Agreement in Distributed Systems: Consensus

Lecture 9

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

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

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”).

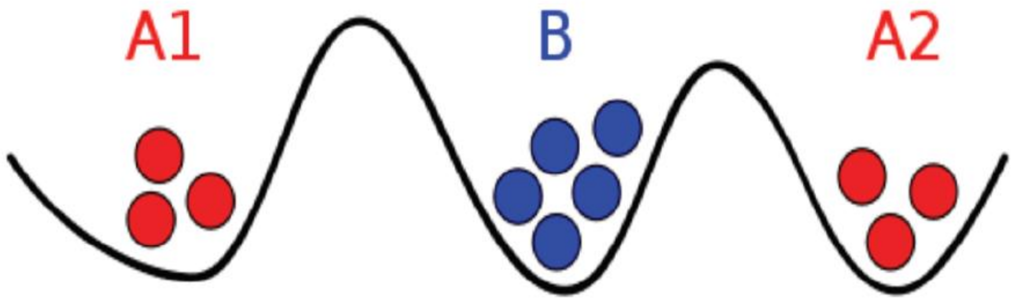
Consensus vs atomic commitment

- **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.

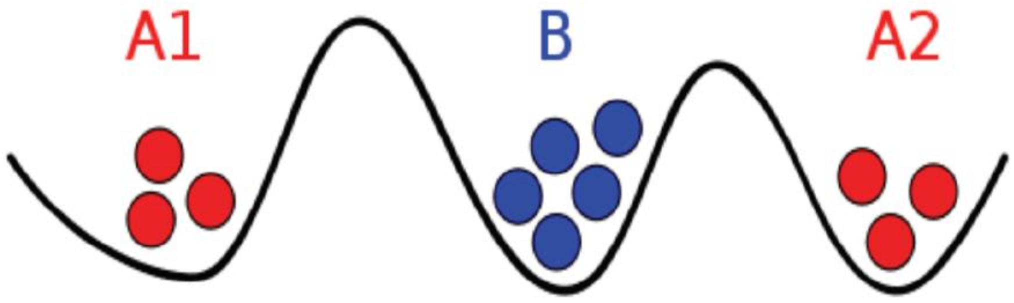
Two Generals Problem

- Two armies want to attack a fortified city.
- Both armies need to attack at the same time in order to succeed.
- The armies can only communicate through messengers
- Messengers can be captured, so message delivery is not reliable.



Solution properties

- **Consistency**
  - both armies decide to attack at the same time
- **Validity**
  - the time to attack was proposed by one of the armies
- **Termination**
  - each army decides to attack after a finite number of messages



When is consensus possible?

- Synchronous systems: known delays, reliably delivery of messages
  - Pre-agree on either A1 or A2 generals proposing the time to attack. Say A1 is the one to propose. A2 will be the one to accept.
  - A1 sets the time of attack to communication delay + some extra time to account for A2's preparation for response.
  - Problem solved

- Asynchronous systems: Unknown delays or unreliable delivery
  - Need acks, but acks could be arbitrarily delayed/lost, too. Therefore I need more acks at every step. Therefore, one general can never be sure that the other will attack. So they can't be guaranteed to reach agreement

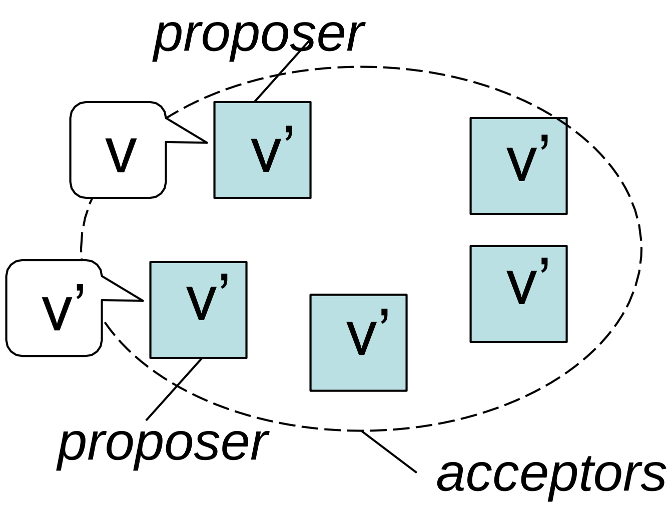
- FLP: Achieving consistency, validity, and termination is provably impossible
  - FLP doesn't mean consensus won't be achieved, just that it can't be guaranteed

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

Paxos: The Basic Idea

- Paxos is similar to 2PC, but with some twists
- One (or more) node decides to be coordinator (*proposer*)
- Proposer proposes a value and solicits acceptance from others (*acceptors*)
- Proposer announces the chosen value or tries again if it's failed to converge on a value



- Hence, Paxos is egalitarian: any node can propose/accept, no one has special powers
- Basic idea is natural in retrospect, but why it works in any detail is incredibly complex!

Challenges Addressed in Paxos

- What if multiple nodes become proposers simultaneously?
- What if the new proposer proposes different values than an already decided value?
- What if there is a network partition?
- What if a proposer crashes in the middle of solicitation?
- What if a proposer crashes after deciding but before announcing results?
- …

Building up to Paxos with Process Resilience

- Basic idea
  - Protect against malfunctioning processes through process replication, organizing multiple processes into process group.

- *k* -fault tolerant group
  - When a group can mask any *k* concurrent member failures (*k* is called degree of fault tolerance).

- Benefit
  - Allow a process to deal with group of processes as a single abstraction

Consensus in Process Resilience

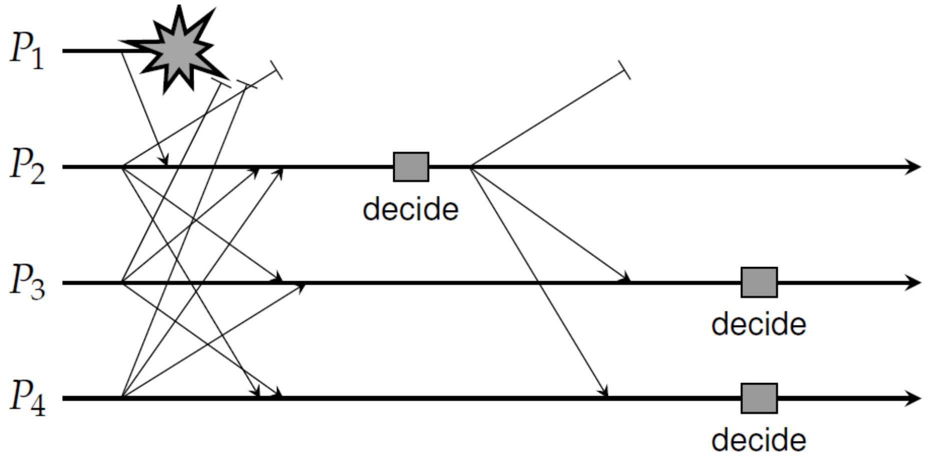
- Assumptions & prerequisites
  - All members are identical
  - All members process commands in the same order

- Reformulation
  - Non faulty group members need to reach consensus on which command to execute next.

Flooding-based consensus

- System model
  - A process group **P** = $\{P_1, \ldots, P_n\}$
  - Fail-stop failure semantics, i.e., with reliable failure detection
  - A client contacts a P_i requesting it to execute a command
  - Every P_i maintains a list of proposed commands
- Basic algorithm (based on rounds)
  - In round r, P_i multicasts its known set of commands $\mathbf{C}^r$ to all others
  - At the end of r, each P merges all received commands into a new **C** .
  - Next command cmd_i selected through a globally shared, deterministic function: $cmd \leftarrow select(\mathbf{C}_j^{r+1})$.

Flooding-based consensus: Example



- Observations
  - P2 received all proposed commands from all other processes and makes decision.
  - P3 may have detected that P1 crashed, but does not know if P2 received anything, i.e., P3 cannot know if it has the same information as P2) cannot make decision (same for P4).
- Action
  - P3, P4 wait until next round without executing command. P2 can broadcast its decision in next round allow P3, P4 to continue.

Flooding-based consensus analysis

- Why does it work?
  - A process moves to next round without executing command only if it detects failure
  - Worst-case, only one non-faulty process remains that can move forward
- But not realistic as it assumes fail-stop system
  - Need to reliable be able to detect failures within a specific time interval

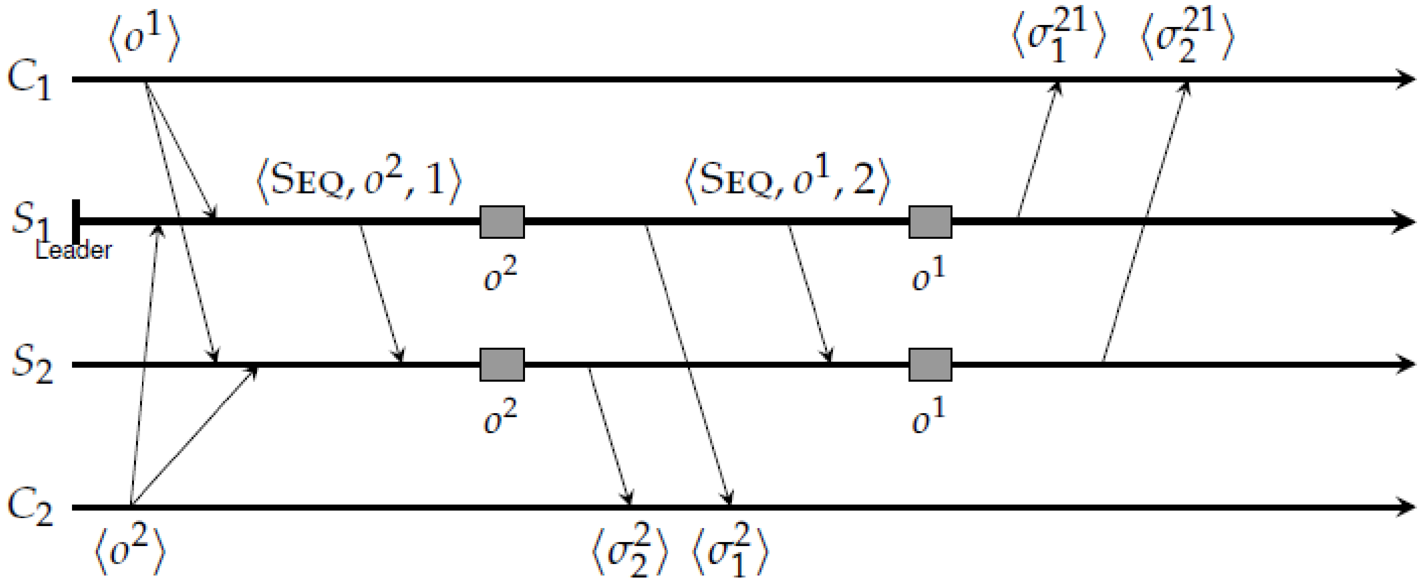
Realistic Consensus: Paxos

- Assumptions (rather weak ones, and realistic)
  - A partially synchronous system (in fact, it may even be asynchronous).
  - Communication between processes may be unreliable: messages may be lost, duplicated, or reordered.
  - Corrupted message can be detected (and thus subsequently ignored).
  - All operations are deterministic: once an execution is started, it is known exactly what it will do.
  - Processes may exhibit crash failures, but not arbitrary failures.
  - Processes do not collude.
- 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 to all commands.
- In Paxos, the primary is called the ***leader***.

Two-server situation

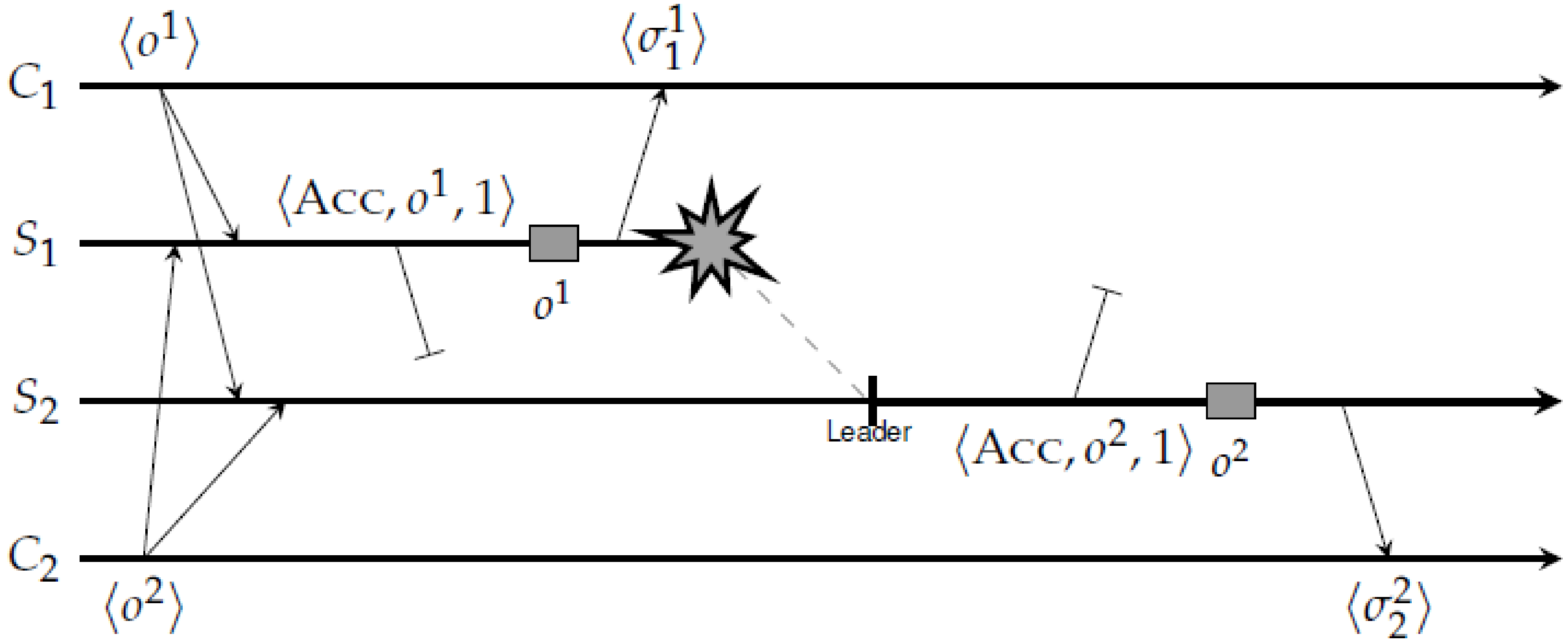


- S: servers, C: client, O^n: operations/commands
- σ_i^j: reponse from server i in state j expressed as sequence of ops carried out

Handling lost messages: Paxos terminology

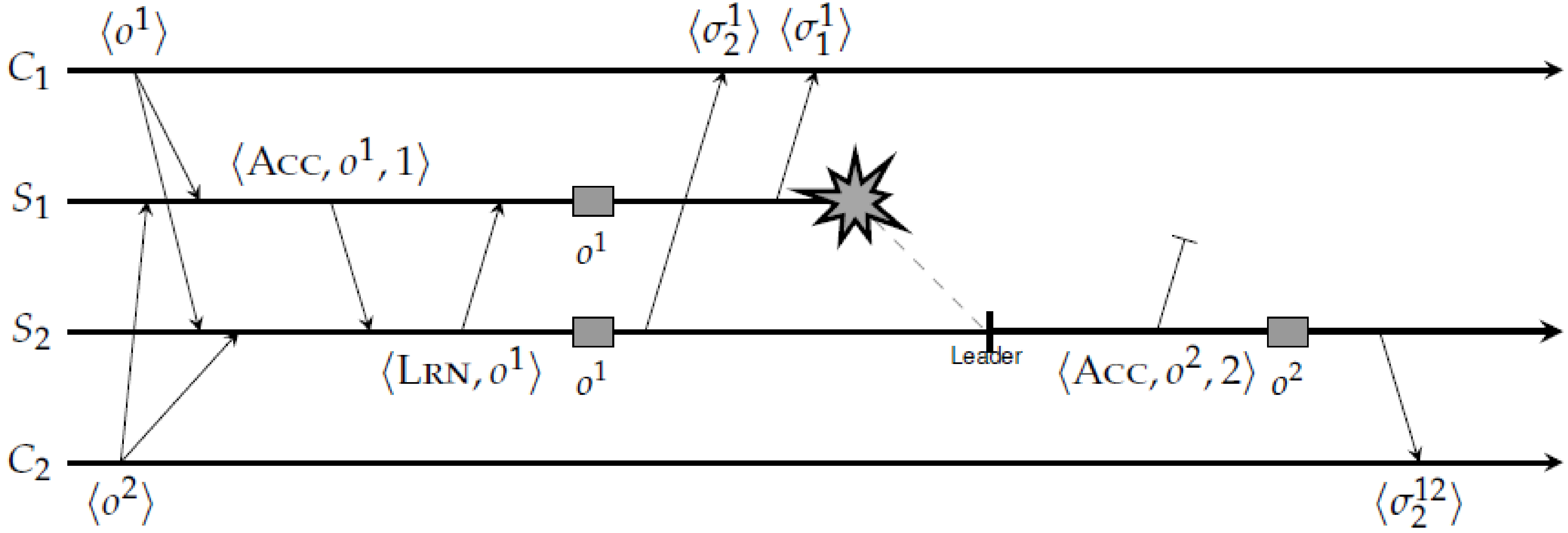
- Some Paxos terminology
  - The leader sends an accept message ACCEPT(o, t) to backups when assigning a timestamp t to command o.
  - A backup responds by sending a learn message: LEARN(o, t)
  - When the leader notices that operation o has not yet been learned, it retransmits ACCEPT(o, t) with the original timestamp.

Two servers & one crash (1)



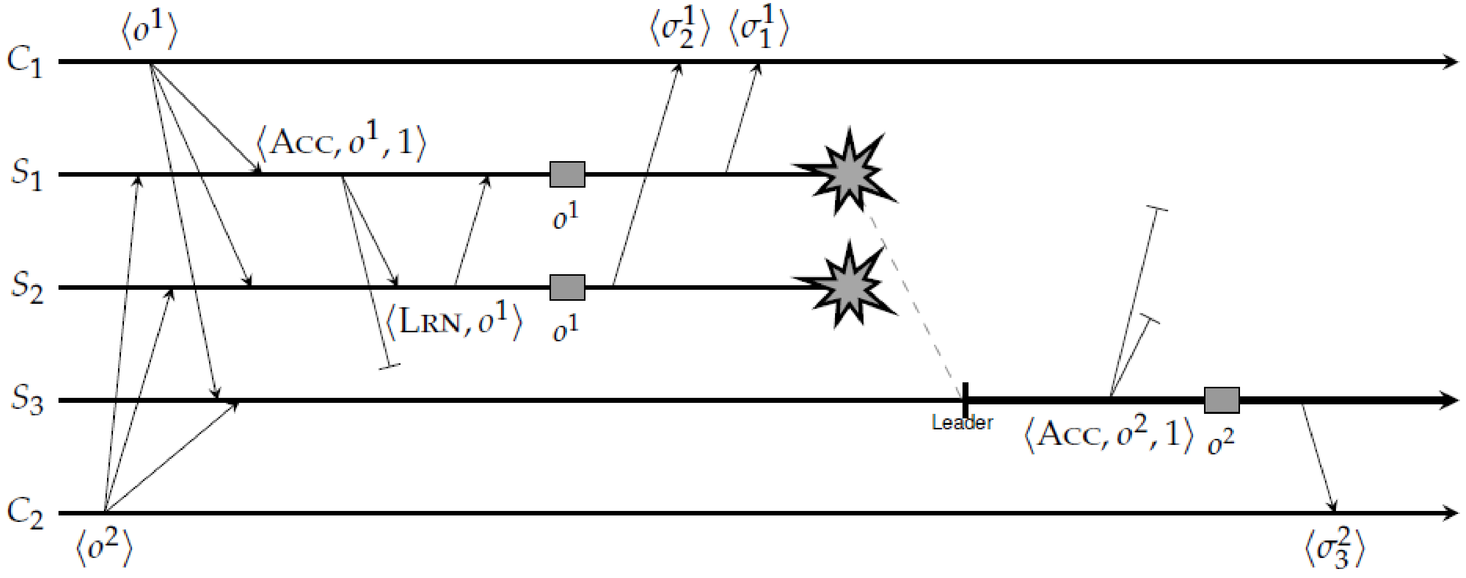
- Problem
  - Primary crashes after executing an operation, but the backup never received the accept message.

Two servers & one crash (2)



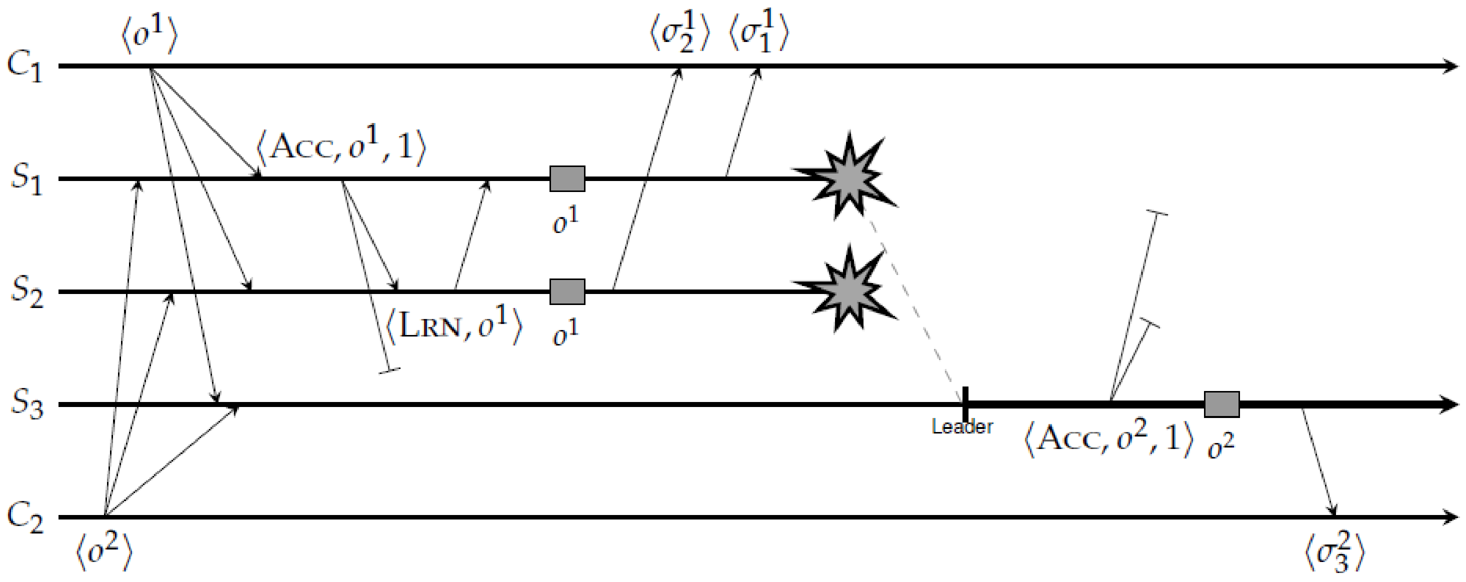
- Solution
  - Never execute an operation before it is clear that is has been learned

How about three servers & 2 crashes



- Observation
  - S_1 should not execute operation until it receives both LEARN messages from S_2 and S_3

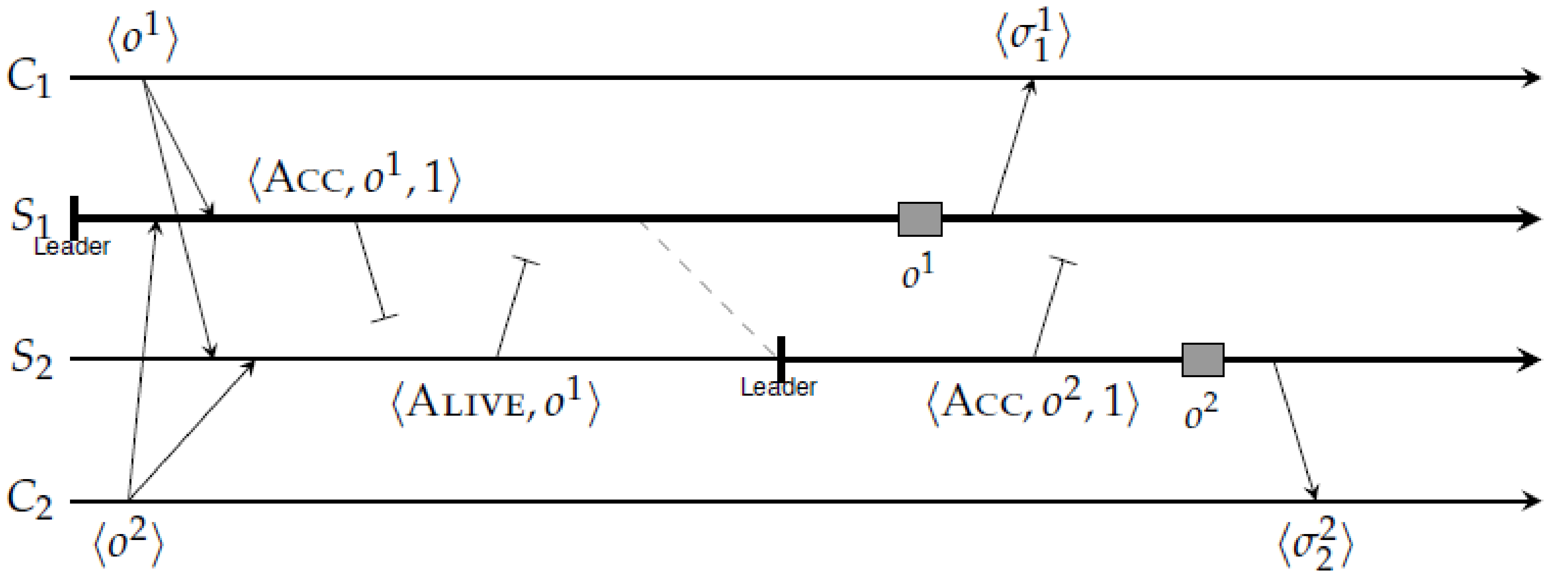
How about three servers & 2 crashes



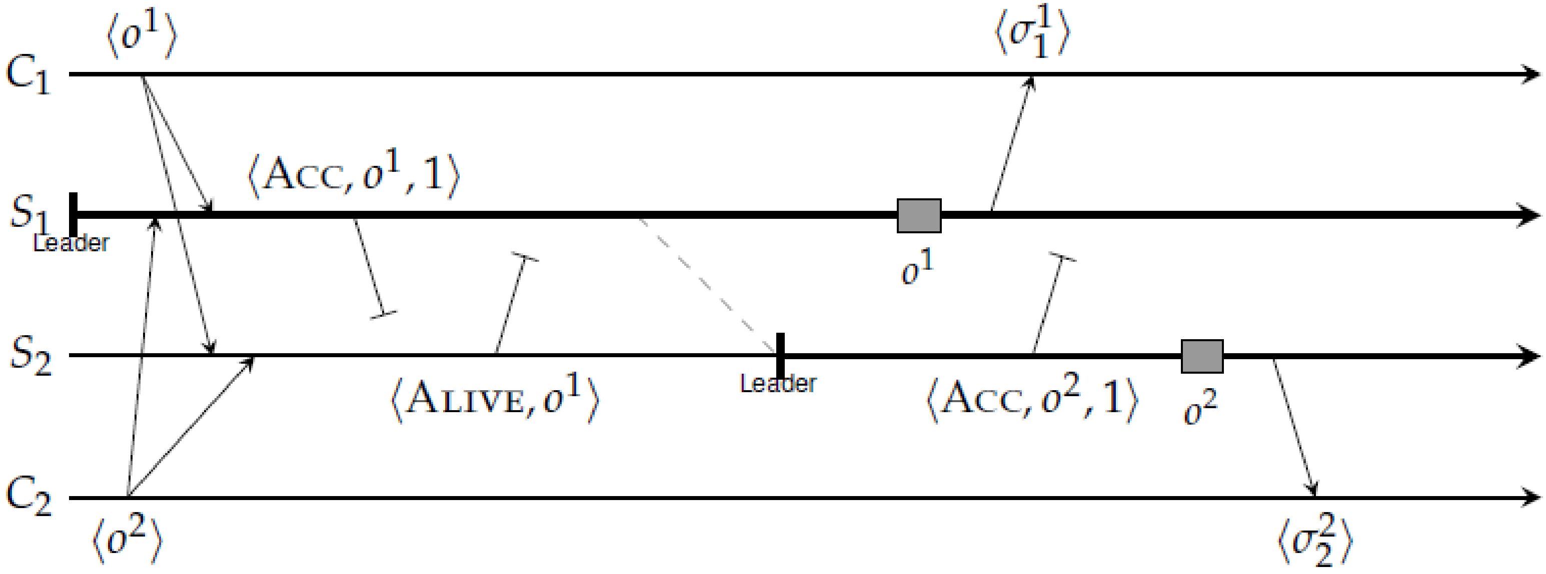
- Scenario: What happens when LEARN(o^1) as sent by S_2 to S_1 is lost?
- Solution: S_2 will also have to wait until it knows that S_3 has learned o^1.
- Paxos fundamental rule: In Paxos, a server S cannot execute an operation o until it has received a LEARN(o) from all other nonfaulty servers.

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
- Adapted fundamental rule: In Paxos with three servers, a server S cannot execute an operation o until it has received at least one (other) LEARN(o) message, so that it knows that a majority of servers will execute o.

Focusing on server crashes

- Assumption
  - Initially, S_1 is the leader.
  - When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, such as $S_1 \rightarrow S_2 \rightarrow S_3$.
  - A client cannot be asked to help the servers to resolve a situation.

- Under these assumptions, observe the following:
  - If either one of the backups (S_2 or S_3) crashes, Paxos will behave correctly because operations at nonfaulty servers are executed in the same order.

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$(o^2, 2) \Rightarrow S_3$ 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.

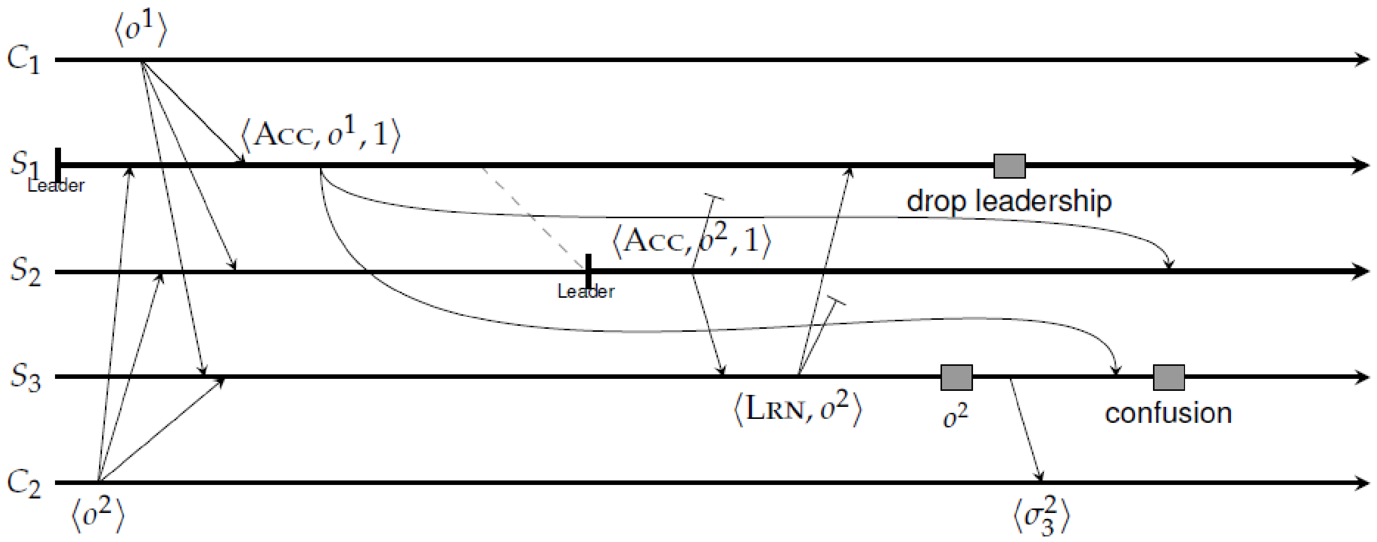
- S_2 missed ACCEPT$(o^1, 1)$
  - S_2 did detect crash and became new leader
  - S_2 sends ACCEPT$(o^1, 1) \Rightarrow S_3$ retransmits LEARN(o^1).
  - S_2 sends ACCEPT$(o^2, 1) \Rightarrow S_3$ tells S_2 that it apparently missed ACCEPT$(o^1, 1)$ from S_1, so that S_2 can catch up.

Scenario: Leader crashes after sending ACCEPT(o^1, 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 , 2), 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.
  -
- S_2 had missed ACCEPT(o^1, 1)
  - As soon as S_2 proposes an operation, it will be using a stale timestamp, allowing S_3 to tell S_2 that it missed operation o^1.

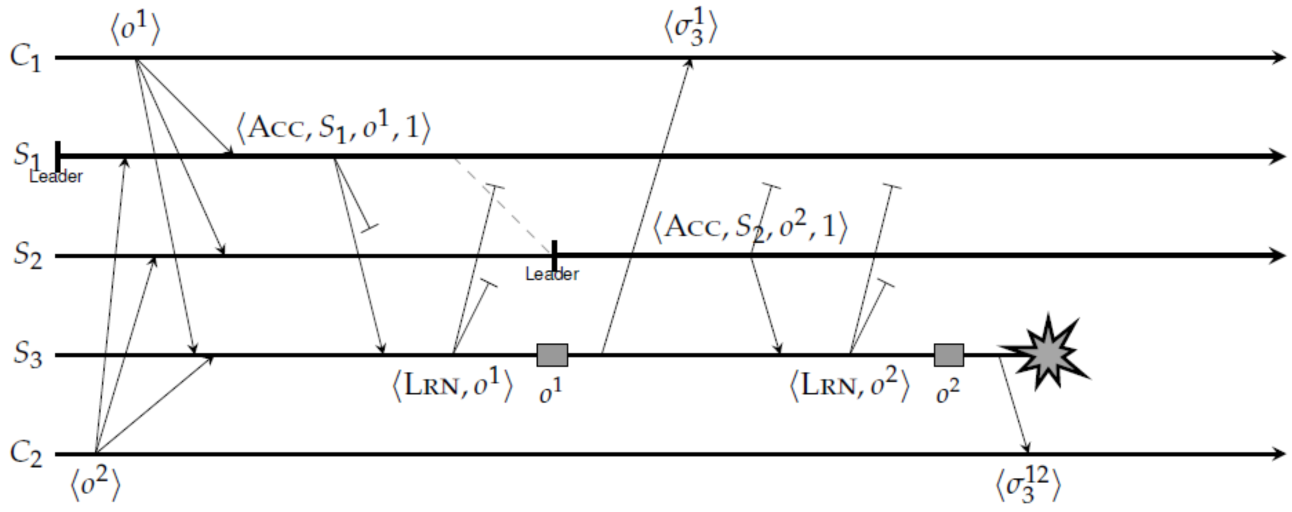
- Observation
  - Paxos (with three servers) behaves correctly when a single server crashes, regardless when that crash took place.

False crash detections



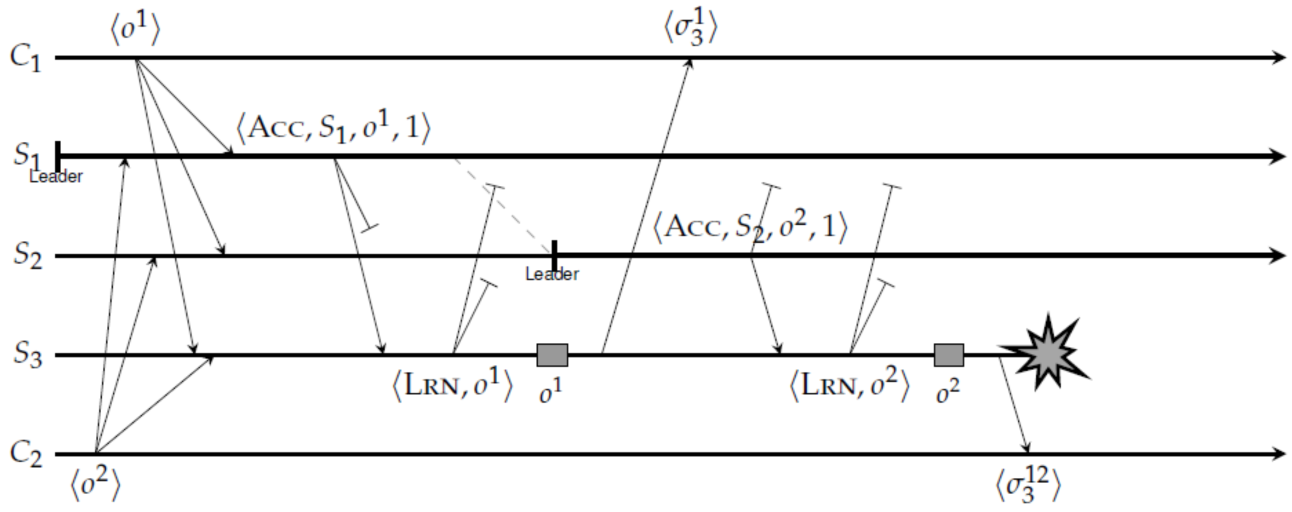
- Problem
  - S_1 ACCEPT message is delayed heavily
  - S_2 falsely detects S_1 as crashed and takes over
  - S_3 receives ACCEPT(o^1, 1), but much later than ACCEPT(o^2 , 1). Cannot do anything.
- Solution
  - If S_3 knew who the current leader was, it could safely reject the delayed ACCEPT(o^1, 1) message $\Rightarrow$ leaders should include their ID in messages.
- With these changes, Paxos behaves correctly in all cases

But what about progress?



- Paxos can still come to a grinding halt
  - LEARN messages from S_3 are lost.
  - S_1 S_2 blocked as they can never know whether S_3 executed O^1 first or O^2 first or neither
  - Clear example of safety but no liveness

Liveness in Paxos



- Root of problem
  - Leadership change happens too quickly. S_2 sends its message out before S_1 operations are completely dealt with.
- Essence of solution
  - When S_2 takes over, it needs to make sure than any outstanding operations initiated by S_1 have been properly flushed, i.e., executed by enough servers.
  - This requires an explicit leadership takeover by which other servers are informed before sending out new accept messages. (more details in book)

Paxos: Detailed protocol

- Each node runs as a *proposer*, *acceptor* and *learner*
- Proposer (leader) proposes a value and solicit acceptance from acceptors
- Leader announces the chosen value to learners

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:

Proposal Number

Round Number	Server Id

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

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 (Prepare)
  - 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

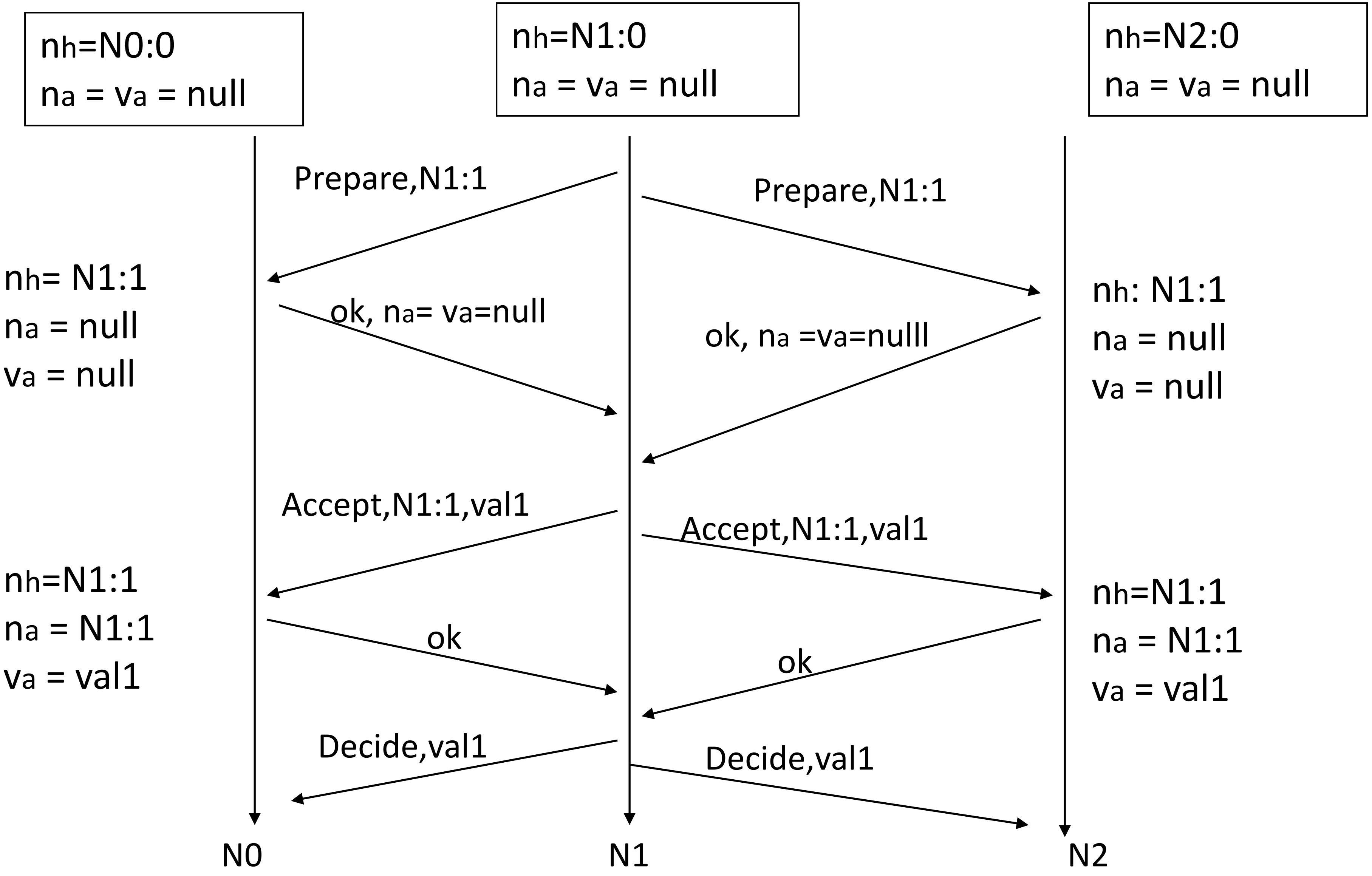
So: newer proposers win the rounds, but with old proposers' values!!!

Paxos operation: 3P protocol

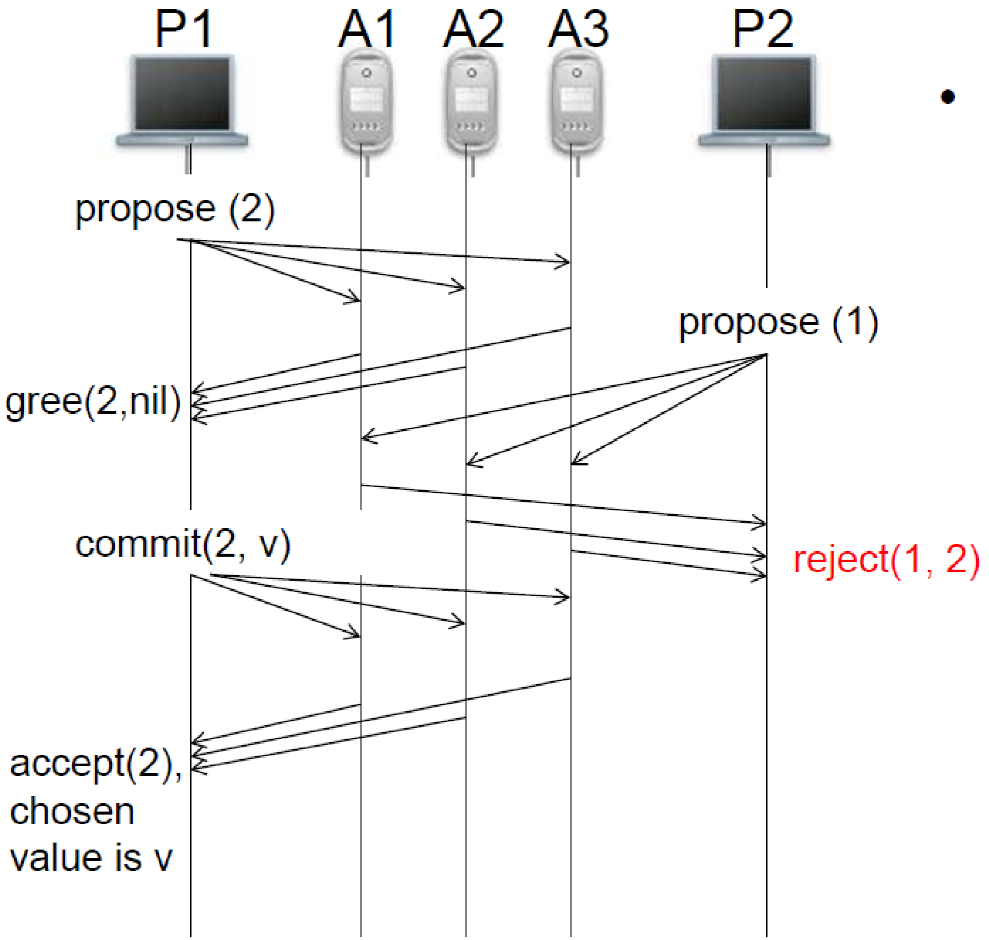
- Phase 3 (Decide)
  - If leader gets accept-ok from a majority
    - Return Done to client
    - Send <DECIDE, Va> to all nodes until you get <DECIDE, OK> back
  - If leader fails to get accept-ok from a majority
    - Delay and restart Paxos

- This phase is so all folks can learn the value chosen previously and the protocol can close

Paxos operation: an example

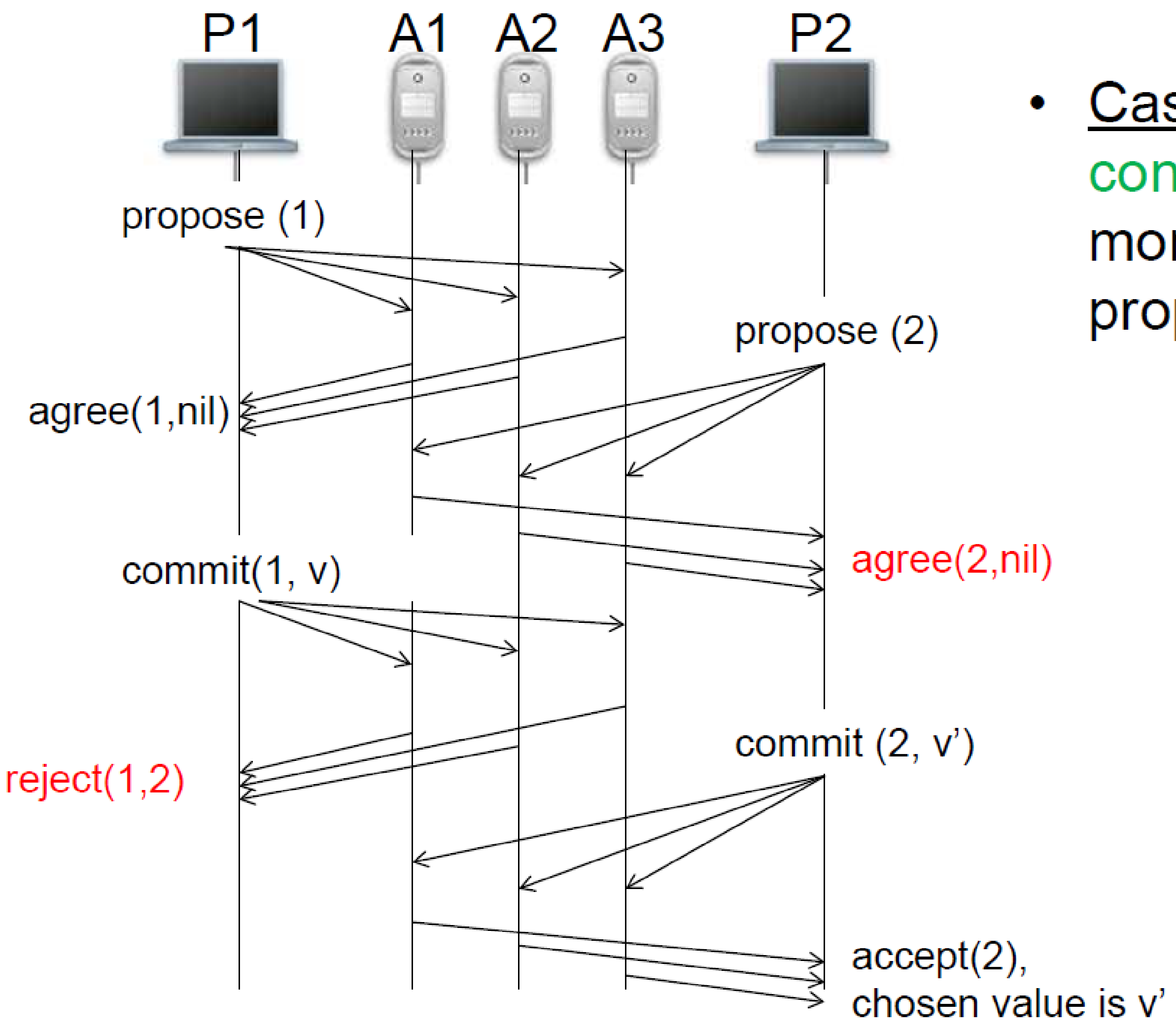


Understanding Paxos: Concurrent Proposers



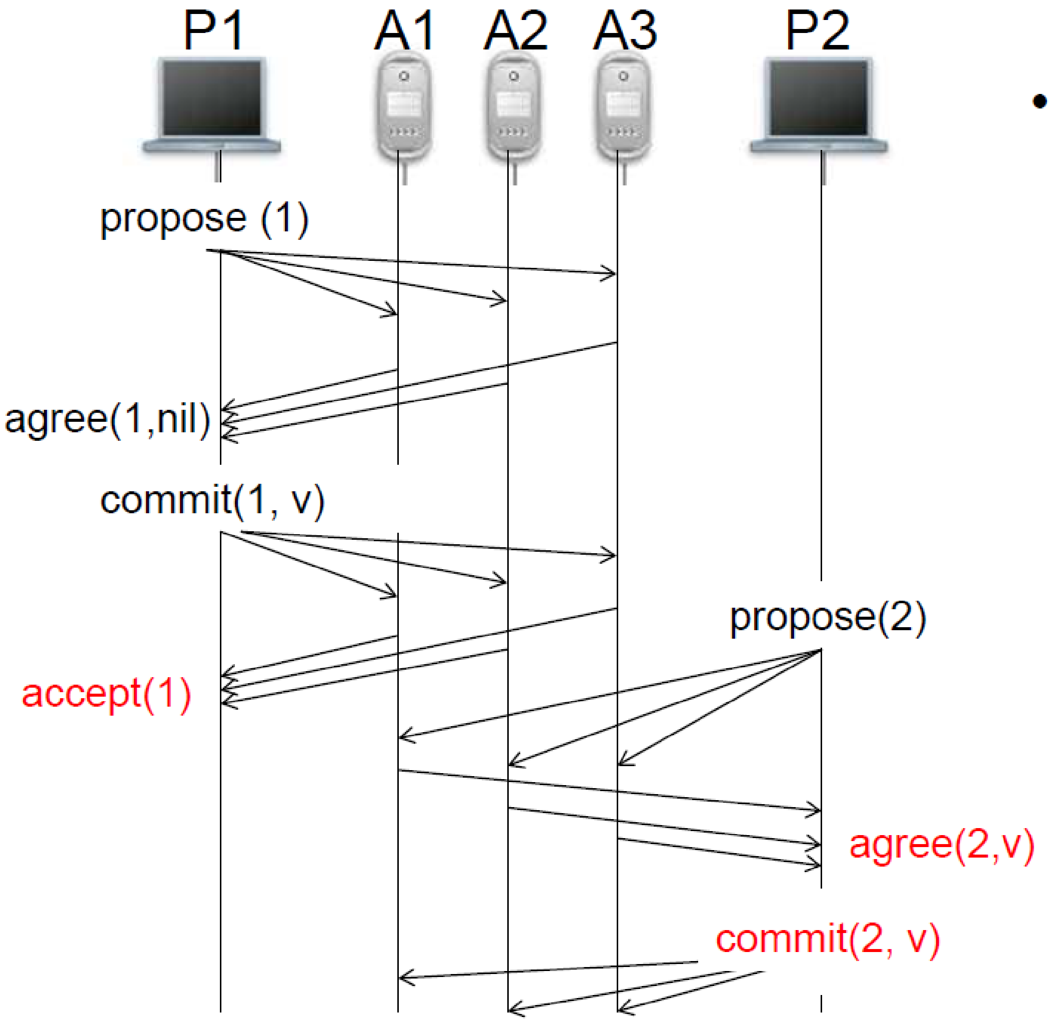
- Case 1: proposals with lower sequence numbers

Understanding Paxos: Concurrent Proposers



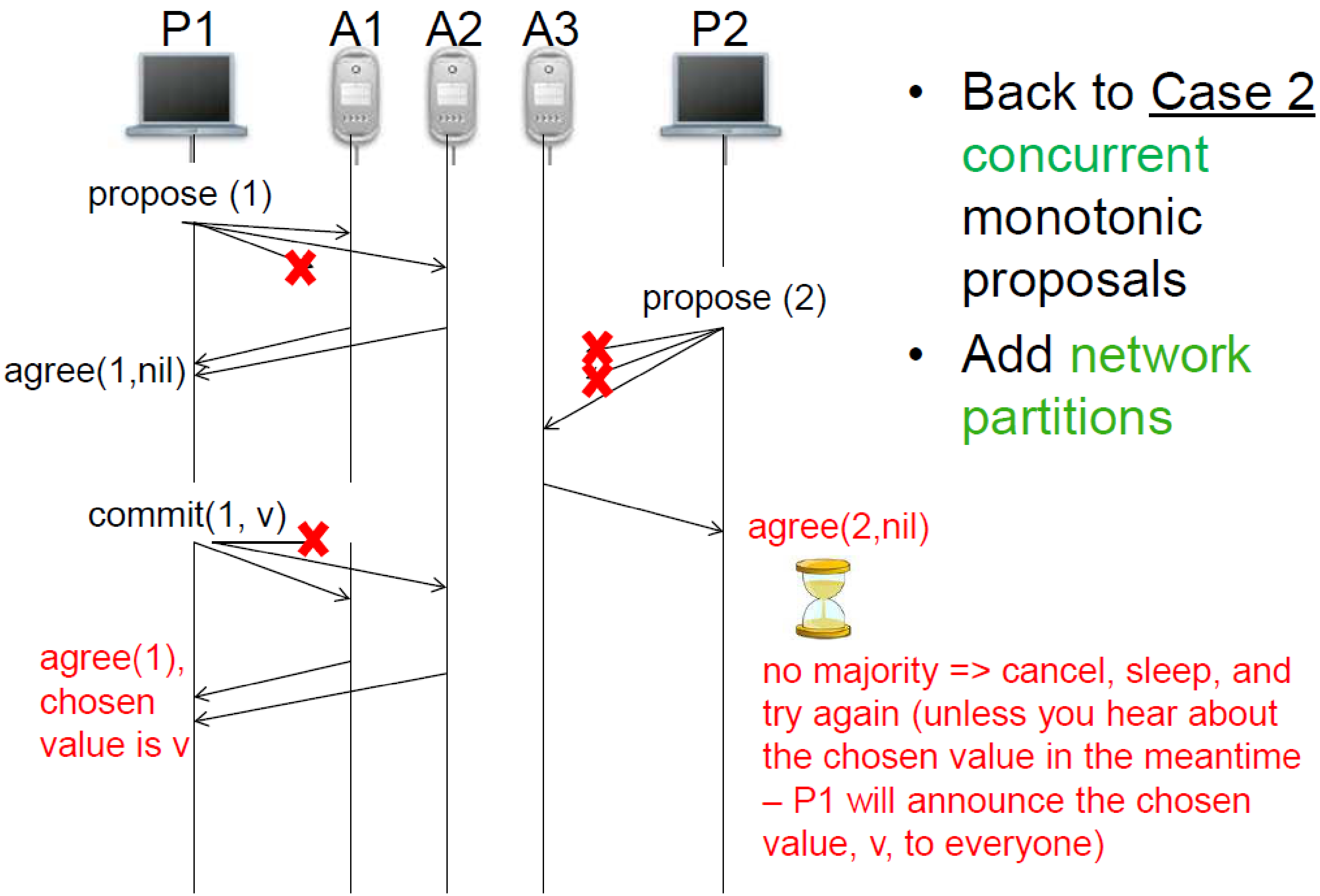
- Case 2: concurrent monotonic proposals

Understanding Paxos: Concurrent Proposers



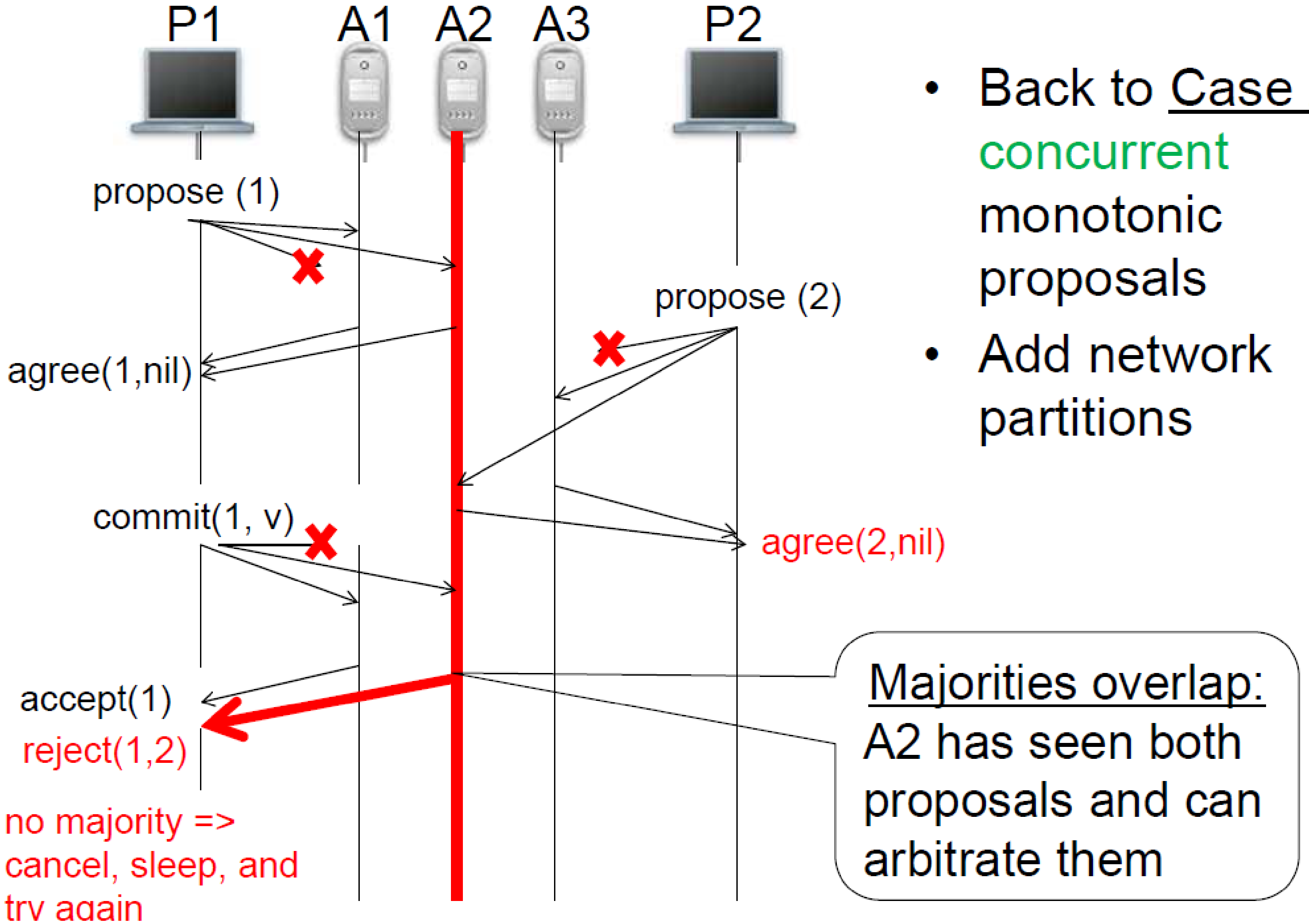
- Case 3: sequential monotonic proposals

Understanding Paxos: Concurrent Proposers



- Back to Case 2 concurrent monotonic proposals
- Add network partitions

Understanding Paxos: Concurrent Proposers



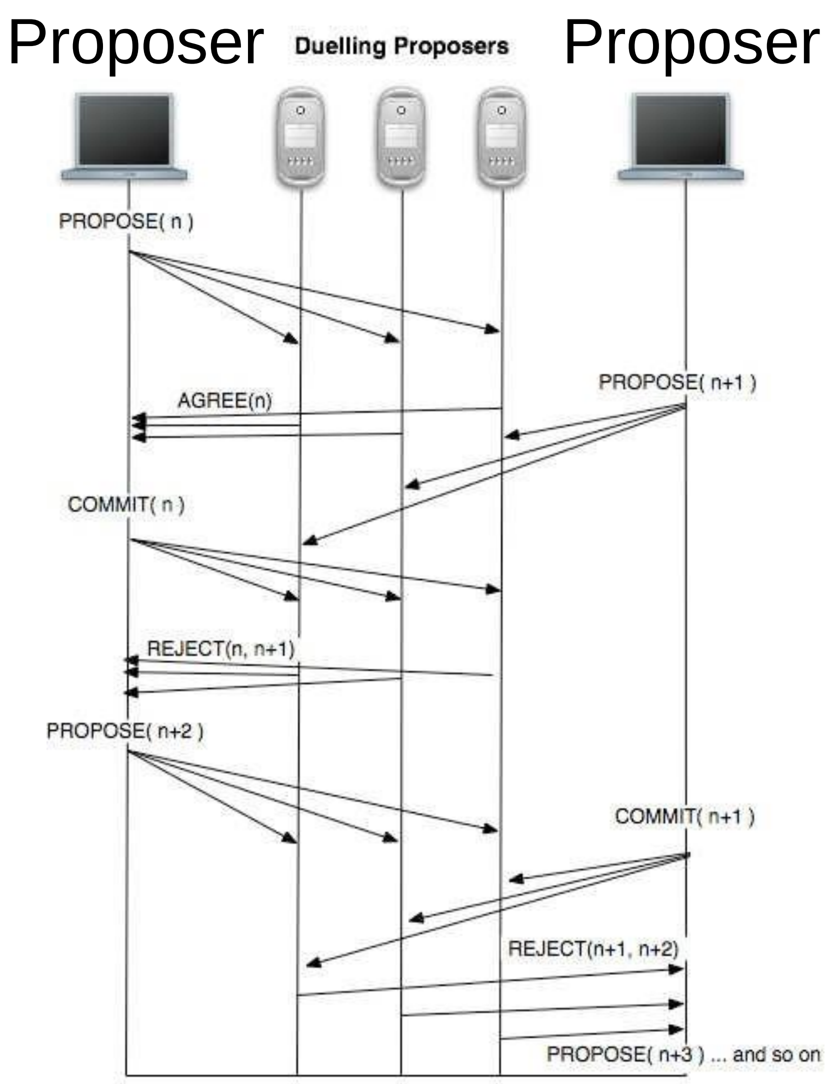
- Back to Case 2 concurrent monotonic proposals
- Add network partitions

Understanding Paxos (Potential quiz qns)

- What if leader fails while sending PROPOSE? ACCEPT?
- What if proposer fails while sending ACCEPT?
- What if a node fails after receiving ACCEPT?
  - If it doesn’t restart …
  - If it reboots …
- Try challenging Paxos with some failures for yourself
- More examples with failures
  - https://columbia.github.io/ds1-class/lectures/07-paxos-functioning-slides.pdf

Paxos May Not Terminate: Dueling proposers

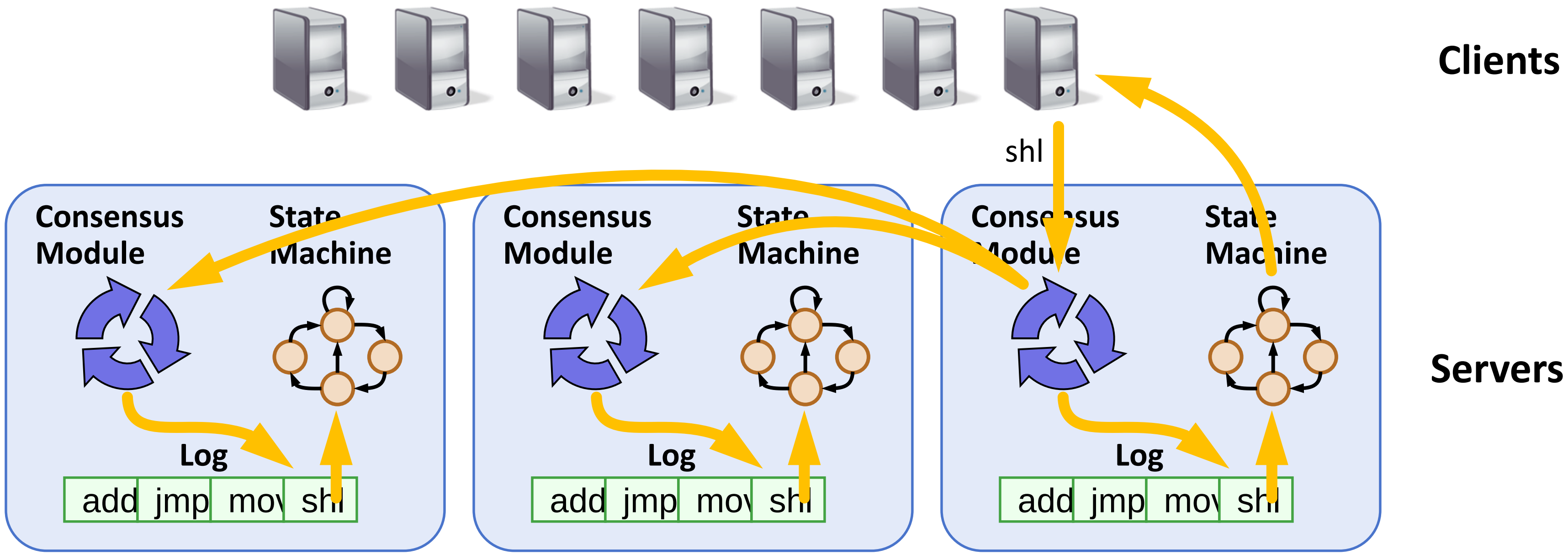
- If two or more proposers race to propose new values, they might step on each other toes all the time
  - With randomness, this occurs exceedingly rarely
- Solutions
  - Randomize delay before restarting proposers to give other proposers a chance to finish choosing
  - Or perform leader election (done by Multi Paxos)



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

Paxos for log replication

- What we saw was basic Paxos (“single decree”):
  - One or more servers propose values
  - System must agree on a single value as chosen
  - Only one value is ever chosen

- What is used is Multi-Paxos:
  - Combine several instances of Basic Paxos to agree on a series of values forming the log
  - Together with a bunch of add-ons
    - Leader election
    - Ability to add/remove nodes from cluster
    - …

Paxos vs. 2/3PC

- Paxos is similar to 2PC/3PC (but not really)
- Remember:
  - 2PC was vulnerable to 1-node failures, especially coordinator failures
  - 3PC was vulnerable to network partitions

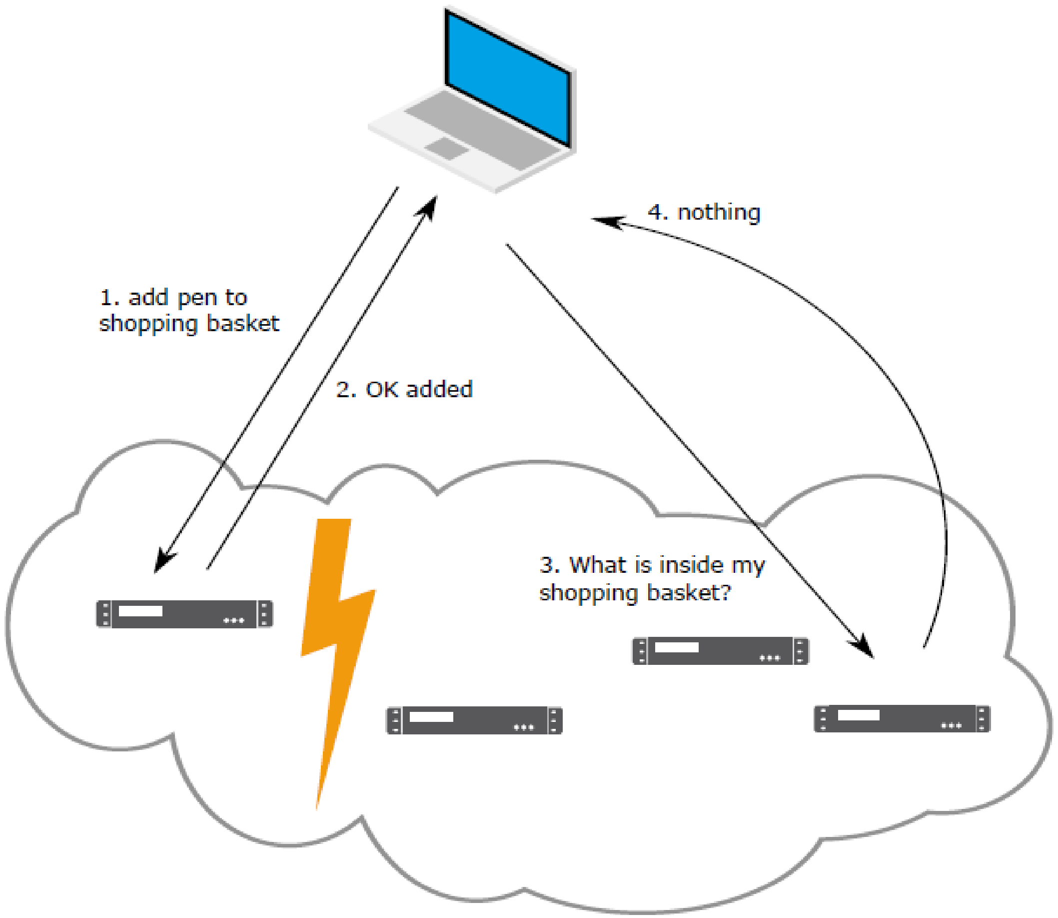
- Paxos deals with these issues using two mechanisms:
  - Egalitarian consensus: no node is special, anyone can take over as coordinator at any time
    - Hence, if one coordinator fails, another one will time out and take over
    - But that requires special ordering and acceptance protocols for proposals
  - 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 Theorem

- First stated by Eric Brewer (Berkeley) at the PODC 2000 keynote
  - Formally proved by Gilbert and Lynch, 2002[4]
- 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

CAP: Why not all three?



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

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, MemCacheDB, Scalaris, ZooKeeper
  - AP oriented: Amazon Dynamo, CouchDB, Cassandra[2], SimpleDB, Riak, Voldemort
- Thus The CAP theorem formally states the trade-offs among different distributed systems properties
  - Beware of terminology misuse (https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html)

Closing

- We have covered a lot of ground
  - Cloud computing & services: Economic fundamentals
  - Virtualization, Docker: Infrastructure fundamentals
  - MapReduce, Spark: Programming fundamentals
  - RDMBS, NFS, AFS, GFS: Storage 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 labs were entirely made possible by the TAs
  - Gia-Lac Tran
  - Rosa Candela
  - Simone Rossi
  - Jonas Wacker
- Finally, this could would not have been possible without such an attentive, inquisitive bunch of students! Thank you!

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