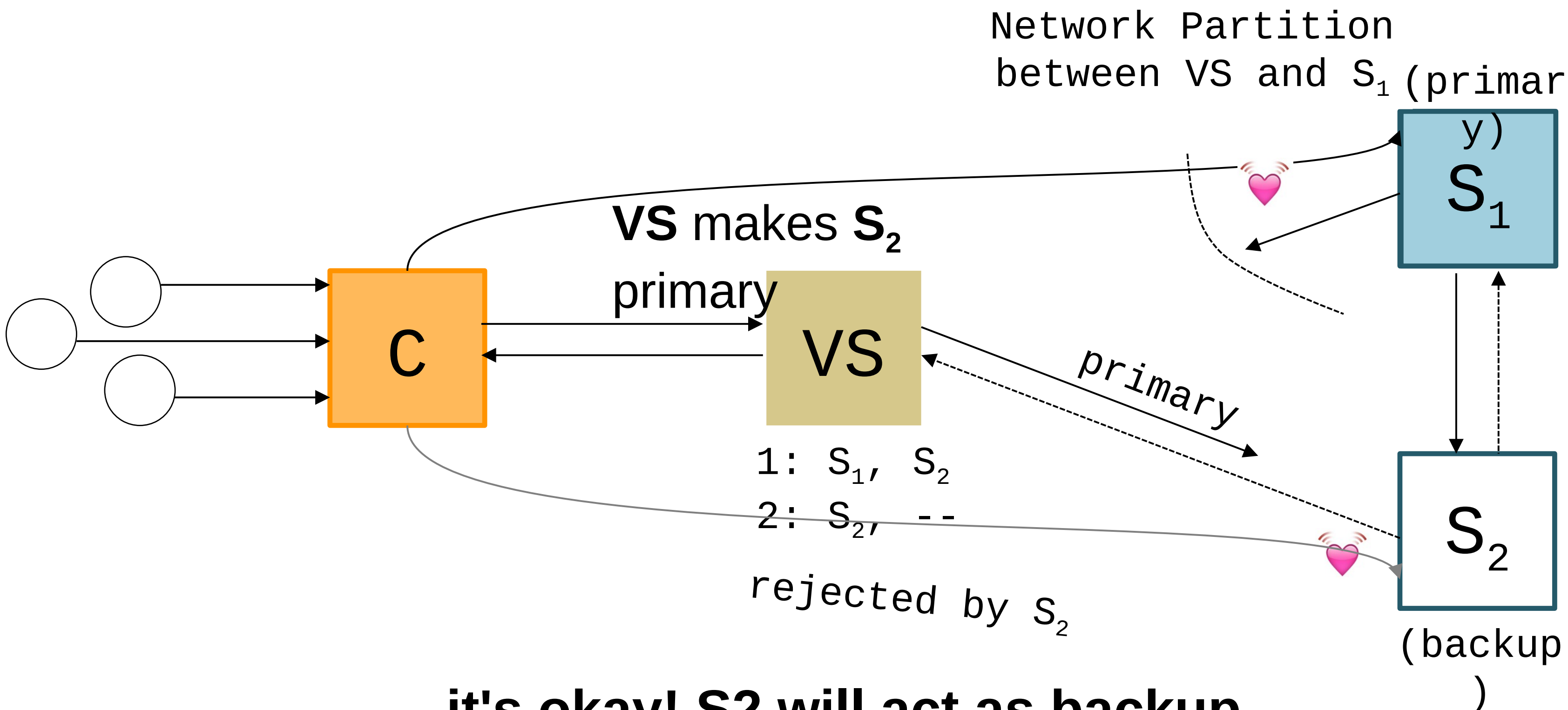
- i.e., it won't accept an update from the backup

Primary in view *i* must have been primary or backup in view *i-1*

Network Partitions



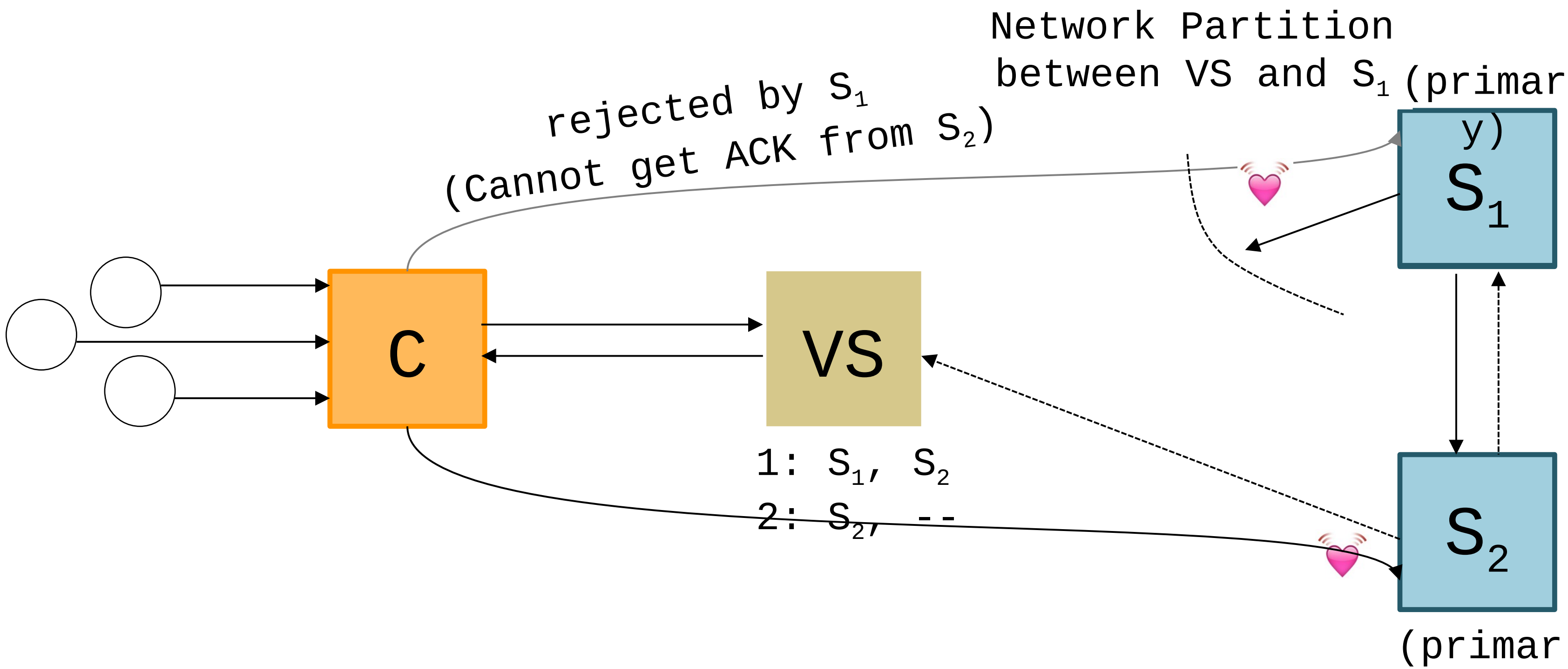
Network Partitions



it's okay! S2 will act as backup
(accept updates from S1, reject coordinator's requests)

problem: what happens before $\mathbf{S_2}$ knows it's the primary?

Network Partitions



also okay! S1 won't be able to act as primary
(can't accept client requests because it won't get ACKs from S2)

problem: what happens after S2 knows it's the primary, but S1 also thinks it is?

Consider S_1 being Partitioned from the VS

Before S2 hears about View #2:

- S1 can process operations from coordinators, S2 will accept forwarded requests
- S2 will reject operations from coordinators who have heard about view #2

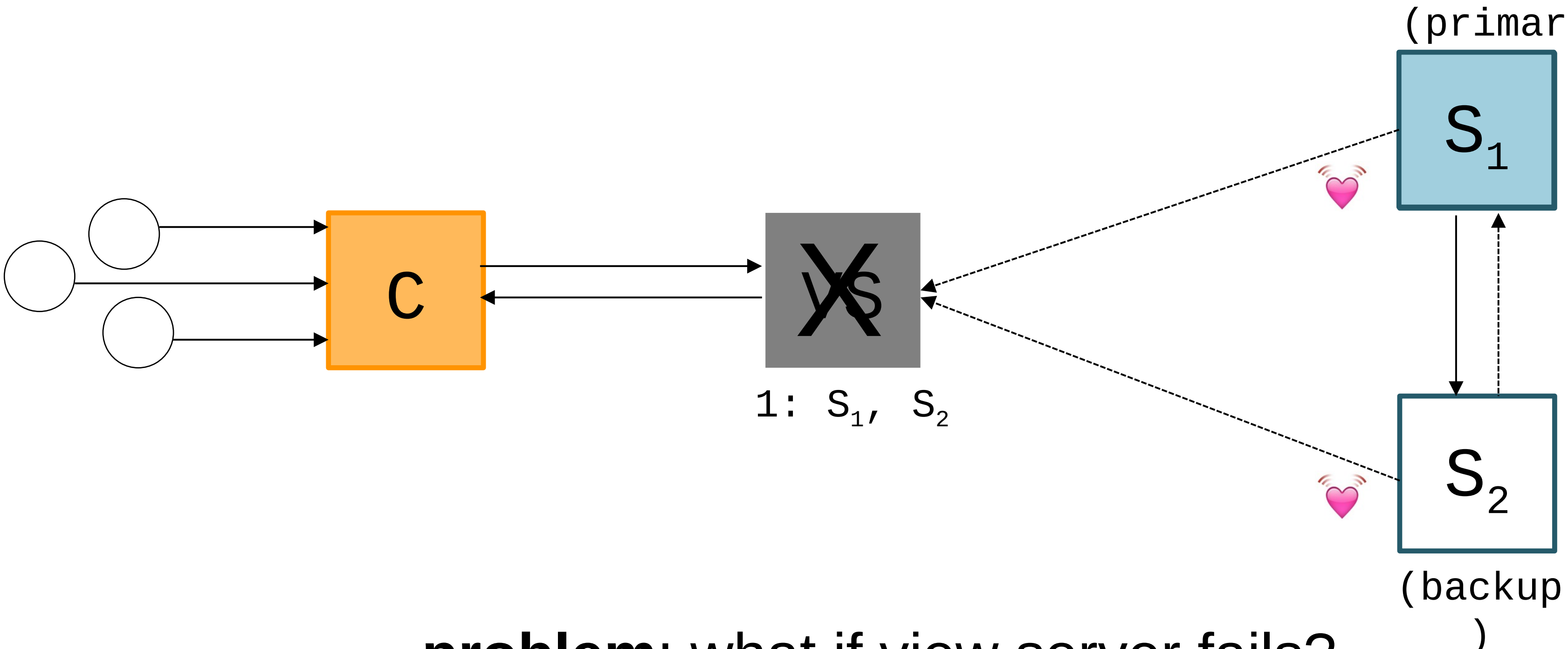
After S2 hears about View #2:

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

The commit point of switch-over:

- When S2 hears about View #2

What if view server fails?



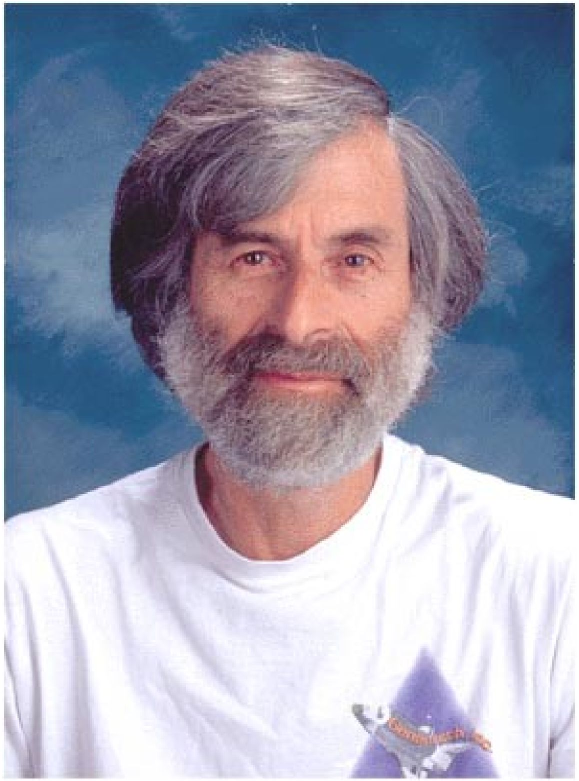
problem: what if view server fails?

Now, we need **Paxos**

Paxos

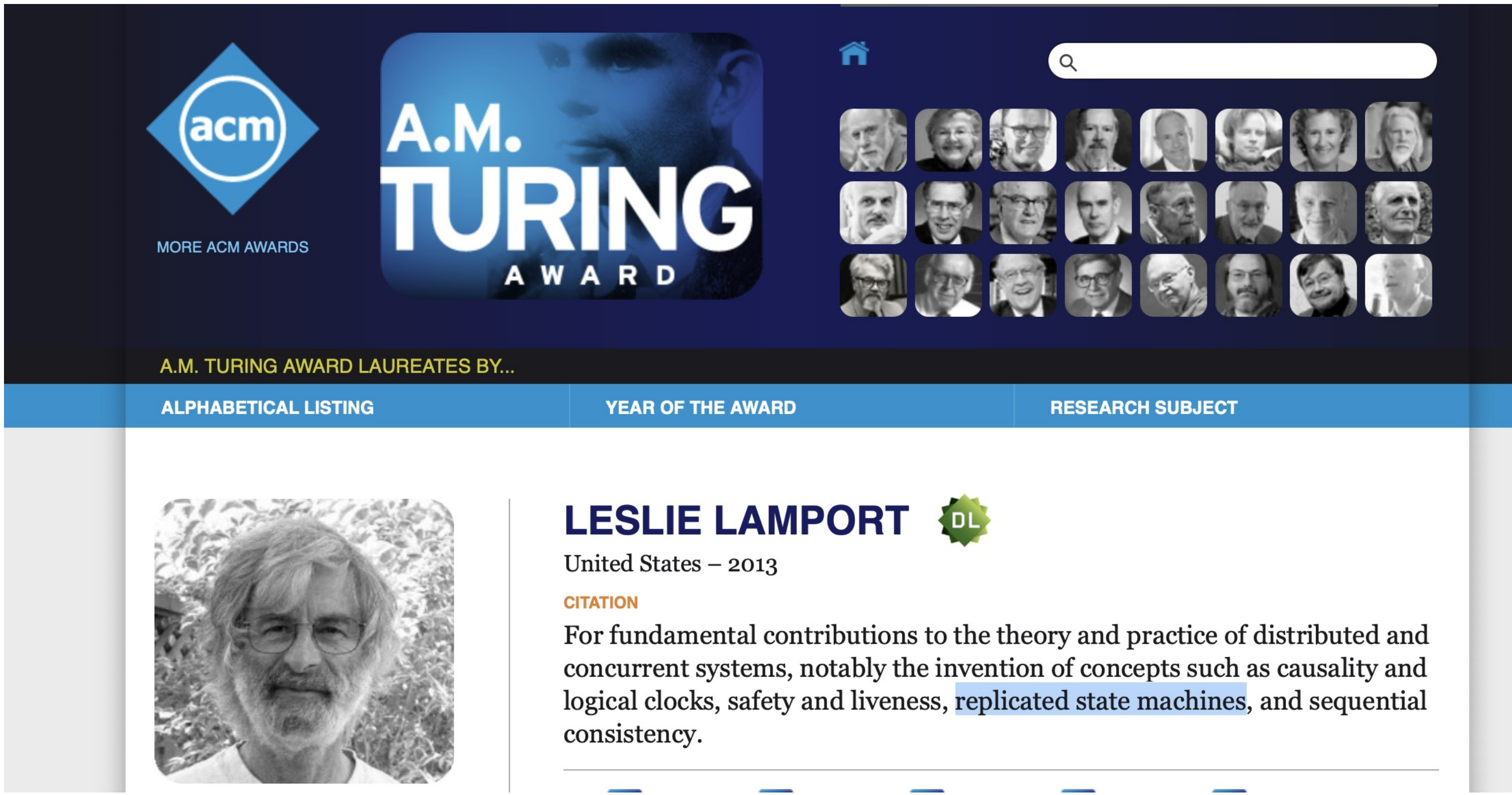
Distributed consensus mechanism

Leslie Lamport



http://ipads.se.sjtu.edu.cn/courses/cse-g/2012f/Schedule_files/paxos-simple.pdf

Paxos solves the consensus problem



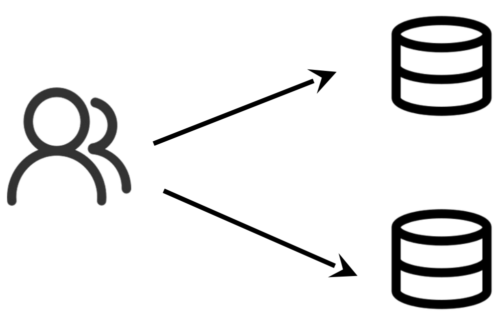
Warmup: Quorum

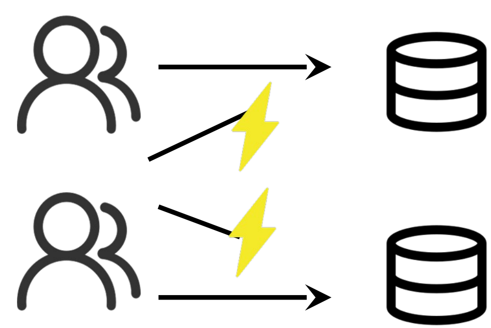
Straw-man of agree on a single value with 2 replications

- Clients send requests to both servers
- Tolerating faults: if one server is down, clients send to the other

Tricky case: what if there's a network partition?

- Each client thinks the other server is dead, keeps using its server
- Bad situation: not single-copy consistency!





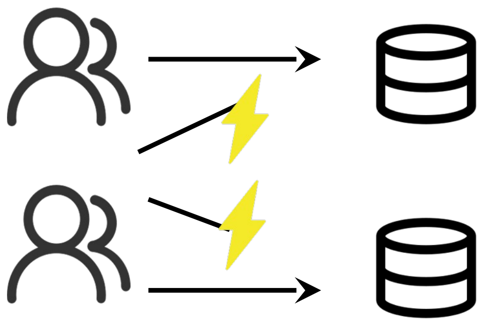
Handling Network Partitions

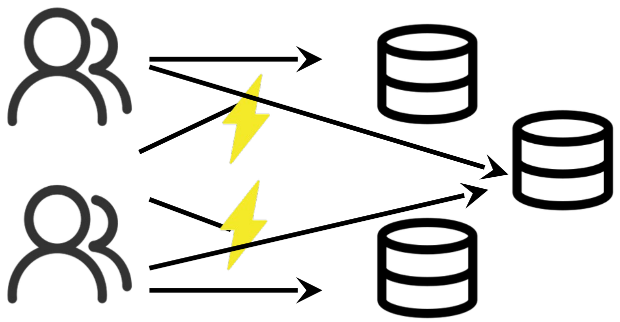
Issue: Clients may disagree about what servers are up

- Hard to solve with 2 servers, but possible with 3 servers

Idea: require a majority servers to perform operation

- In case of 3 servers, 2 form a majority
- If client can contact 2 servers, it can perform operation (otherwise, wait)
- Thus, can handle any 1-server failure





Quorum

Simplified assumption: there is only one writer

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

Define separate read &write quorums: Q_r & Q_w

- Qr + Qw > Nreplicas (Why?)
  - Confirm a write after writing to at least Qw of replicas
  - Read at least Qr agree on the data or witness value

Example

- In favor of reading: $N_{replicas} = 5$, $Q_w = 4$, $Q_r = 2$
- In favor or updating: $N_{replicas} = 5$, $Q_w = 2$, $Q_r = 4$
- Enhance availability by $Q_w = N_{replicas}$ & $Q_r = 1$

Majority in Distributed Systems

Why does the majority rule work?

- Any two majority sets of servers overlap
- Suppose two clients issue operations to a majority of servers
- Must have overlapped in at least one server, will help ensure single-copy

When is it OK to reply to client?

- Must wait for majority of replicas to reply
- Otherwise, if a minority crashes, remaining servers may continue without op

Remaining question: what if multiple writers exist?

Single-decree Paxos

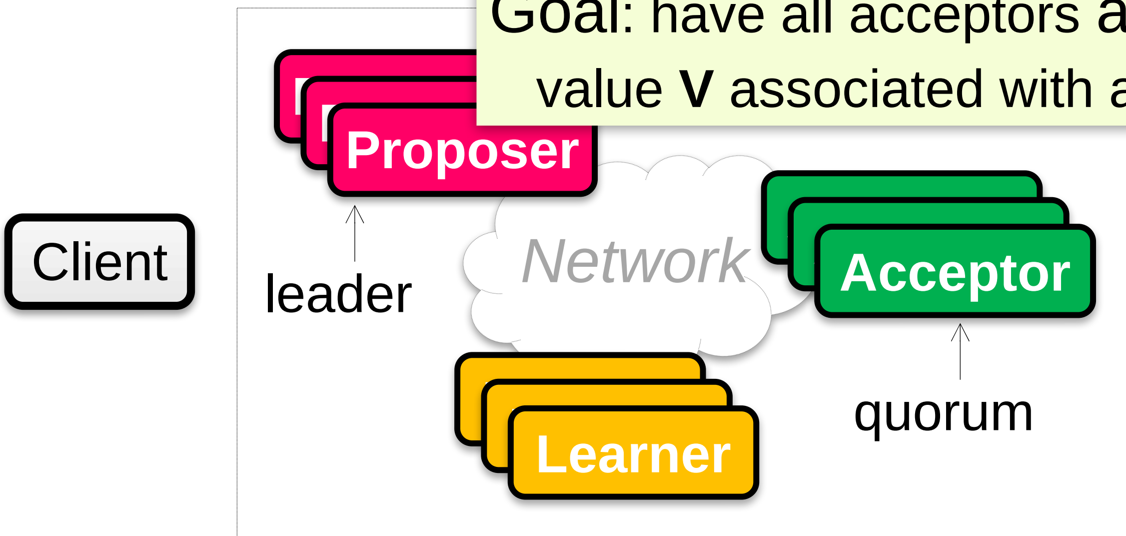
Agree on a single value

Paxos' properties: correct + fault-tolerance

- *No guaranteed* termination (i.e., lack of availability guarantee)

Goal: have all acceptors agree to a value **V** associated with a proposal

E.g., a view

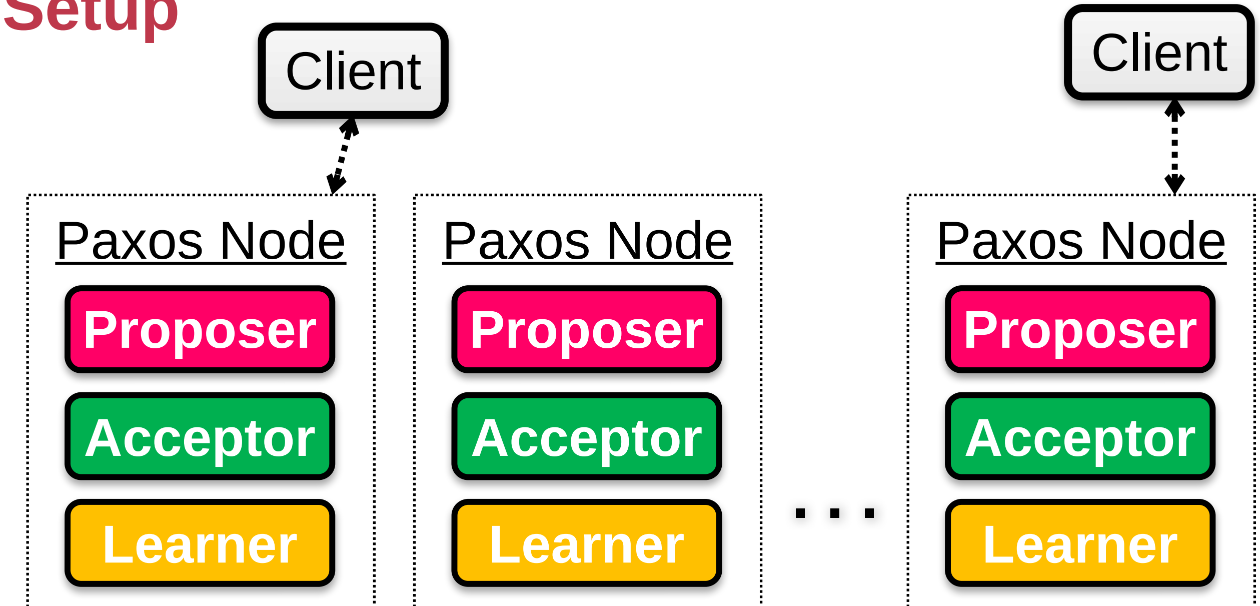


Paxos: one machine may serve several roles

Paxos Players

Client	makes a request
Proposer	Get a request and run the protocol **Leader** = elected Coordinator
Acceptor	Remember the state of the protocol **Quorum** = any majority of Acceptors
Learner	When agreement has been reached, a Learner executes the request and/or sends a response back to the Client

Paxos Setup


Client
Client
Paxos Node
Proposer
Acceptor
Learner
Paxos Node
Proposer
Acceptor
Learner
. . .
Paxos Node
Proposer
Acceptor
Learner

General Approach

One proposer decides to be the leader (optional)

Leader proposes a value and solicits acceptance from acceptors (majority)

Leader announces result or try again

What if >1 proposers become leaders simultaneously?

What if there is a network partition?

What if a leader crashes in the middle of solicitation?

What if a leader crashes after deciding but before announcing results?

. . .

Political Science 101

Paxos has rounds; each round has a unique ID (N, i.e., proposal number)

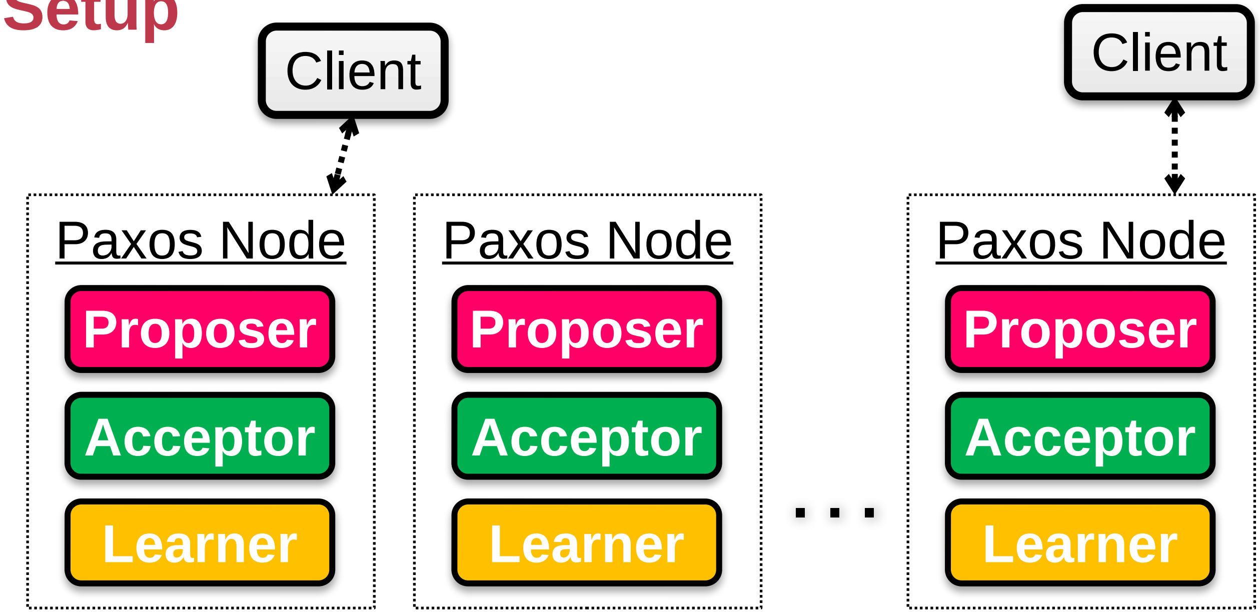
Rounds are asynchronous

- Time synchronization not required
- If you are in round *j* and hear a message from round *j+1*, abort everything and move over to round *j+1*
- Use timeouts; may be pessimistic

Each round itself broken into phases

- Phases are also asynchronous

Paxos Setup



$\mathbf{N_a}$: highest proposal number accepted

$\mathbf{V_a}$: accepted value of $\mathbf{N_a}$

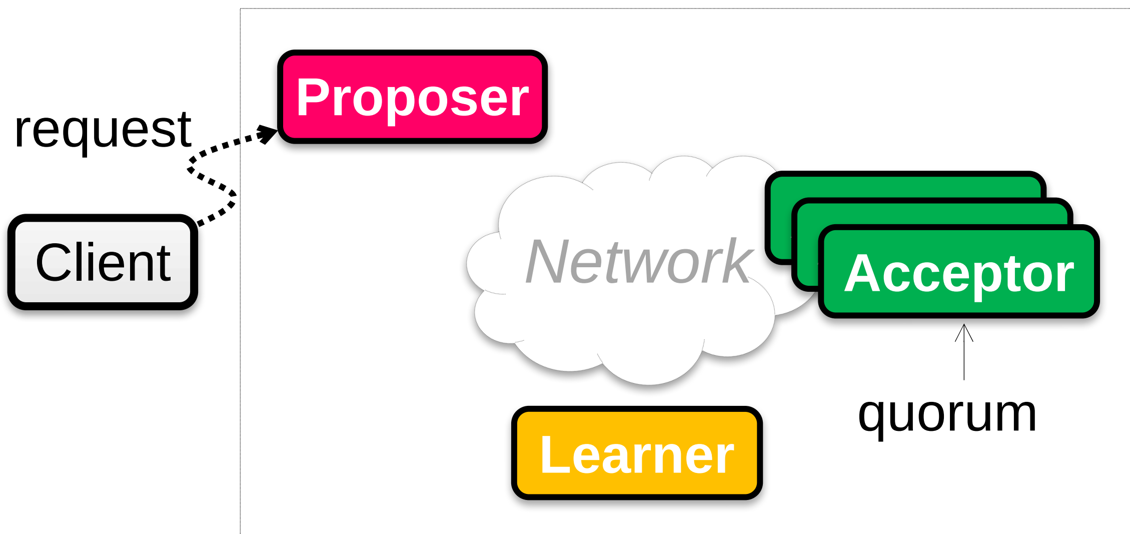
$\mathbf{N_h}$: highest proposal number seen

$\mathbf{M_n}$: my proposal number

each round of Paxos, each Node

Paxos in Action: Phase 0

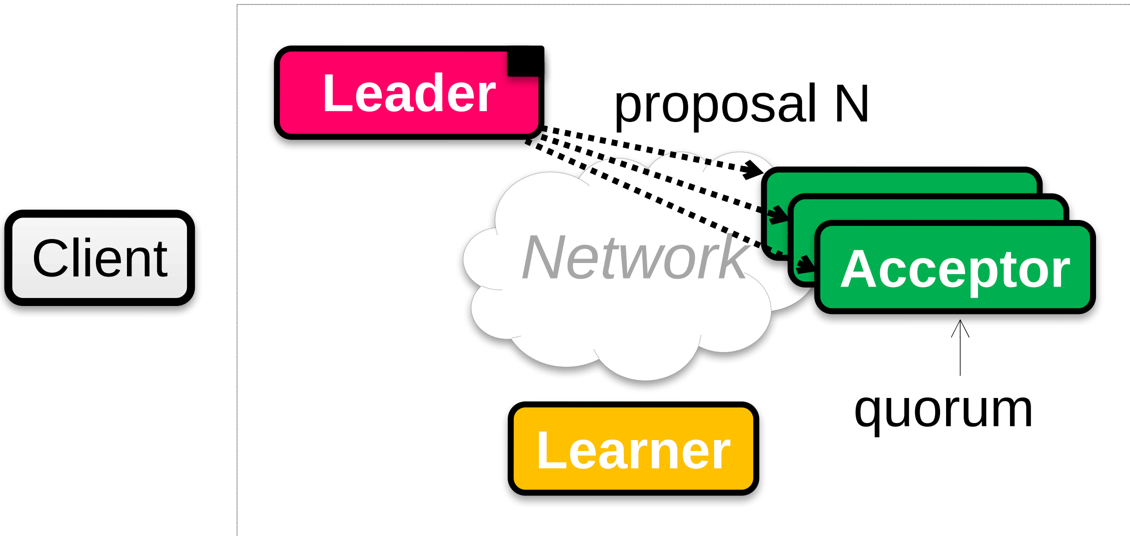
Client sends a request to a proposer



Paxos in Action: Phase 1a (Prepare)

Leader creates a proposal N and send to quorum

N is greater than **any** previous proposal number seen by this proposer

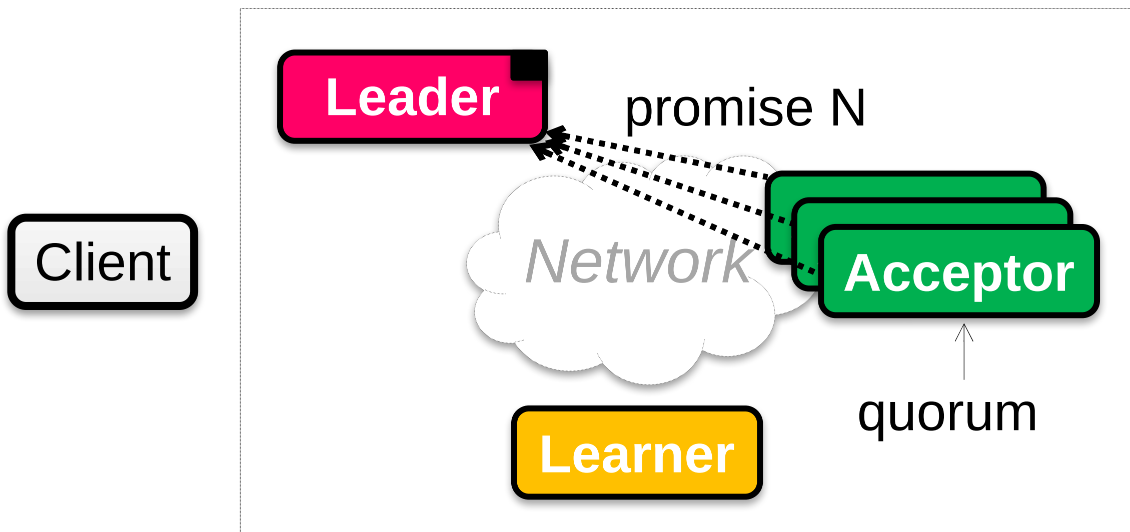


Paxos in Action: Phase 1b (Prepare)

Acceptor: **if** proposal ID > any previous proposal

1. reply with the highest past proposal number and value
2. promise to ignore all IDs < N

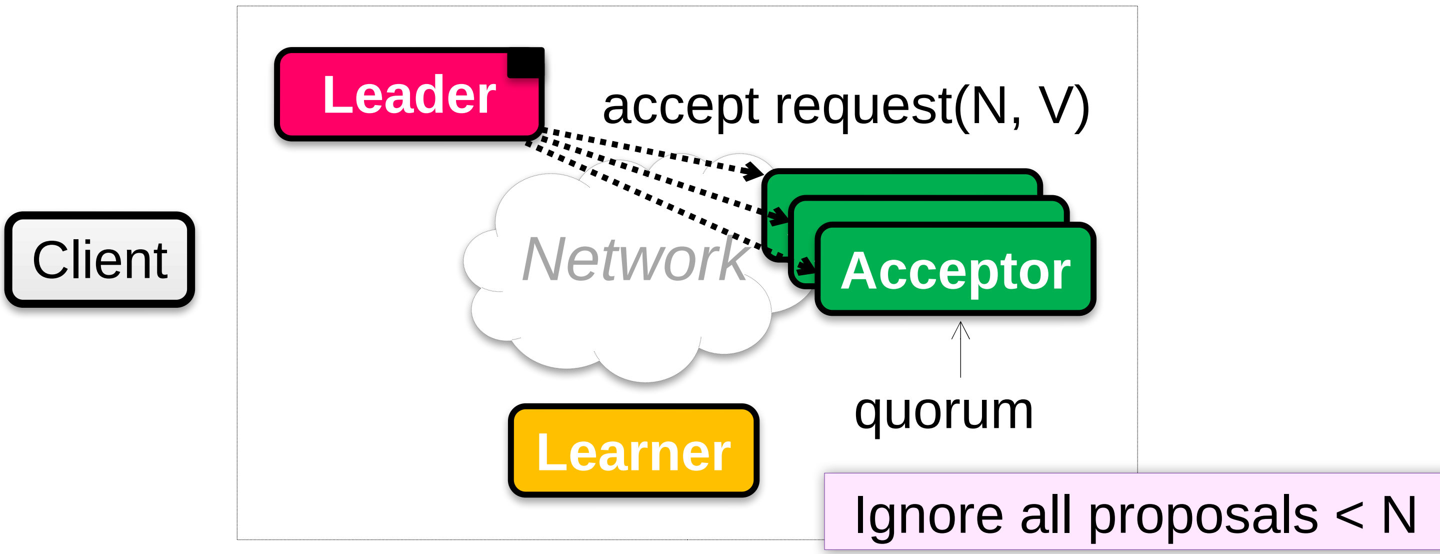
else ignore (proposal is rejected)



Paxos in Action: Phase 2a (Accept)

Leader: if receive enough promise

1. **set a value V to the proposal V, if any accepted value returned, replace V with the returned one**
2. **send accept request to quorum with the chosen value V**

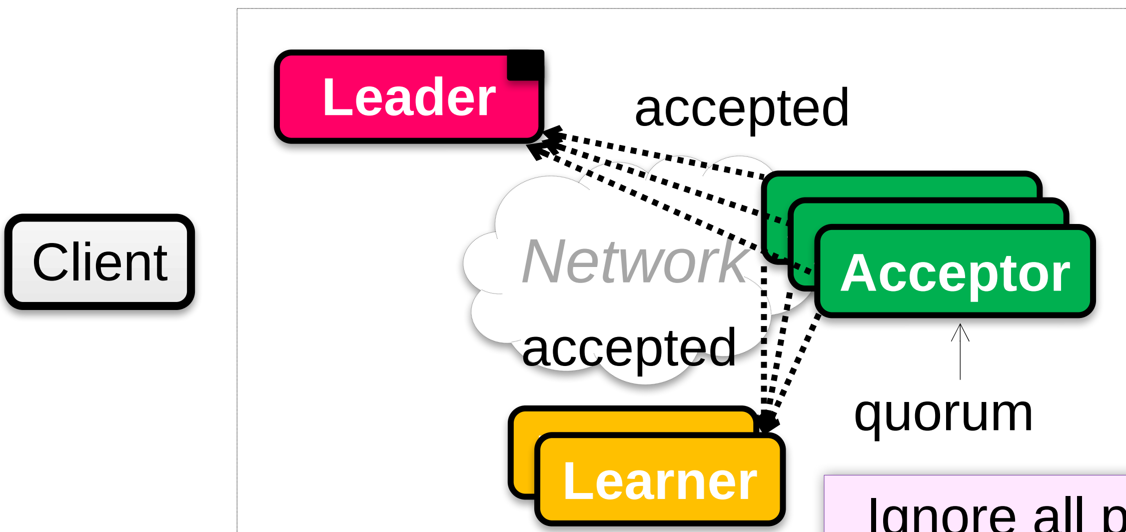


Paxos in Action: Phase 2b (Accept)

Acceptor: if the promise still holds

1. **register the value V**
2. **send accepted message to Proposer/Learners**

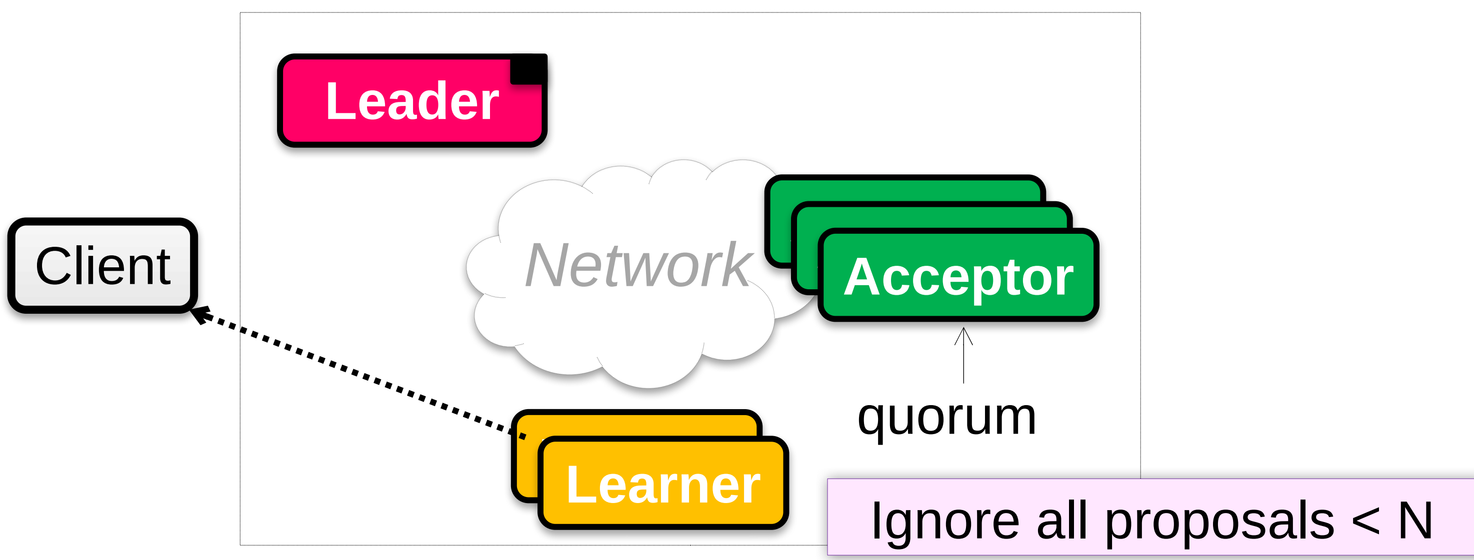
else ignore the message



Ignore all proposals < N

Paxos in Action: Phase 3 (Learn)

Learner: responds to Client and/or take action on the request



N_a: highest proposal number accepted
V_a: accepted value of **N_a**
N_h: highest proposal number seen
M_n: my proposal number

Paxos Pseudo-code

A node decides to be Leader

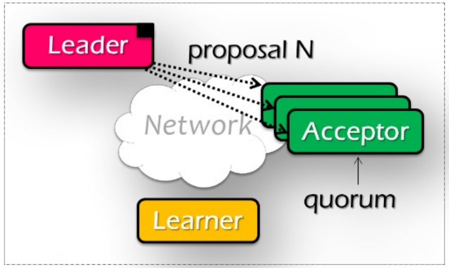
Leader chooses $M_n > N_h$

Leader sends <proposal, M_n> to all nodes

Acceptor receives <proposal, N>

if $N < N_h$
    reply <promise-reject>
else
    $N_h = N$
    reply <promise-ok, N_a, V_a>

Ignore all proposals $< N_h$



N_a: highest proposal number accepted
V_a: accepted value of **N_a**
N_h: highest proposal number seen
M_n: my proposal number

Paxos Pseudo-code

If Leader gets promise-ok from a majority

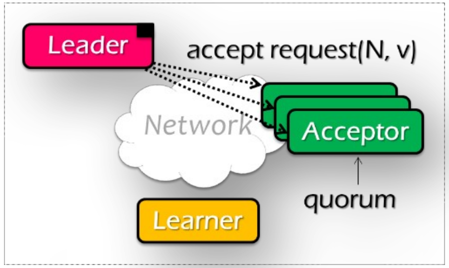
if V != null, V = the value of the highest **N_a** received
if V = null, then Leader can pick any **V**
send <accept, **M_n**, **V**> to all nodes

If Leader fails to get majority promise-ok

delay and restart Paxos

Upon receiving <accept, N, V>

if $N < N_h$
    reply <accept-reject>
else
    $N_a = N$; $V_a = V$; $N_h = N$;
    reply <accept-ok>



Paxos Pseudo-code

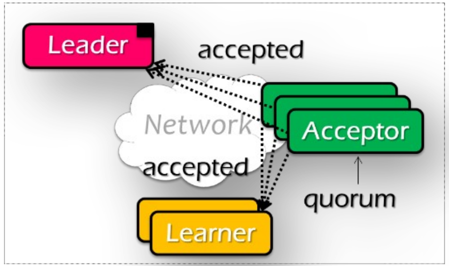
N_a: highest proposal number accepted
V_a: accepted value of **N_a**
N_h: highest proposal number seen
M_n: my proposal number

If Leader gets accept-ok from a majority

send <decide, V_a> to all nodes

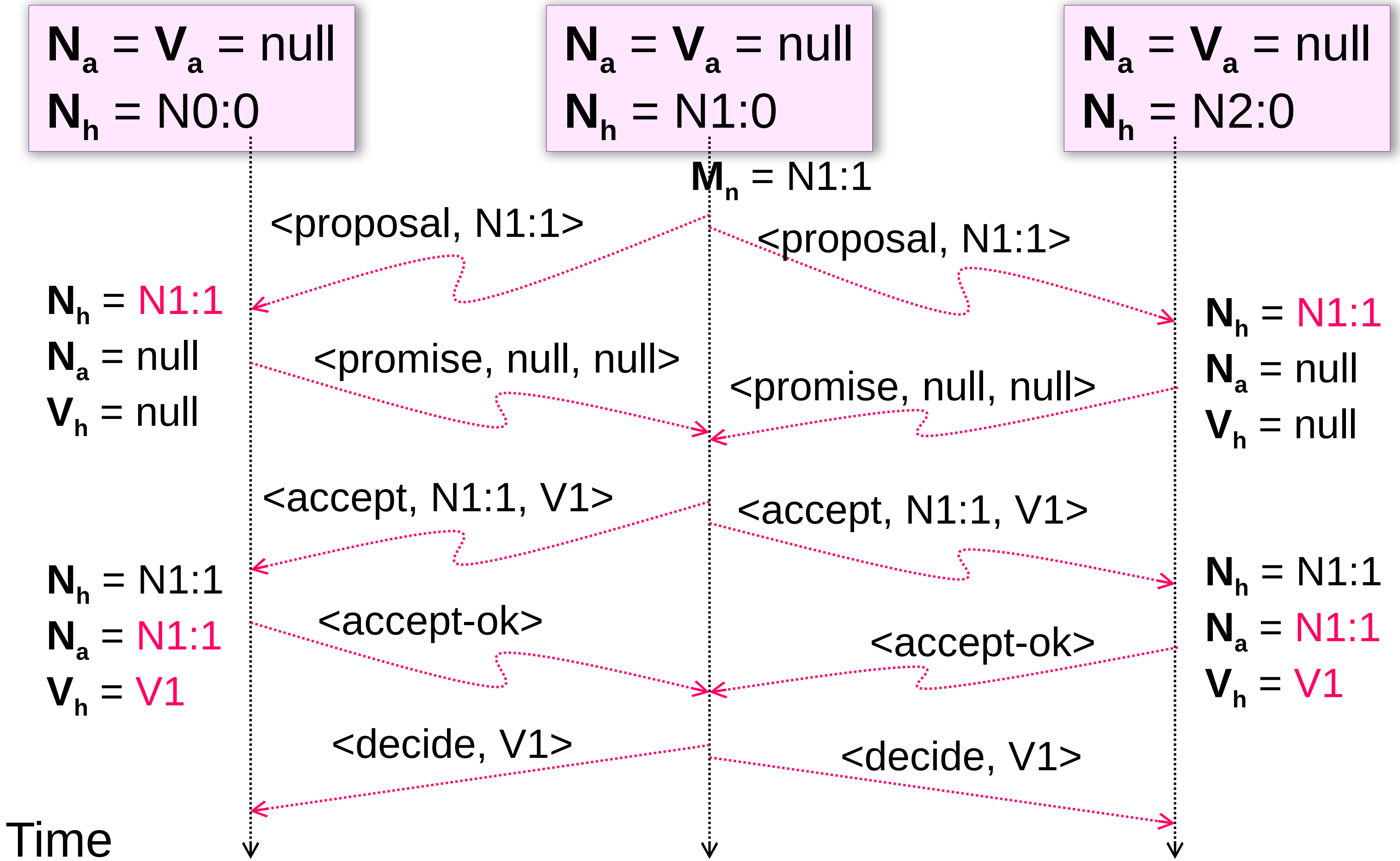
If Leader fails to get majority accept-ok

delay and restart Paxos



Paxos Pseudo-code

N_a: highest proposal number accepted
V_a: accepted value of N_a
N_h: highest proposal number seen
M_n: my proposal number



N_a: highest proposal number accepted
V_a: accepted value of **N_a**
N_h: highest proposal number seen
M_n: my proposal number

Inside of Paxos

Why setups multiple acceptors?

- Failure of the single acceptor halts decision

Why not accepts the first proposal and rejects the rest?

- Leader dies
- Multiple leaders result in no majority accepting

What if more than one leader is active?

- Can both leaders see a majority of promises?

N_a: highest proposal number accepted
V_a: accepted value of N_a
N_h: highest proposal number seen
M_n: my proposal number

Inside of Paxos

When is the value V chosen?

- Leader receives a majority <promise, …>
- A majority acceptors receive <accept, N, V>
- Leader receives a majority <accepted, …>

What if acceptor fails after sending promise?

- Must remember N_h

What if acceptor fails after receiving accept?

- Must remember N_h and N_a V_a

What if leader fails while sending accept?

- Propose M_n again

N_a: highest proposal number accepted
V_a: accepted value of N_a
N_h: highest proposal number seen
M_n: my proposal number

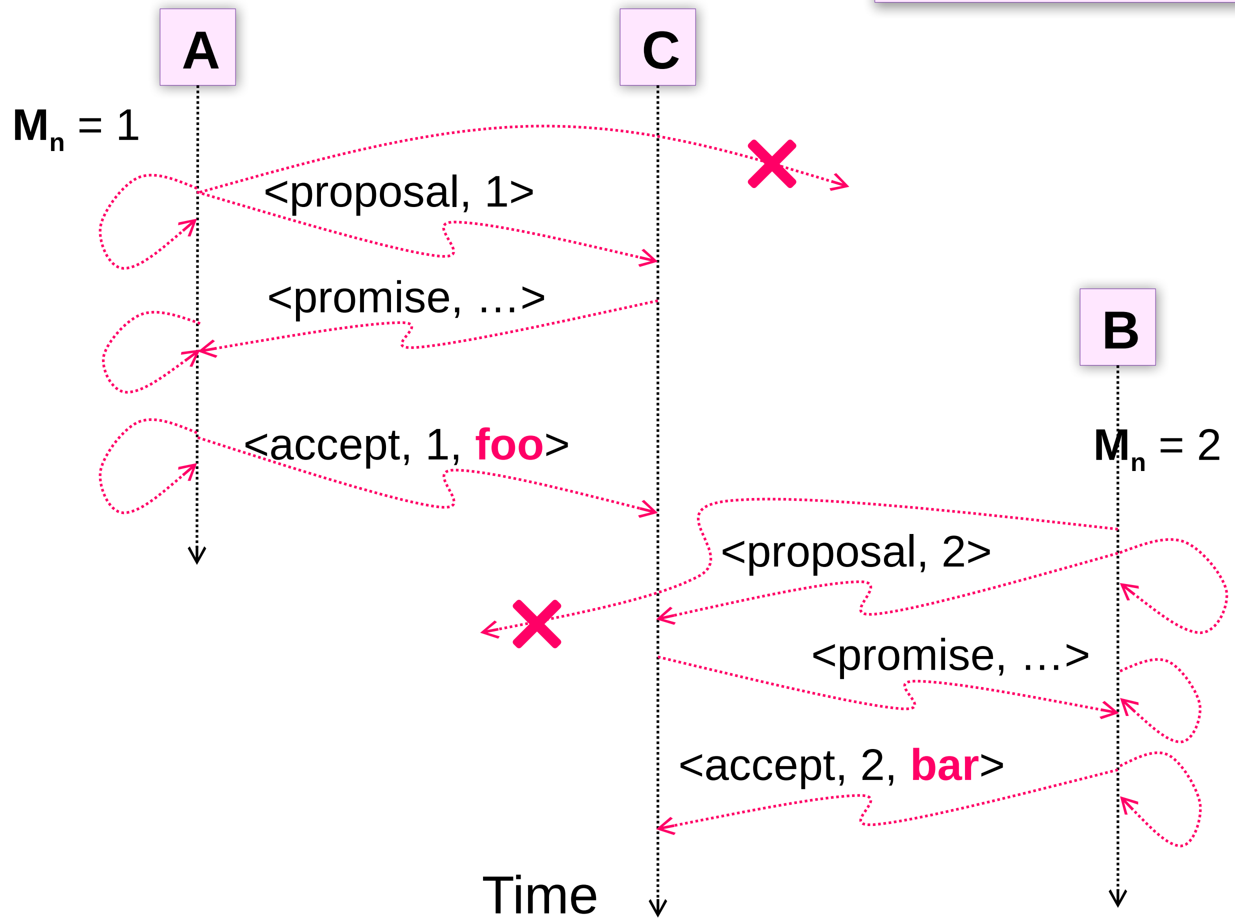
Question

Suppose that the acceptors are A, B, and C. A and B are also proposers. How does Paxos ensure that the following sequence of events can't happen? What actually happens, and which value is ultimately chosen?

- **A sends <prepare, 1> requests with proposal number 1, and gets responses from A, C**
- **A sends <accept,1, "foo"> to A and C and gets responses from both. Because a majority accepted, A thinks that "foo" has been chosen. However, A crashes before sending an <accept, 1, "foo"> to B**
- **B sends <prepare, 2> messages with proposal number 2, and gets responses from B and C**
- **B sends <accept, 2, "bar"> messages to B and C and gets responses from both, so B thinks that "bar" has been chosen**

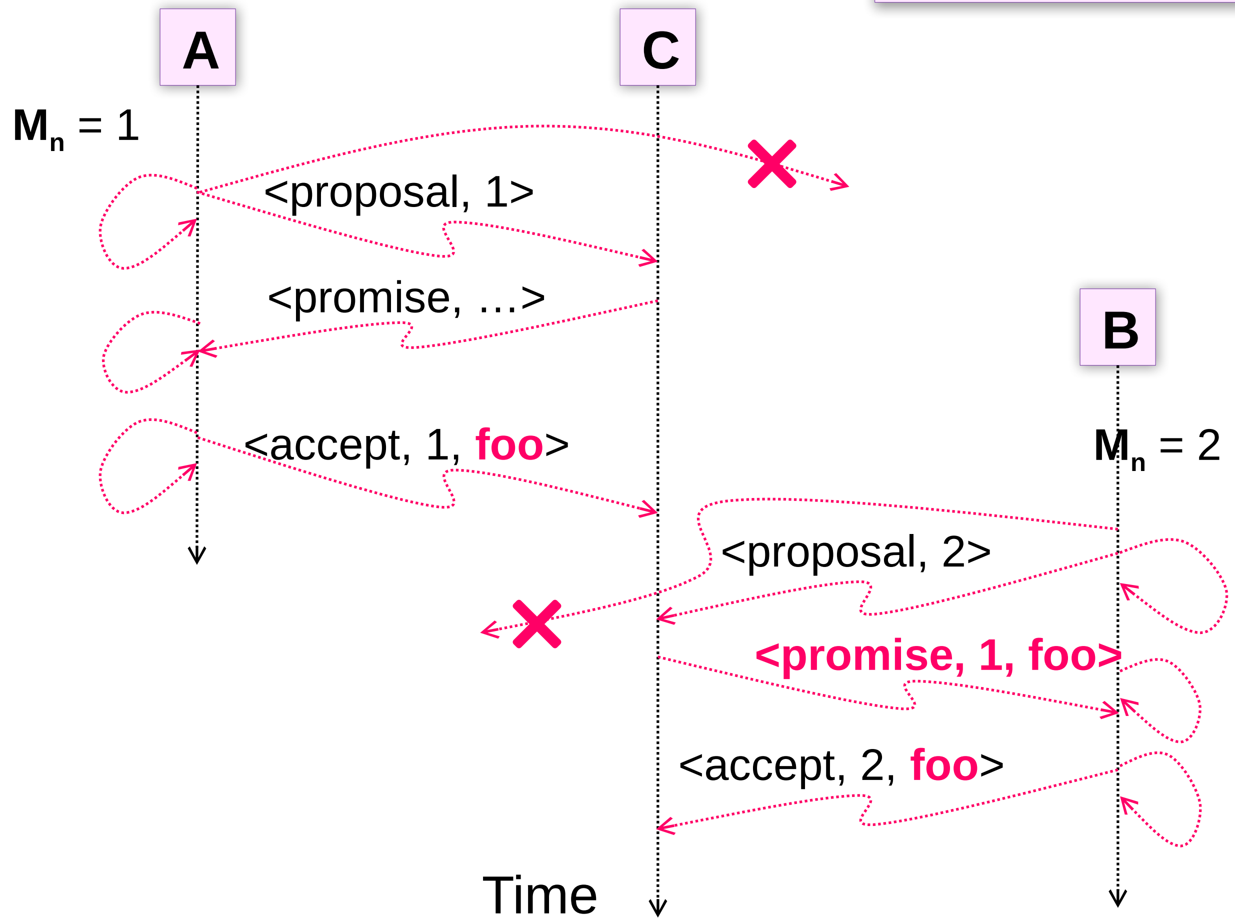
Question

$\mathbf{N_a}$: highest proposal number accepted
$\mathbf{V_a}$: accepted value of $\mathbf{N_a}$
$\mathbf{N_h}$: highest proposal number seen
$\mathbf{M_n}$: my proposal number



Question

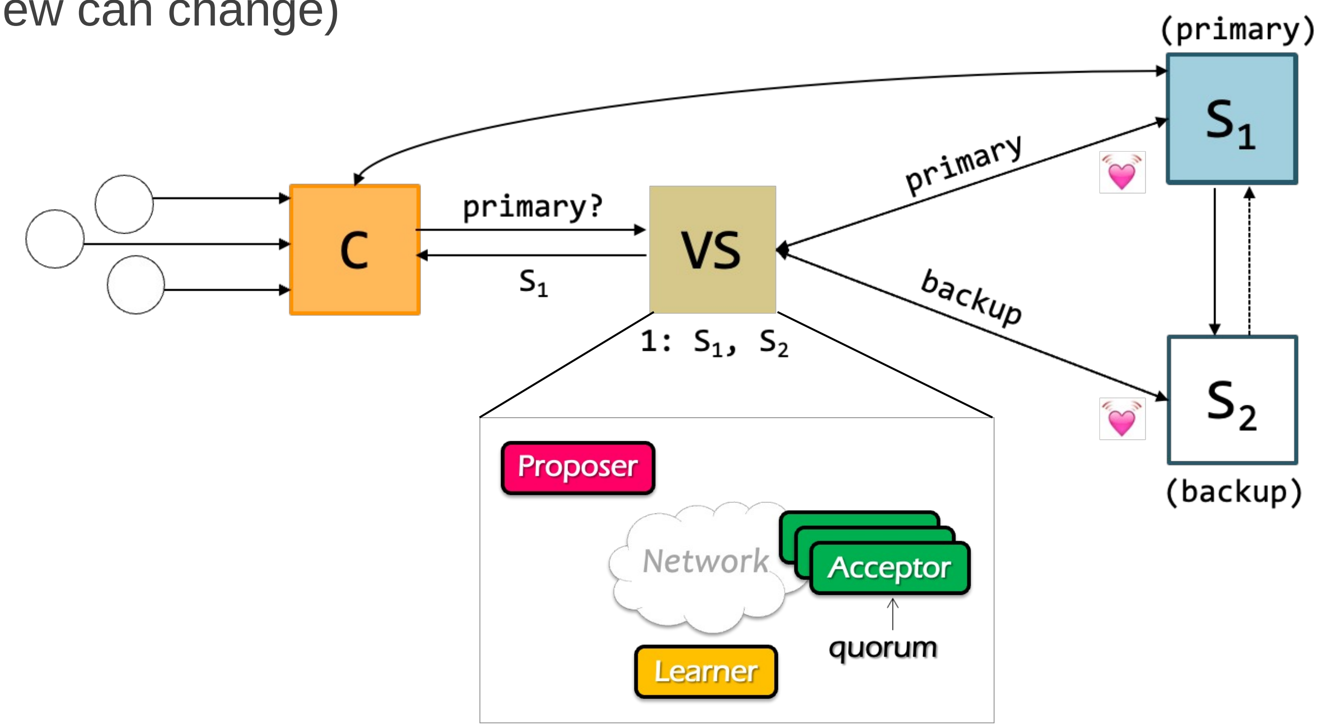
$\mathbf{N_a}$: highest proposal number accepted
$\mathbf{V_a}$: accepted value of $\mathbf{N_a}$
$\mathbf{N_h}$: highest proposal number seen
$\mathbf{M_n}$: my proposal number



Question

Does single-decree Paxos works fine for our view server example?

– When can it go wrong? Accepting a single value is not enough (because view can change)



Paxos Summary

Paxos allows us to ensure consistent (total) ordering over a set of events in a group of nodes

- Events = commands / actions / state updates

Each machine will have the latest state or a previous version of the state

Multi-Paxos

Agree on a sequence of values

Multi-Paxos builds on top of the basic Paxos

Useful when agreeing on a sequences of values, examples including:

- Views in primary-backup replication
- Logs in a replicated state machine
    - i.e., use Multi-Paxos to implement RSM

The basic approach

- Run a separate instance of Paxos to agree on the value of each index
- Each instance of Paxos has its own copy of state
    - highest proposal seen
    - accepted proposal number
    - accepted proposal value

Basic Multi-Paxos

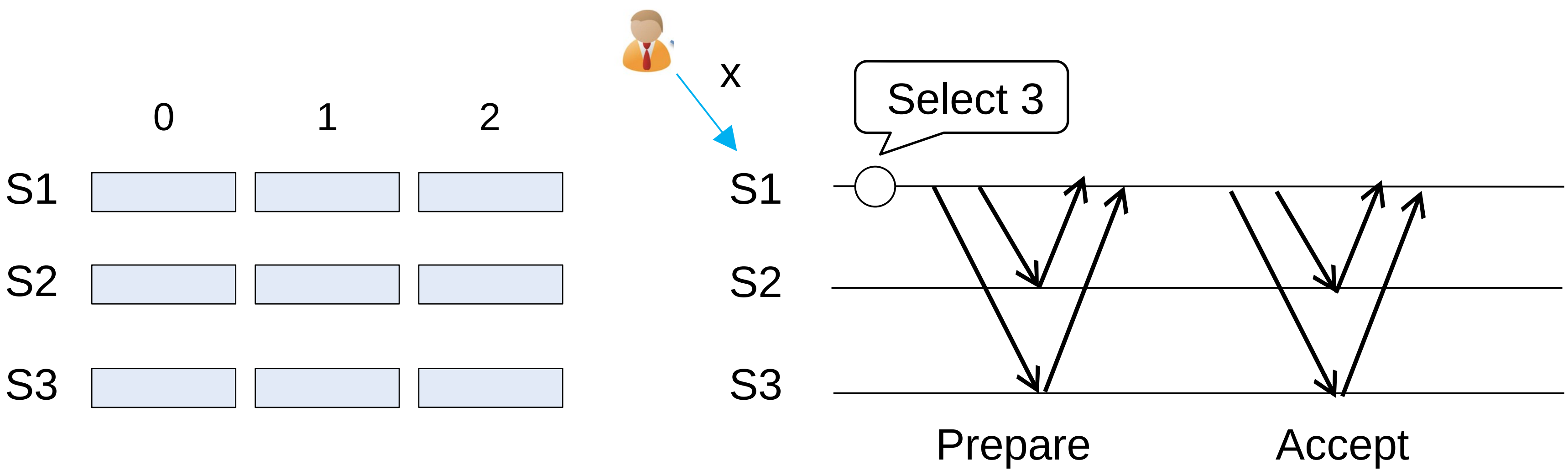
Question: what could happen if instance 3 has already gotten a value?

Server simultaneously acts as proposer, acceptor & learner

After receiving a value, the server:

① Decides where to place the value (e.g., the latest)

② Start Paxos at the decided position



Basic Multi-Paxos is inefficient

With multiple concurrent proposers, conflicts and restarts are likely

- higher load → more conflicts
- Will be slow (but still correct!)

2 rounds of RPCs for each value chosen

- Prepare, Accept

Solution

① Select a leader: most of the time, only one server can propose

② Batch prepare requests from multiple instances sent from a leader

Multi-paxos uses a distinguished proposer (leader)

Distinguished proposer (aka. leader)

- The only one that issues proposals
    - i.e., reduce proposer conflicts

Client sends the commands only to the leader

- Decides the value position

Note, single leader is *not* necessary for Multi-paxos

- E.g., if two or more servers act as proposers at the same time, the protocol still works correctly

Prepare message batching

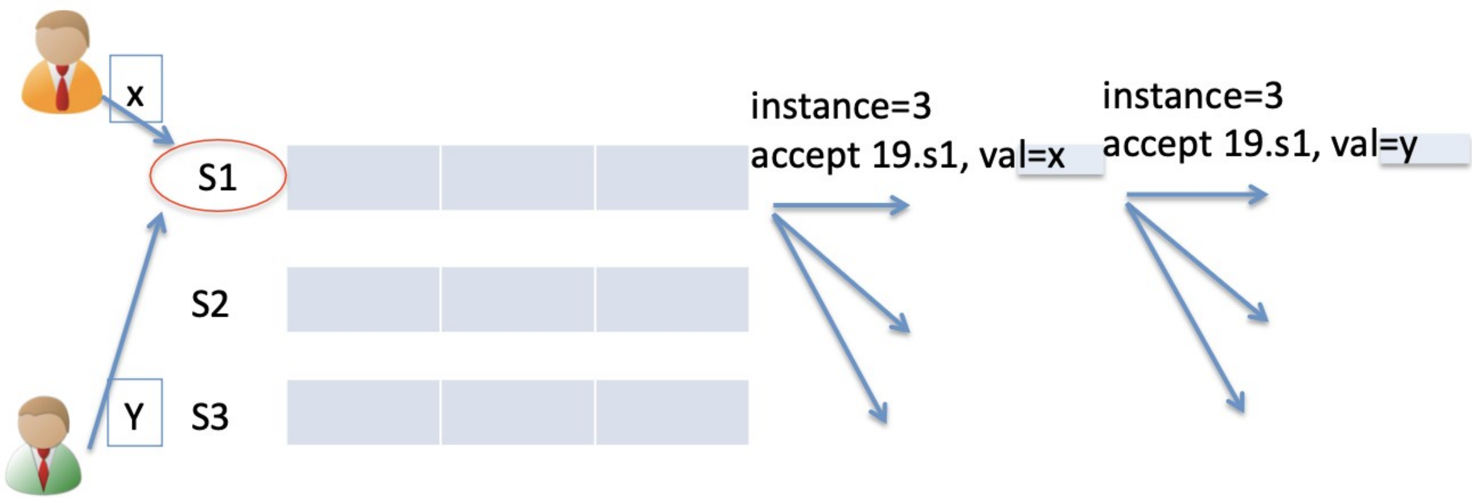
All instances of the leader share the same state

– i.e., highest proposal number seen

Can use one message to prepare for a batch of instances

Benefits of batched prepare message

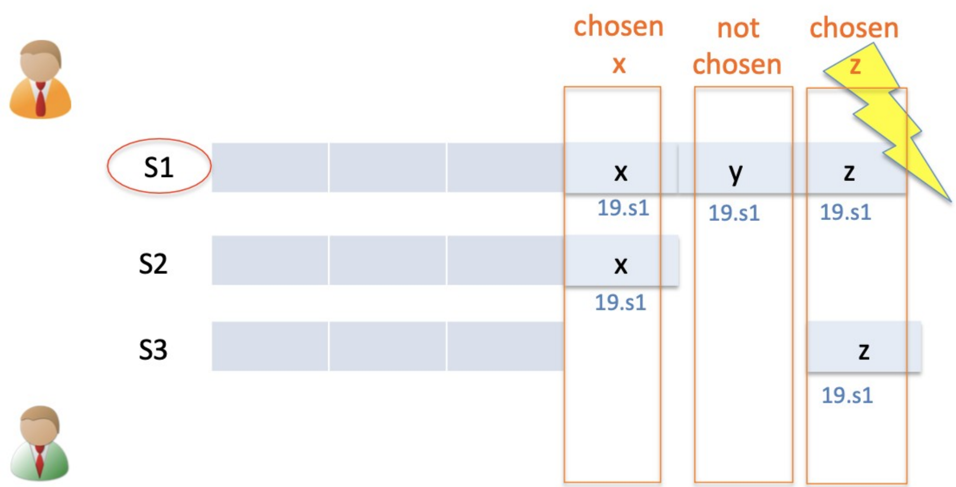
Leader only needs to run the accept phase to replicate an operation during normal time



Multi-Paxos can run multiple instances concurrently

i.e., the prefix of chosen values are not contiguous

- Need to re-run Paxos after an old leader crashes to fill the holes
- If there is no value chosen, use a no-op to fill the hole



Paxos Summary

Paxos allows us to ensure consistent (total) ordering over a set of events in a group of nodes

- Events = commands / actions / state updates

Each machine will have the latest state or a previous version of the state

Single Paxos Summary

To make a change to the system

1. Tell the **proposer**(**leader**) the event

    (NOTE: these requests may occur concurrently)

2. The **leader** picks its next highest ID and asks proposal to all the **acceptors** with that ID
3. When the majority of **acceptors** accept the proposal, accepted event are sent to **learners**
4. The **learners** do event (e.g., update system state)

Paxos for RSM

Fault-tolerant RSM requires consistent replica view

- View: <primary, backups> (e.g., <node1, node2>)

All active nodes must agree on the sequence of view changes

- <vid-1, primary, backups>, . . .

Use Paxos to agree on the <primary, backups>

- Each Paxos instance agree to a single view

  (e.g., <2, Node1, Node2>)

Paxos itself can also be used to implement RSM

- E.g., agree on multiple states
- Usually inefficient: e.g., two RTTs to agree on a value
- Multi-Paxos can batch prepares to improve the performance