Agreement in Distributed Systems: Consensus

Lecture 11

Recap of last week

- Agreement in distributed systems
  - How do we get all nodes in a distributed system to act in concert despite failures?
- Agreement Requirements
  - Safety (correctness)
    - All nodes agree on the same value
    - The agreed value X has been proposed by some node
  - Liveness (fault tolerance, availability)
    - If less than some fraction of nodes crash, the rest should still reach agreement

Recap of last week: Atomic commitment

- **Atomic commitment problem**
  - One type of agreement problem: Participants need to agree on a value, but they have specific constraints on whether they can accept any particular value.

- We looked specifically at atomic commitment in distributed databases: **How to provide atomicity (A of ACID) in the presence of failures?**
  - 2-phase commit: Safe but not live due to blocking
  - Non-blocking 3-phase commit: Live but cannot handle network partition

Consensus: Formal definition

- Problem
  - A collection of processes, Pi.
  - They propose values Vi (e.g., time to attack, client update, lock requests, …), and send messages to others to exchange proposals.
  - Different processes may propose different values, but they can all accept any of the proposed values.
  - Only one of the proposed values will be “chosen” and eventually (once all failures are addressed) all of the nodes learn that *one chosen value*.
- Requirements:
  - **Consistency:** once a value is chosen, the chosen value of all working processes is the same.
  - **Validity:** the chosen value was proposed by one of the nodes.
  - T**ermination:** eventually they agree on a value (a.k.a., a value is “chosen”).

Consensus vs atomic commitment

- **Consensus** : participants need to agree on a value, but they are willing and capable to accept any value.
  - Ex: A group’s decision on where to meet (say, which specific room on campus of those that are of suitable size) can probably be cast as a consensus problem: most likely no one cares where they meet, but they all need to agree on the same value.

- Contrast with atomic commitment: participants need to agree on a value, but they have specific constraints on whether they can accept any particular value.
  - Ex: A group’s decision on when to meet is probably an atomic commitment problem, because each participant has his/her own calendar constraints.

Fischer-Lynch-Paterson Impossibility Result

- What FLP says: you can’t guarantee both safety and progress when there is even a single fault at an inopportune moment

- What FLP doesn’t say: in practice, how close can you get to the ideal (always safe and live)?

- Consensus protocols like Paxos get close in practice
  - The topic of this lecture

Paxos

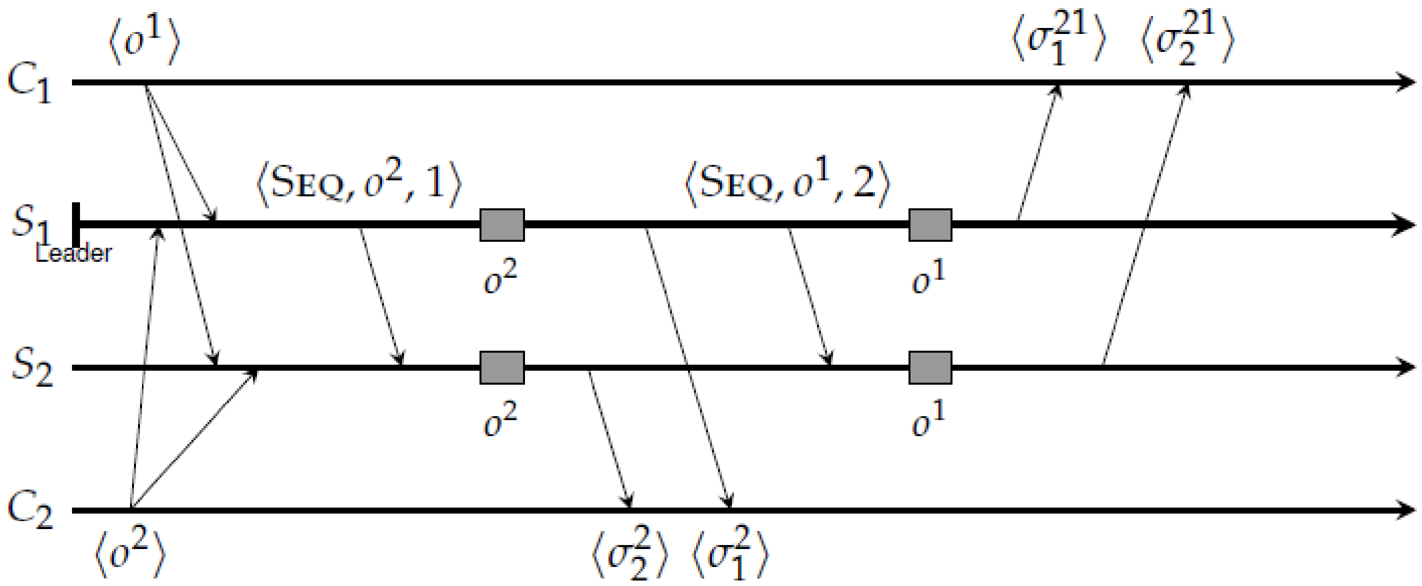
- The most popular fault-tolerant agreement protocol
  - Google Chubby (Paxos-based distributed lock service)
  - Google Spanner: geo-distributed transactional database
  - Yahoo Zookeeper (Paxos-based distributed lock service)
  - Open source: libpaxos (Paxos-based atomic broadcast)
- Paxos' properties: completely-safe and largely-live
- Safety
  - If agreement is reached, everyone agrees on the same value. The value agreed upon was proposed by some node
- Fault tolerance (i.e., as-good-as-it-gets liveness)
  - If less than half the nodes fail, the rest nodes reach agreement *eventually*
- No guaranteed termination (i.e., imperfect liveness)
  - Paxos may not always converge on a value, but only in very degenerate cases that are improbable in the real world

Let us build up Paxos from scratch to understand where many consensus algorithms actually come from.

Paxos: Starting point

- We assume a client-server configuration, with initially one primary server.
- To make the server more robust, we start with adding a backup server.
- To ensure that all commands are executed in the same order at both servers, the primary assigns unique sequence numbers/timestamps to all commands.
- In Paxos, the primary is called the ***leader***.

Two-server situation

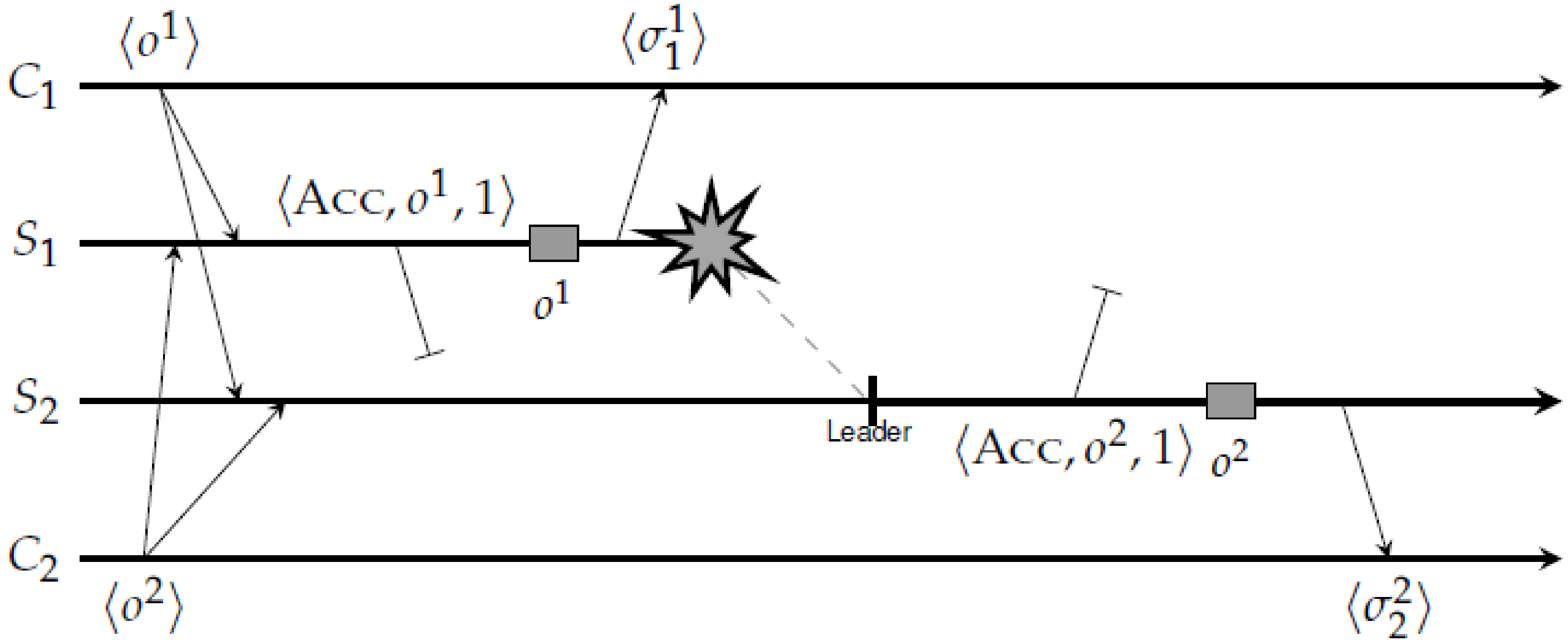


- S: servers, C: client, O^n: operations/commands
- σ_i^j: reponse from server i in state j expressed as sequence of ops carried out

Handling lost messages: Paxos terminology

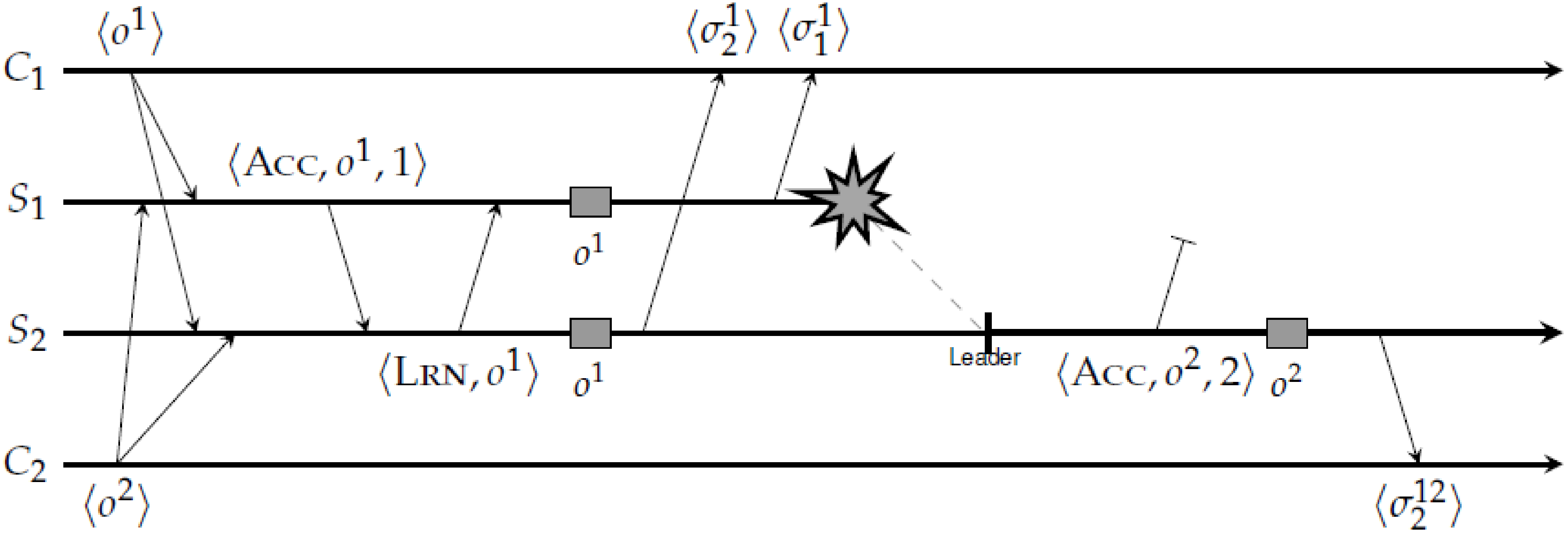
- Some Paxos terminology
  - The leader sends an accept message ACCEPT(*o*, *t*) to backups when assigning a timestamp *t* to command *o*.
  - A backup responds by sending a learn message: LEARN(*o*, *t*)

Two servers & one crash (1)



- Problem
  - Primary crashes after executing an operation, but the backup never received the accept message.

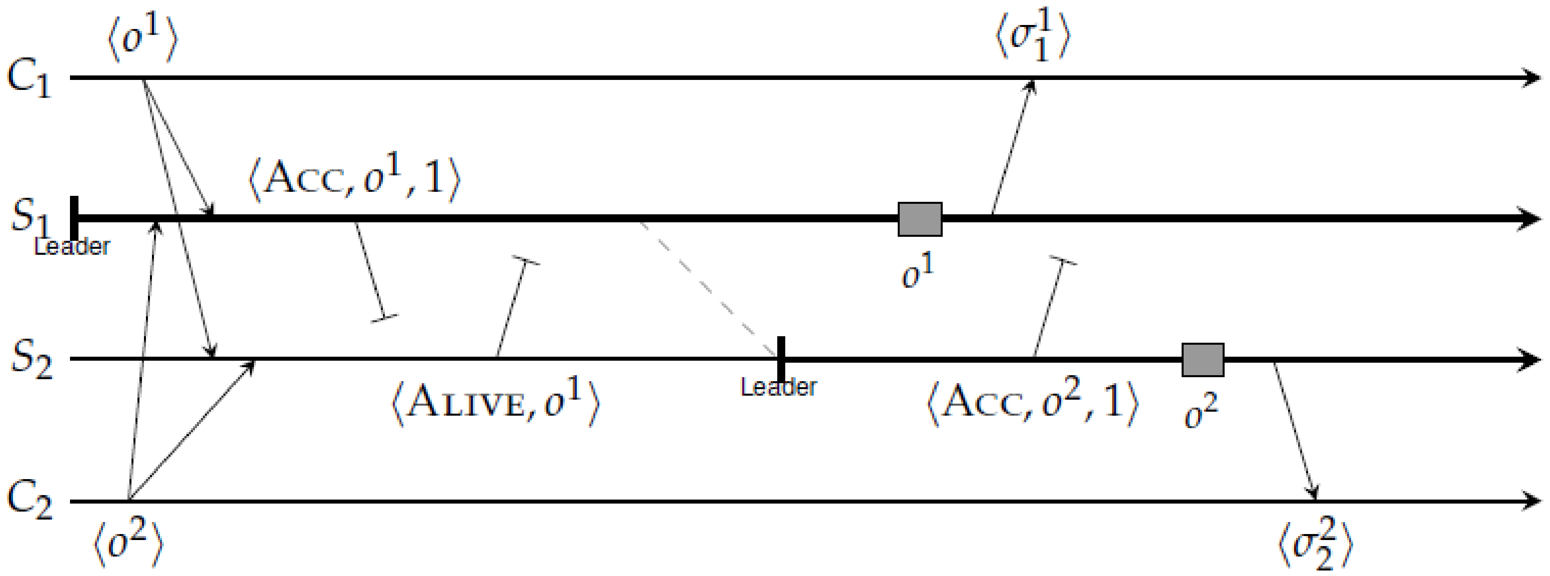
Two servers & one crash (2)



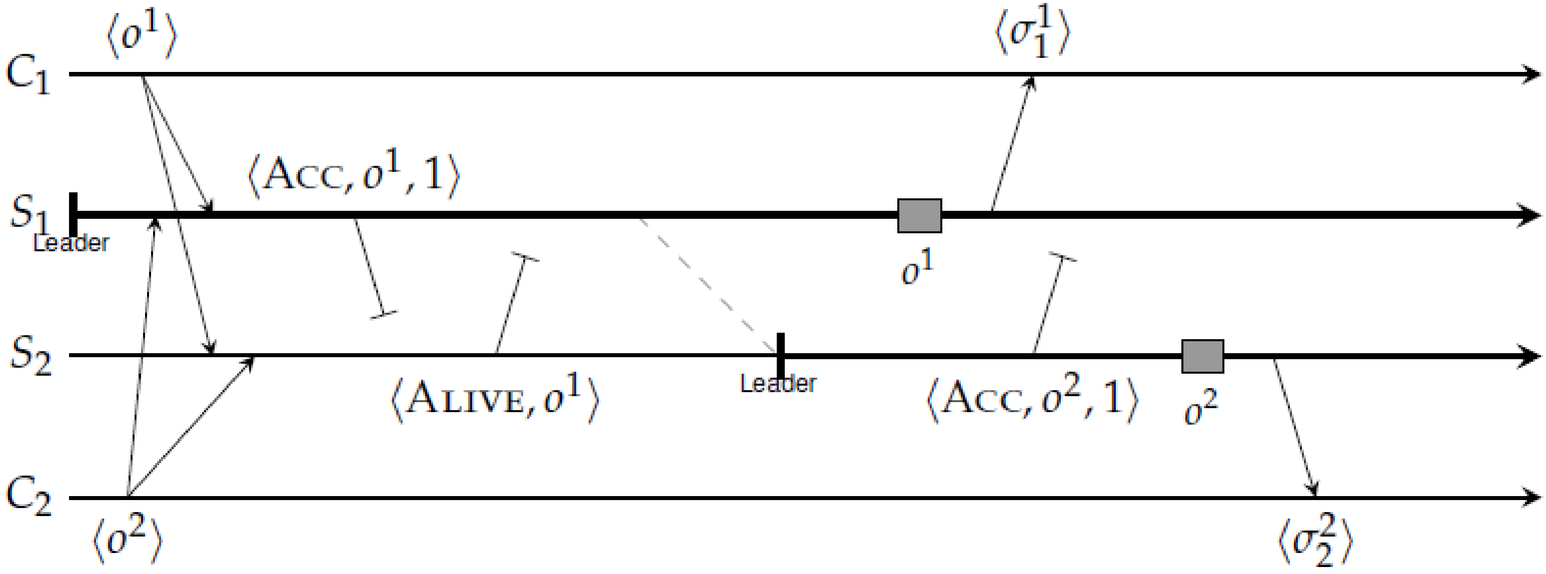
- Solution
  - **Never execute an operation before it is clear that is has been learned**
  - When the leader notices that operation *o* has not yet been learned, it retransmits ACCEPT(*o*, *t*) with the original timestamp.

Failure detection assumption

- Unrealistic assumption: Process can reliably detect crashes
- Only solution for failure detection in async. system: heartbeat
  - Each server sends out message "I'm alive"
  - Other servers set timeout on expected receipt of such messages
- But what happens if heartbeat is delayed?

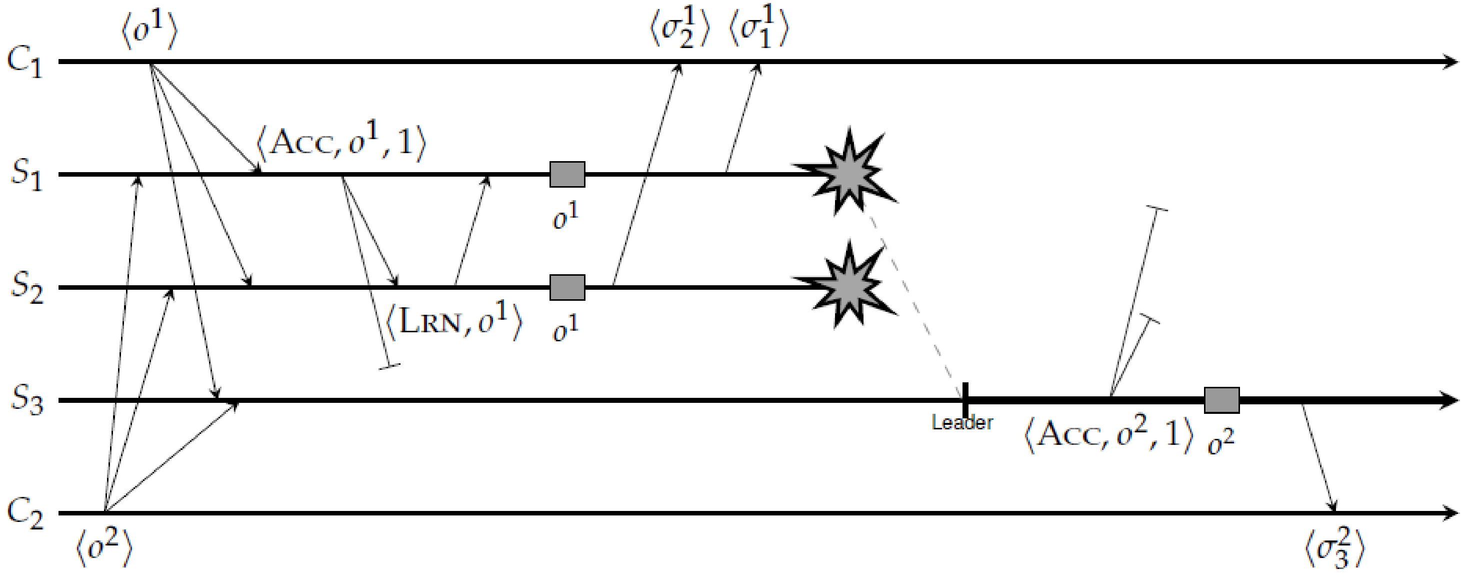


Required number of servers



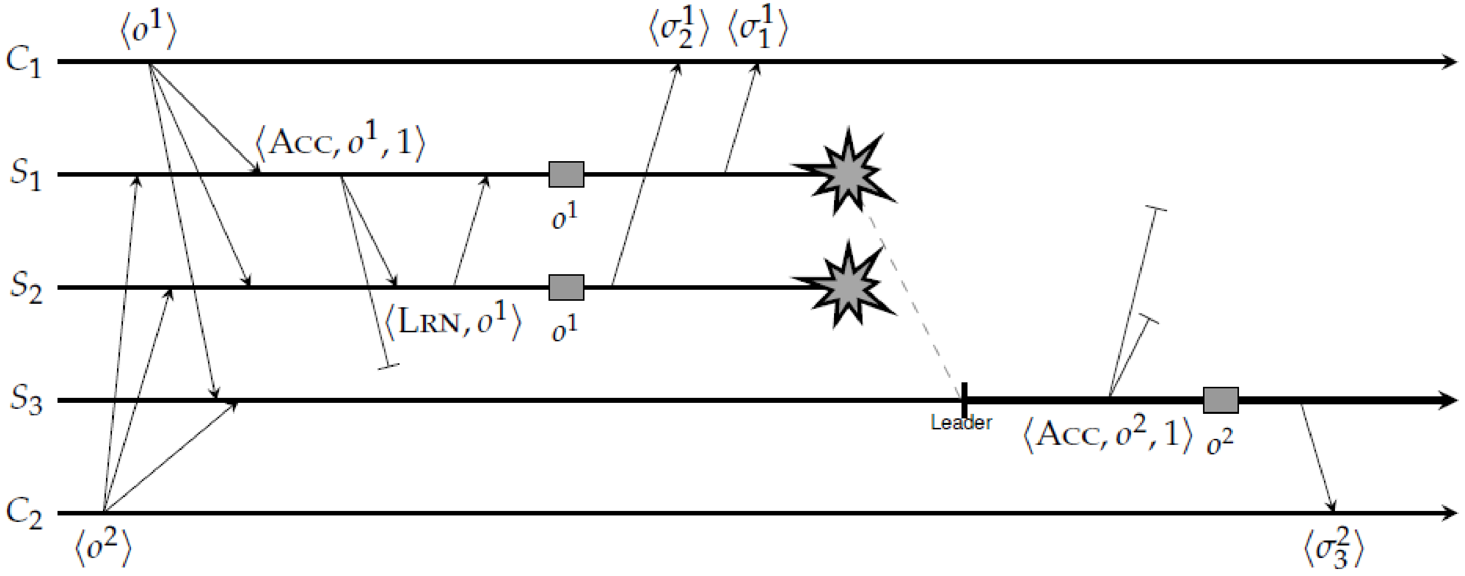
- Observation
  - **Paxos needs at least three servers**
  - **In Paxos with three servers, the leader cannot execute an operation o until it has received at least one (other) LEARN(o) message**

Required #servers in general case



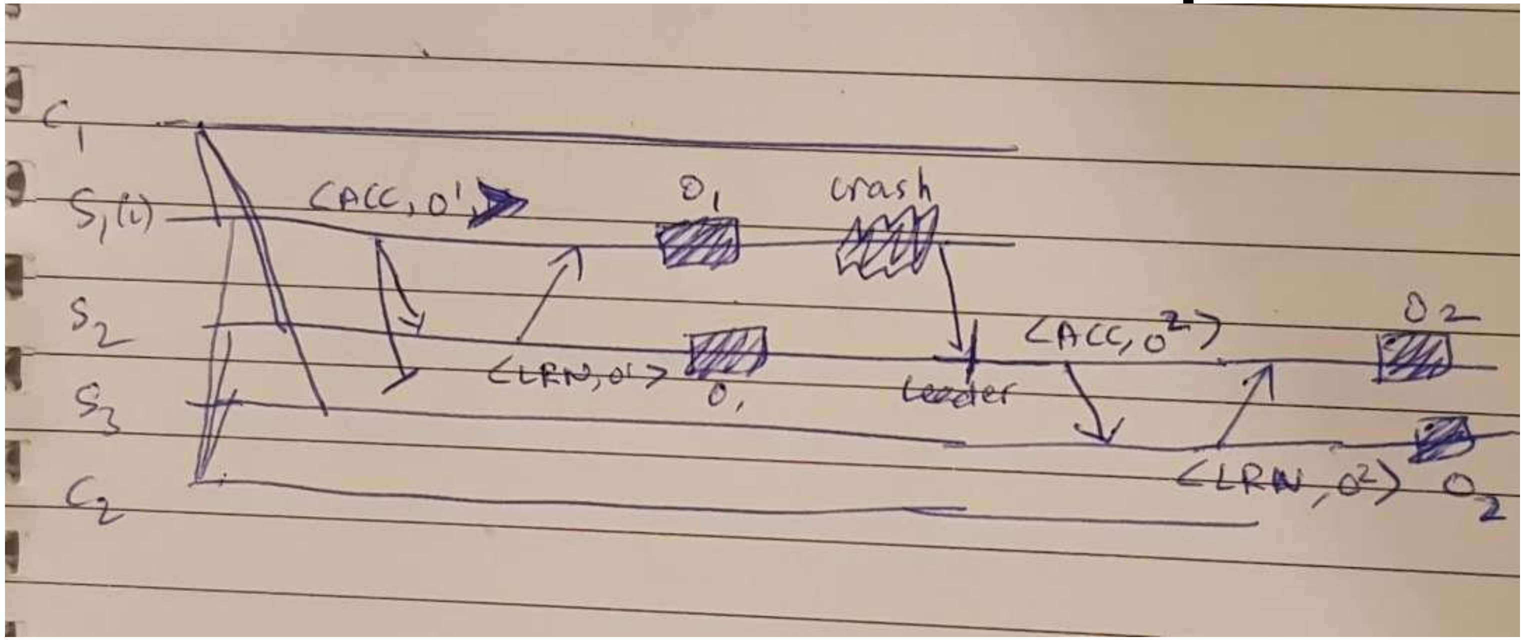
- Scenario
  - A crash of majority of servers can leave the remaining non-faulty ones inconsistent

Required #servers in general case



- Scenario
  - A crash of majority of servers can leave the remaining non-faulty ones inconsistent
- In general case to tolerate m failures, you need 2m + 1 servers
  - m+1 will give majority

Scenario: Need for timestamp



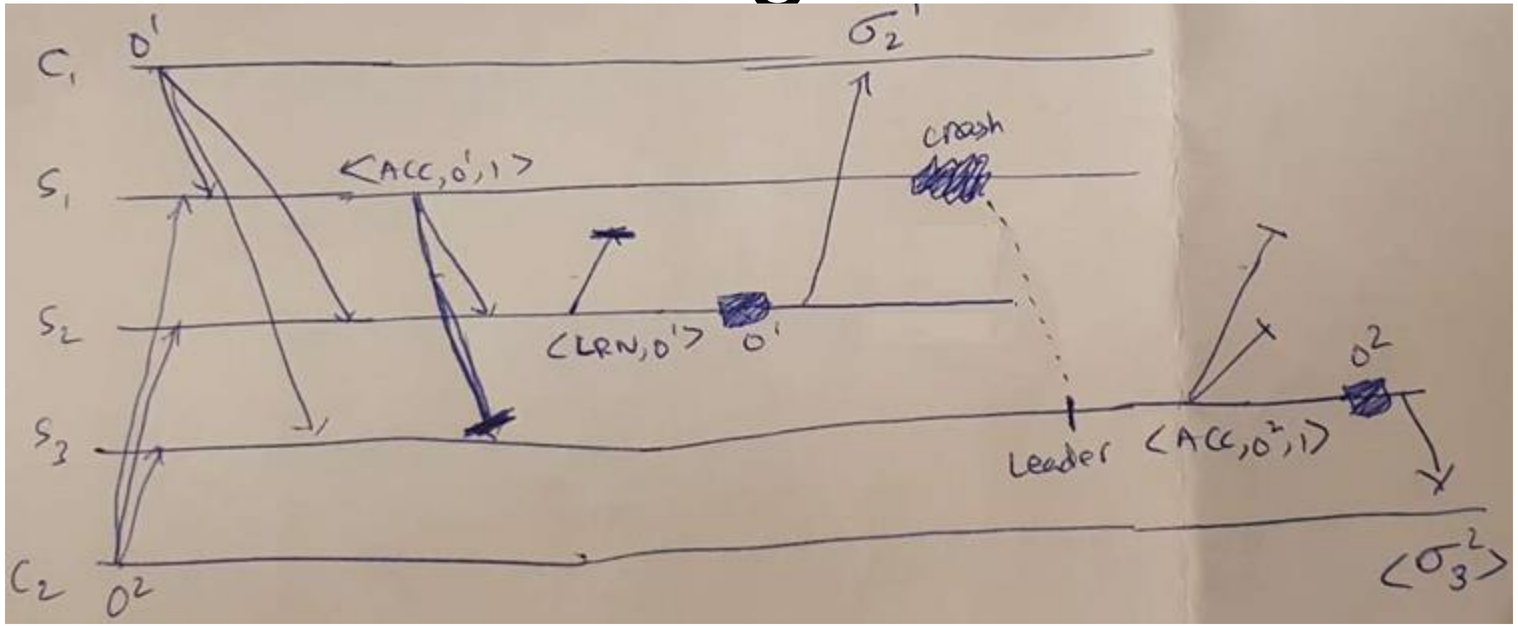
- S_3 is completely ignorant of any activity by S_1
  - Leader crashes after executing o^1
  - S_3 never even received ACCEPT(o^1)
  - S_2 received ACCEPT(o^1), detects crash, and becomes leader.
  - S_2 sends ACCEPT(o^2)
  - S_3 executes o^2 => Now, S_3 has done o^2 without doing o^1

Scenario: Leader crashes after executing o^1

- S_3 is completely ignorant of any activity by S_1
  - S_3 never even received ACCEPT(o^1).
  - S_2 received ACCEPT(o^1), detects crash, and becomes leader.
  - S_2 sends ACCEPT(o^2)
  - S_3 executes o^2 => Now, S_3 has done o^2 without doing o^1

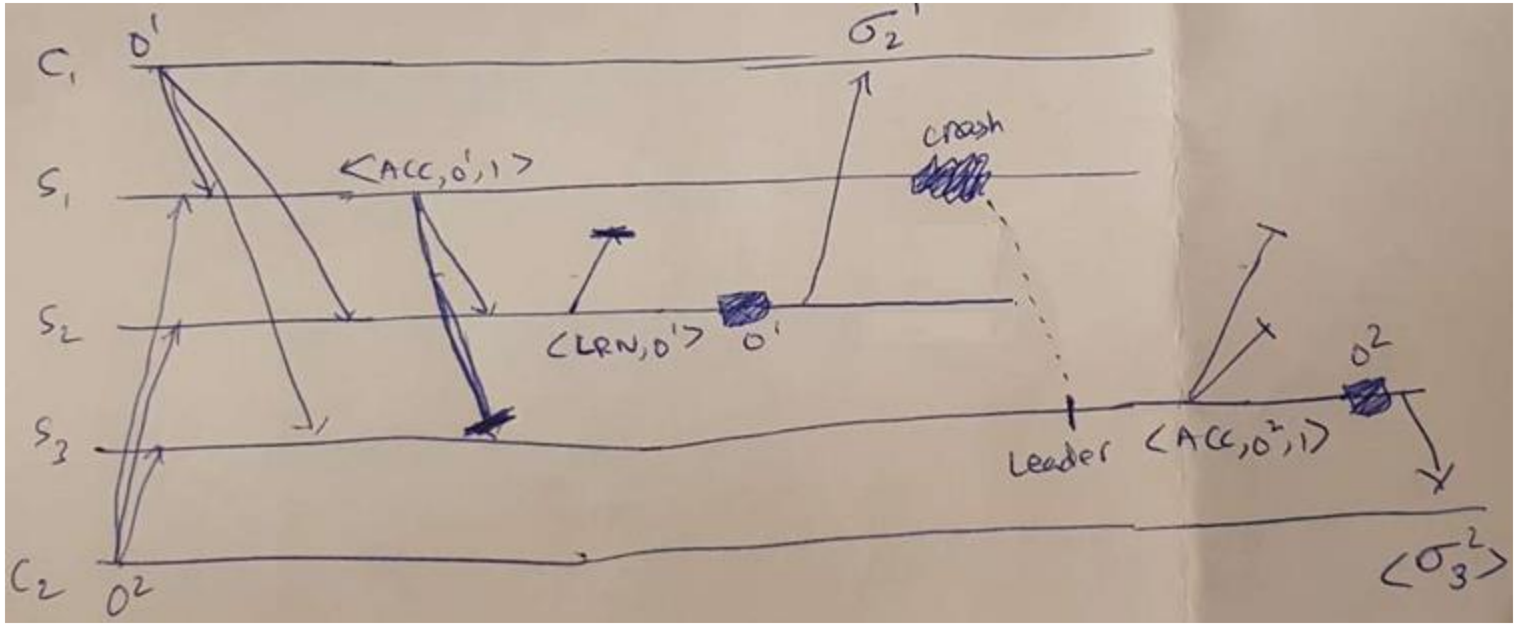
- Solution
  - Use timestamp to bring inconsistent server up to date
    - S_2 received ACCEPT(o^1,1), detects crash, and becomes leader.
    - S_2 sends ACCEPT(o^2 , 2)
      - Use a higher timestamp that what was seen before
    - S_3 sees unexpected timestamp and tells S_2 that it missed o^1.
    - S_2 retransmits ACCEPT(o^1, 1), allowing S_3 to catch up.

Need for broadcasting LEARN



- Scenario: What happens when LEARN(o^1) as sent by S_2 to S_1 is lost?
  - S_1 will not execute the operation but S_2 will

Need for broadcasting LEARN



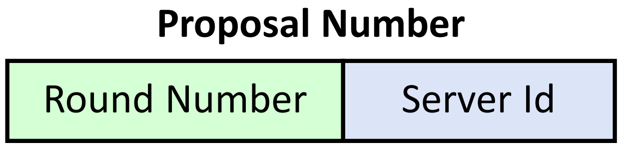
- Scenario: What happens when LEARN(o^1) as sent by S_2 to S_1 is lost?
- Solution: S_2 will also have to wait until it knows that S_3 has learned o^1.
- **Paxos fundamental rule: In Paxos, no server S (not just leader) can execute an operation o until it has received a LEARN(o) from a majority of non-faulty servers (2 in the 3 server case)**

Paxos: Detailed protocol

- Each node runs as a *proposer* and *acceptor*
- Proposer (leader) proposes a value and solicit acceptance from acceptors
- A value is *chosen* (a.k.a., consensus is reached) when a majority of acceptors have accepted it.
- A proposer announces a chosen value or tries again if it's failed to converge on a value.
- The protocol
  - guarantees *consistency* (all non-faulty nodes choose the same value)
  - guarantees *validity* (the chosen value was proposed by a proposer)
  - *ensures termination* (eventually, a value is chosen) with high probability but does not guarantee it.

Paxos Proposal Numbers

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



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

Paxos operation: node state

- Each node maintains:
  - my_n: my proposal # in the current Paxos
  - n_a: highest proposal # accepted
  - v_a: corresponding accepted value
  - n_h: highest proposal # seen

Paxos operation: 3P protocol

- Phase 1 (Propose)
  - A node decides to be leader (and propose)
  - Leader choose $my_n > n_h$
    - Could be done by simply by incrementing global counter and adding server id
  - Leader sends <prepare, my_n> to all nodes
  - Upon receiving <prepare, n> acceptor does the following

If $n < n_h$

reply <prepare-reject>

Else

$n_h = n$

reply <prepare-ok, n_a, v_a>

This node will not accept any proposal lower than n

Paxos operation: 3P protocol

- Phase 2 (Accept):
  - If leader gets prepare-ok from a majority

    V = non-empty value corresponding to the highest n_a received

    If V= null, then leader can pick any V

    Send <accept, my$_n$, V> to all nodes
  - If leader fails to get majority prepare-ok
    - Delay and restart Paxos
  - Upon receiving <accept, n, V> acceptor does following

    If $n < n_h$

    reply with <accept-reject>

    else

    $n_a = n$; $v_a = V$; $n_h = n$

    reply with <accept-ok>

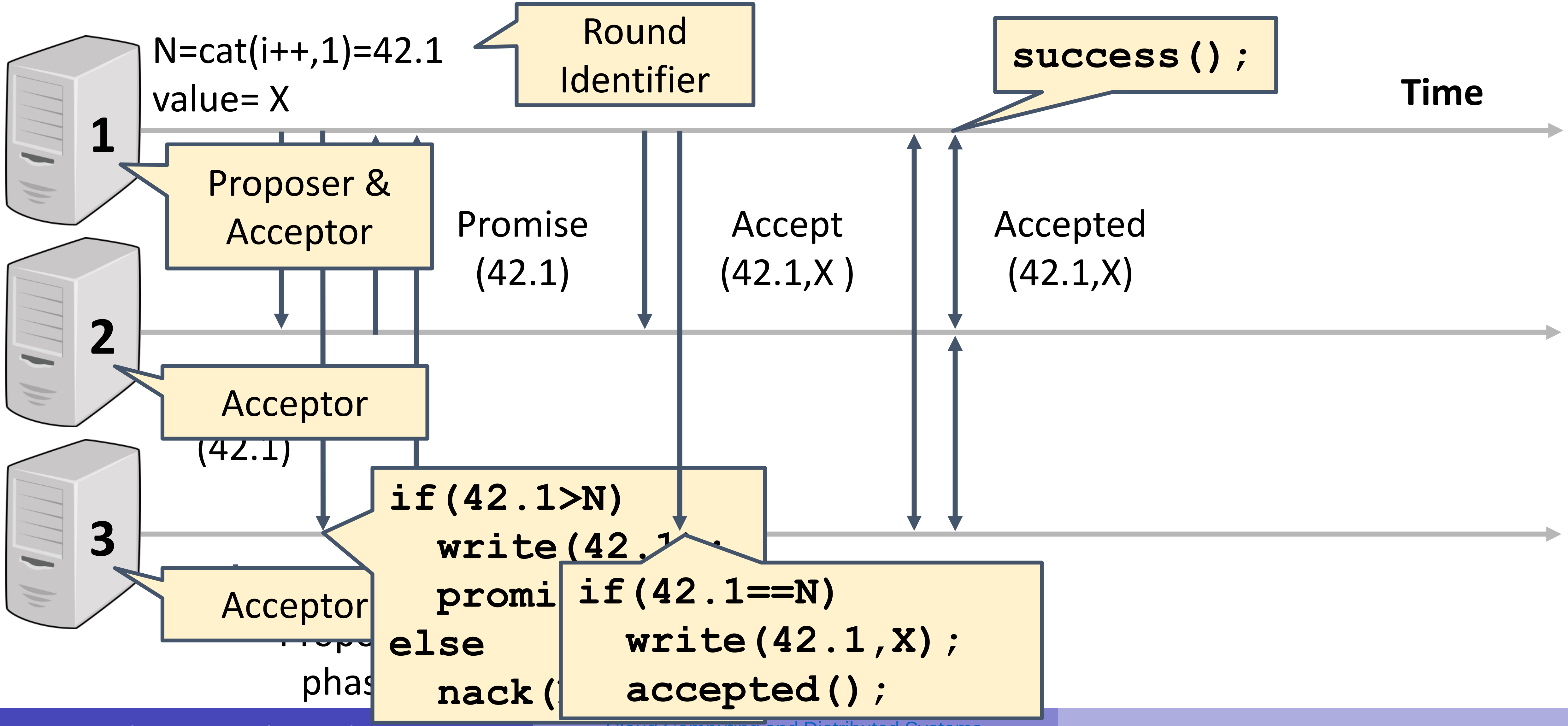
A point in the past, but its proposer didn't quite finish his job, then that value will remain in perpetuity

So: newer proposers win the rounds, but with old proposers' values!!!

Paxos operation: 3P protocol

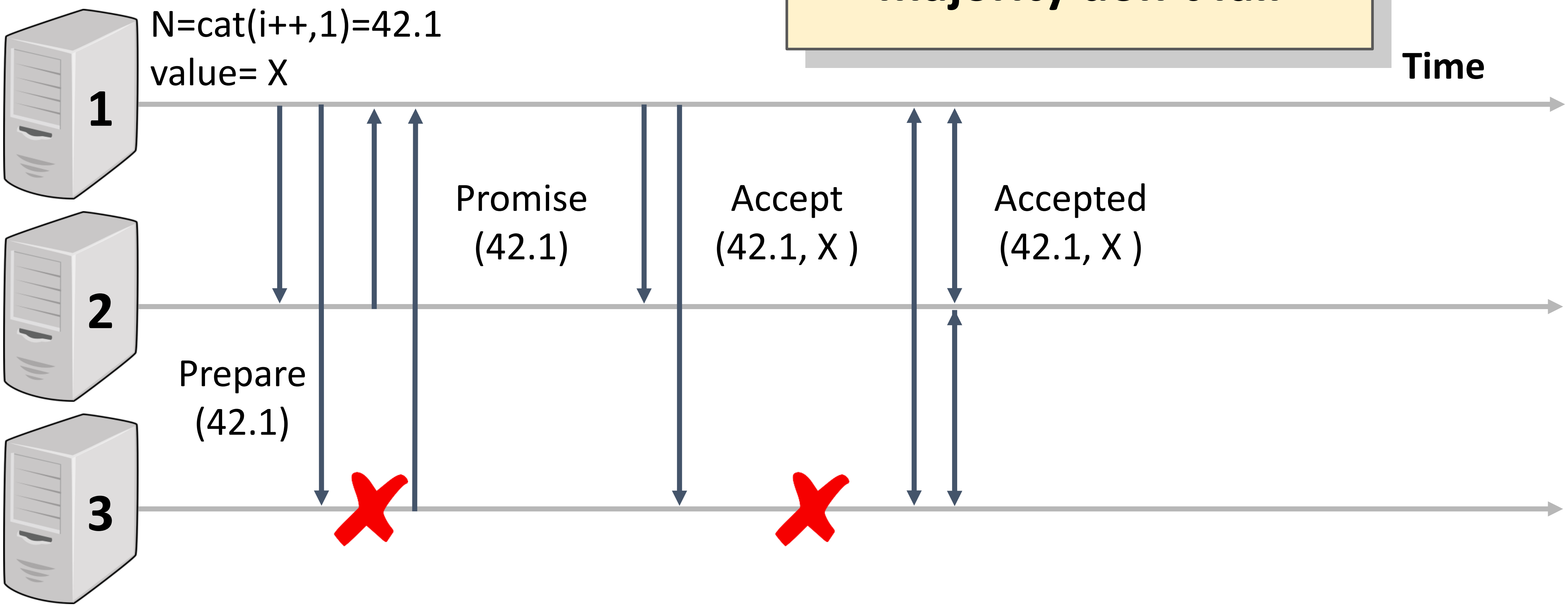
- Phase 3 (Decide)
  - If leader gets accept-ok from a majority
    - Return Done to client
    - Send <DECIDE, Va> to all nodes until you get <DECIDE, OK> back
      - This phase is so that nodes close the protocol, and so that nodes that might not have heard previous ACCEPT messages learn the chosen value.
  - If leader fails to get accept-ok from a majority
    - Delay and restart Paxos

Basic Paxos (Example without phase 3)

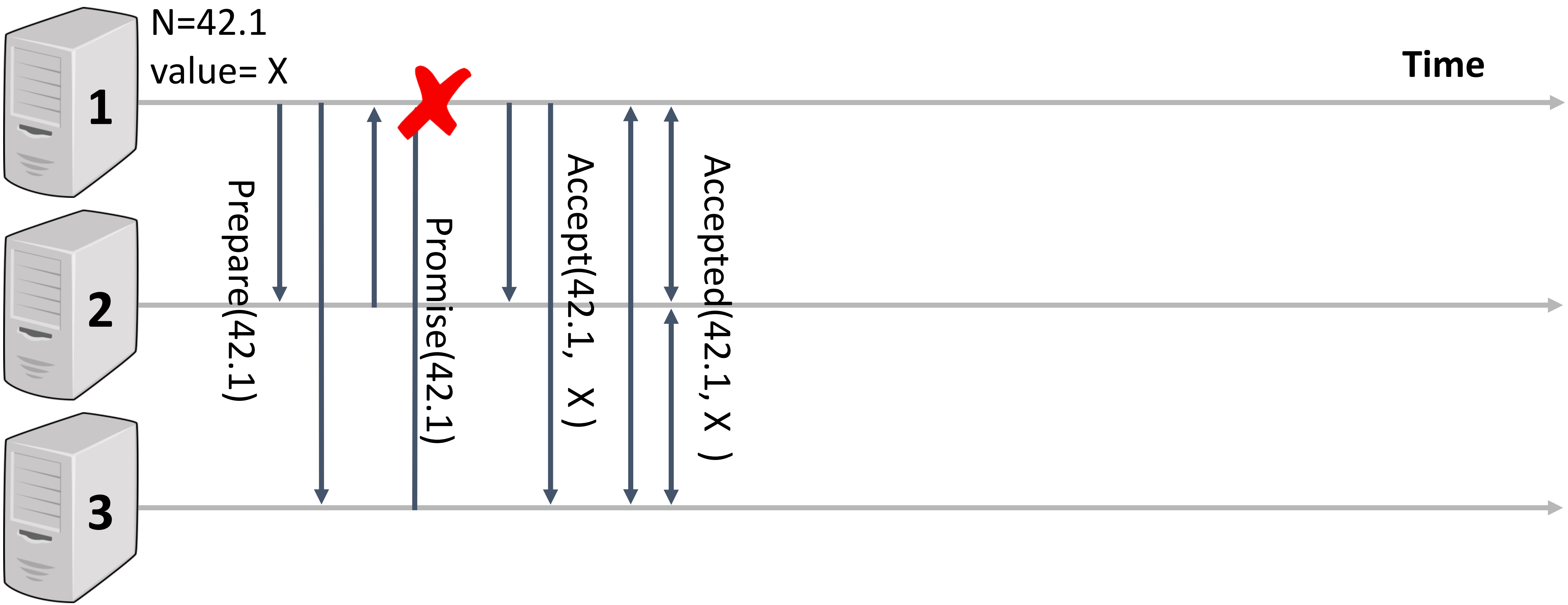


Failures: Acceptor

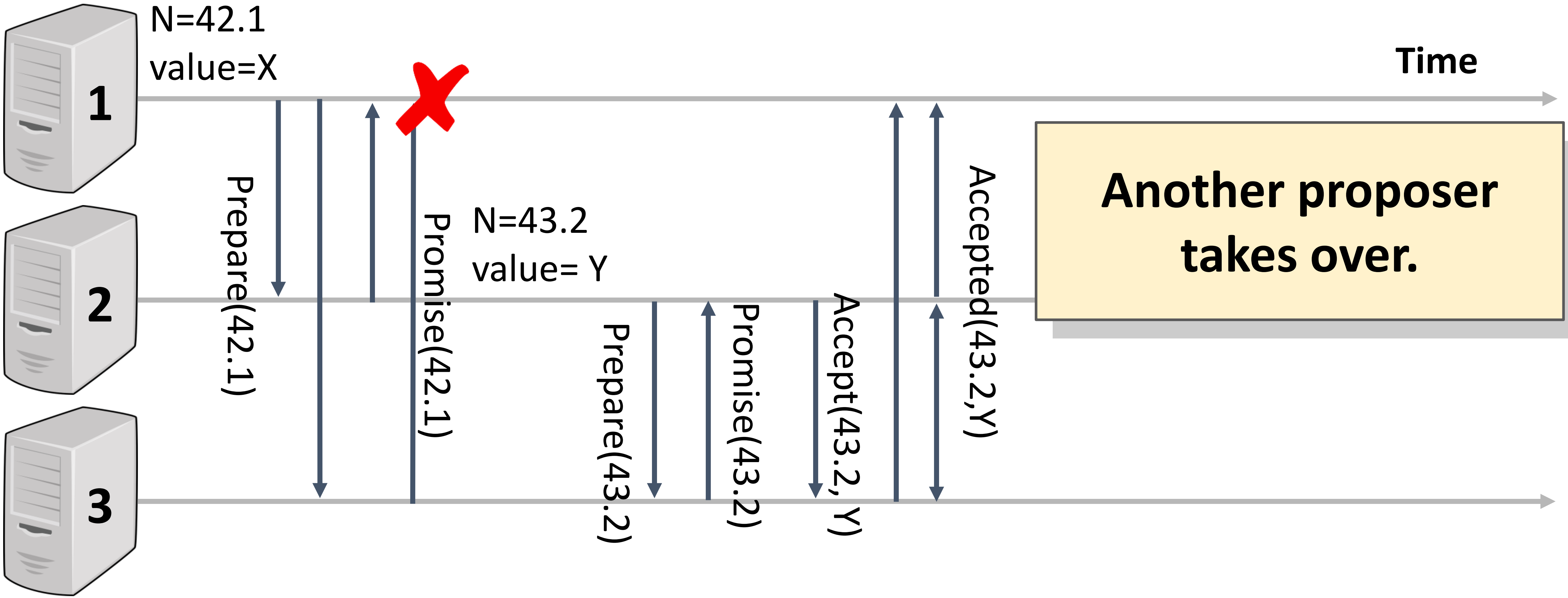
No problem as long as majority don't fail



Failures: Proposer in Prepare Phase

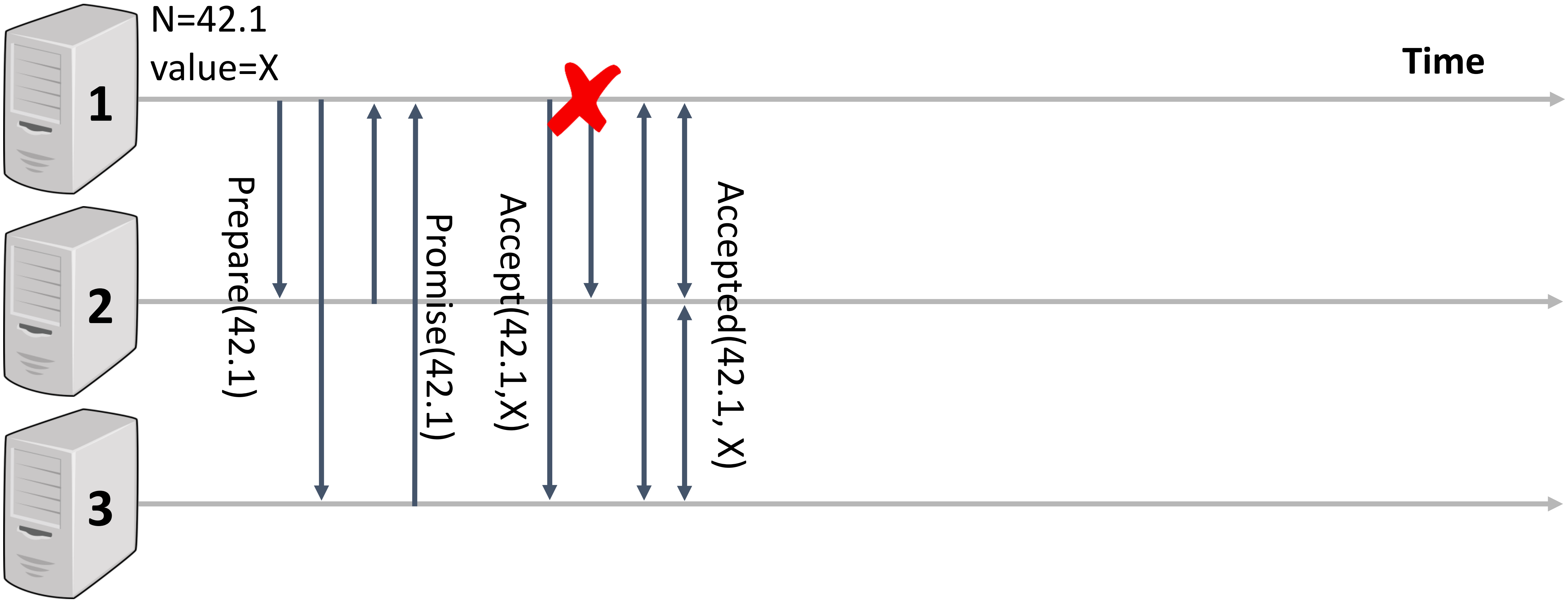


Failures: Proposer in Prepare Phase

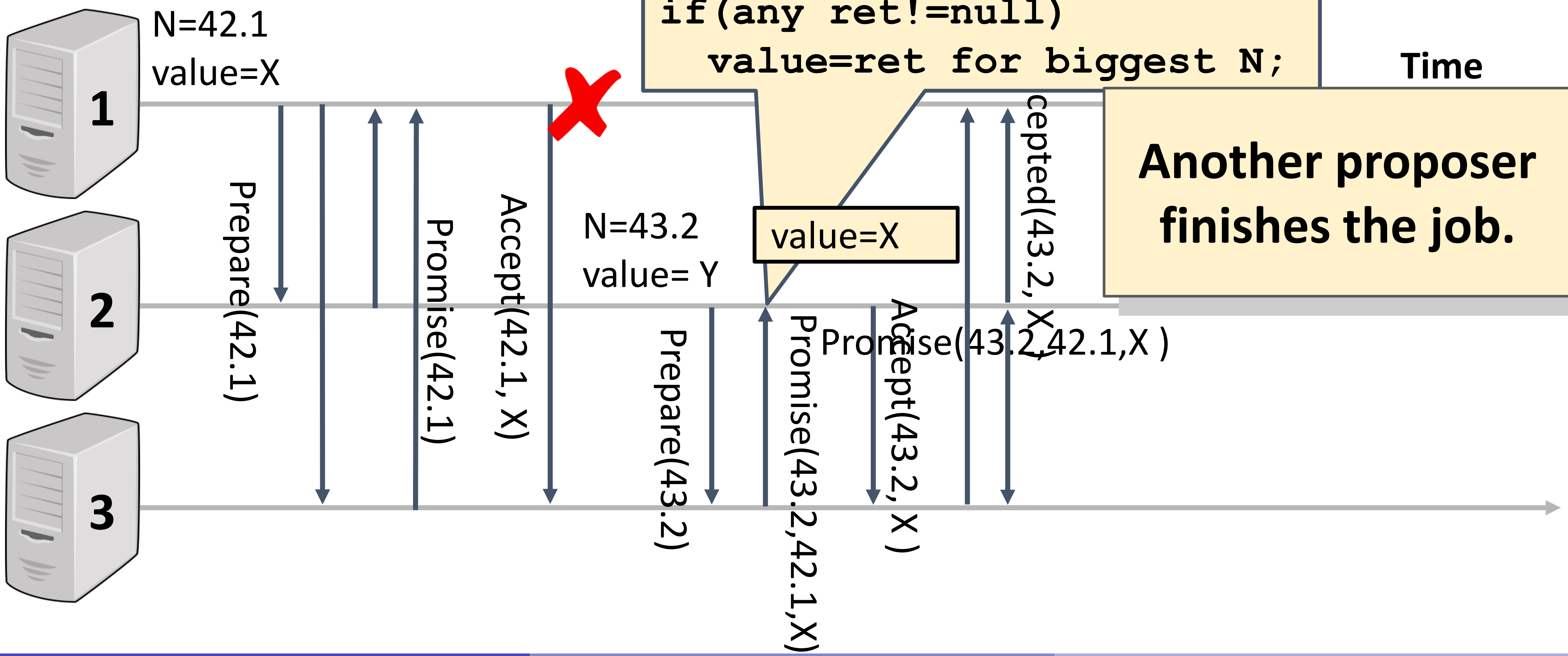


Another proposer takes over.

Failures: Proposer in Accept Phase



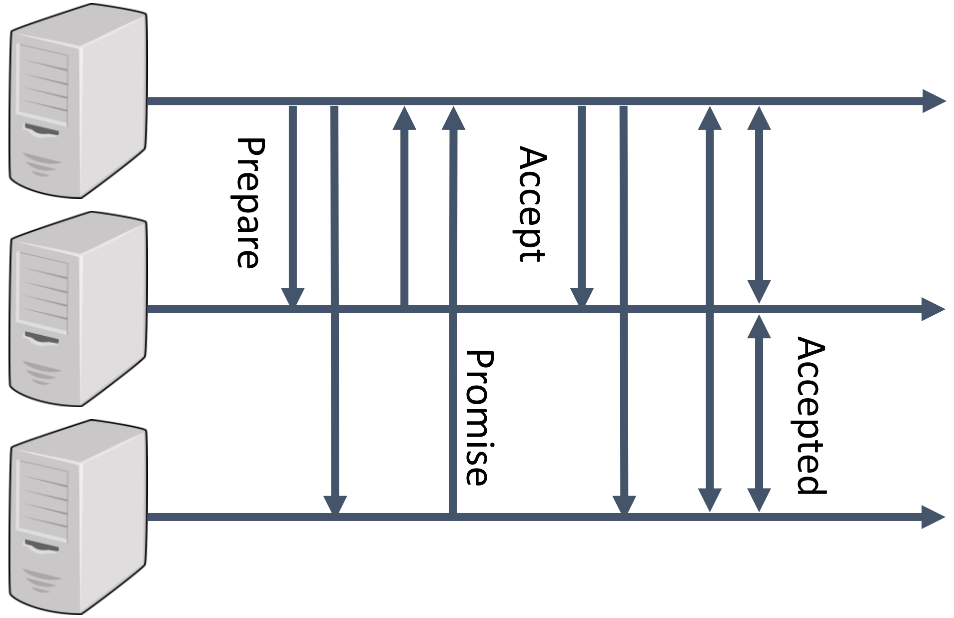
Failures: Proposer in Accept Phase




```
if(any ret!=null)
  value=ret for biggest N;
```

Another proposer finishes the job.

What could go wrong?



- One proposer
  - One or more acceptors fails
    - Still works as long as majority are up
  - Proposer fails in prepare phase
    - No-op; another proposer can make progress
  - Proposer fails in accept phase
    - Another proposer overwrites
    - Another proposer finishes the job

- Two or more "simultaneous" proposers
  - A bit more complex and Paxos can come to a grinding halt due to livelock between proposers (no proposer is able to get majority)!

Paxos in the real world

- Solving livelock with explicit leader election (ex: Bully algorithm)
- Developer: Creating a log of agreements (make more than one decision)
  - Multi-Paxos
- Sysadmin work: Adding and removing nodes from Paxos
  - Naming and cluster membership
- Software teser: Testing and debugging is hard
- Researcher: Do you want to handle byzantine failures?
  - What happens if **N=∞** due to disk corruption?

Need to implement? Don't! Just use a library!
Or use RAFT

Performance

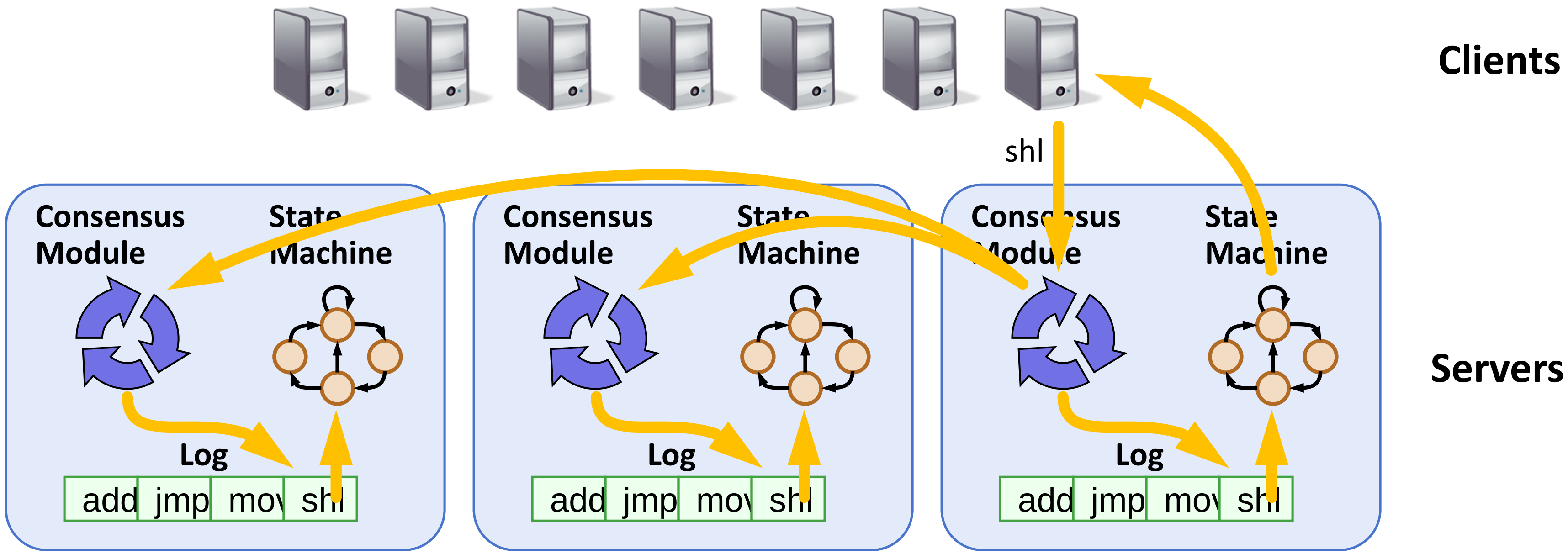
- Basic Paxos latency:
  - 2 network round trips +
  - 2 disk writes
- Multi-Paxos latency:
  - Converges on 1 network round trip + 1 disk write
    - Assuming failures are rare
- Improving reliability: want multiple sites!
  - But that implies cross-rack/datacenter/zone/continent messaging
- Bandwidth can be increased through:
  - Batching operations into one Paxos instance
    - Each paxos round completes multiple transactions
  - Parallel Paxos instances

Trade-off: more latency for more reliability.

Putting together Paxos + 2PC: Sharding & Replication

- We talked about single node database
  - ACID properties of transactions, Isolation with 2PL
- In practice, databases are distributed
  - Data sharded/partitioned for *scalability*
  - We talked about 2PC for atomicity across shards
- In practice, shards are replicated for fault tolerance
  - Each database holding replica typically uses a write-ahead log to guarantee durability (D of ACID)
  - How do you make sure all replicas update the log in the same order?

Practical Paxos Use Case: Replicated Log



- All servers execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up

Example: Google Spanner

- A database system with millions of nodes, petabytes of data,distributed across datacenters worldwide
- Consistency properties:
  - Serializable transaction isolation
  - Many shards, each holding a subset of the data; atomic commit of transactions across shards

- Many standard techniques:
  - Paxos replication within a shard
  - Two-phase locking for serializability
  - Two-phase commit for cross-shard atomicity

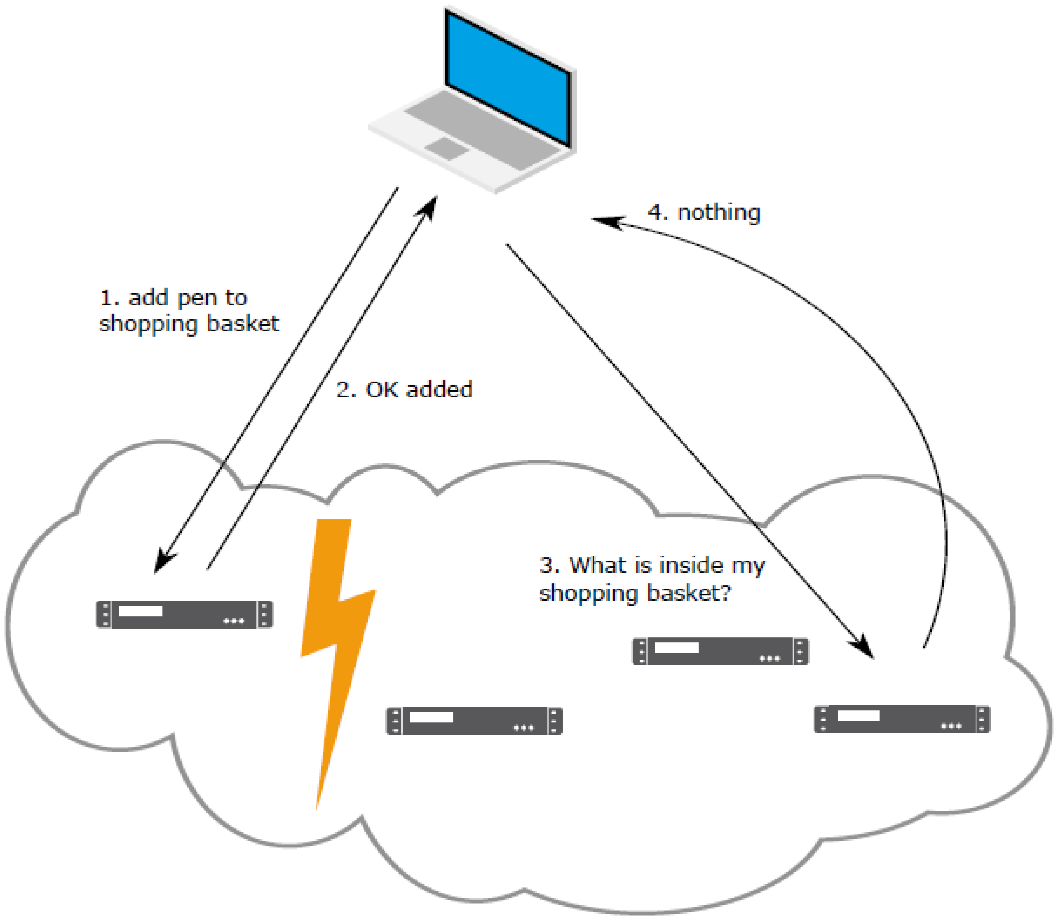
Paxos vs. 2/3PC

- Remember:
  - 2PC was vulnerable to 1-node failures, especially coordinator failures
  - 3PC was vulnerable to network partitions

- Paxos deals with these issues using two mechanisms:
  - Egalitarian consensus: no node is special, anyone can take over as coordinator at any time
    - Hence, if one coordinator fails, another one will time out and take over
    - But that requires special ordering and acceptance protocols for proposals
  - Safe majorities: instead of requiring all participants to answer Yes, Paxos requires only half + 1 of the nodes
    - Because you cannot have two simultaneous majorities, which avoids partitions

- But, If you don't have a majority of non-faulty nodes, Paxos will prioritize consistency over availability
  - Writes will not succeed.

CAP: Why not all three?



If there is a network partition either C or A will break

CAP Theorem

- First stated by Eric Brewer (Berkeley) at the PODC 2000 keynote
  - Formally proved by Gilbert and Lynch in 2002
- Consistency (specifically Linearizability)
  - Linearizability = sequential consistency + real-time constraint
- Availability
  - a system is available if every request to a non-failing node always receives a response, eventually
- Partition tolerance
  - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes
- The theorem says: between Consistency, Availability, Partition tolerance, you can choose only two

Design Tradeoff

- Network partitions occur outside anyone's control in real life
  - Cannot sacrifice the Partition-Tolerance property
- In the event of a network partition either A or C is maintained: it is the choice of the designer
- Practical distributed systems are CP or AP
  - CP oriented: BigTable, Hbase, MongoDB, Redis, MemCached, …
  - AP oriented: Amazon Dynamo, CouchDB, Cassandra, …
- Thus The CAP theorem formally states the trade-offs among different distributed systems properties

Closing

- We have covered a lot of ground
  - Cloud computing & service models
  - Virtualization, Docker: Infrastructure fundamentals
  - MapReduce, Spark: Programming fundamentals
  - RDMBS, NFS, AFS, GFS: Data management fundamentals
  - Consistency & fault tolerance: Distributed systems fundamentals
- You have all the tools to build a cloud application
- Cloud jobs are in demand. Go put your theoretical knowledge into practice with AWS, Azure, or GCP

The top 10 most in-demand hard skills globally

1. Blockchain
2. Cloud computing
3. Analytical reasoning
4. Artificial intelligence

Acknowlegements

- This course was built with material from
  - CMU cloud computing, distributed systems courses by Prof. Majd Sakr, Prof. Dave Anderson, Prof. Satya Mahadev
  - Columbia distributed systems course by Prof. Roxana Geambasu
  - NYU distributed systems by Prof. Jinyang Li
  - MIT distributed systems course by Prof. Robert Morris
  - Distributed systems book by Andy Tanenbaum & Maarteen van Steen
- The final project were made possible by Eugenio Marinelli
- Finally, this could would not have been possible without you guys! Thank you!

Good luck!
Maybe your future be cloudy with lots of (vegan) meatballs!