# Fault Tolerant Systems - 1

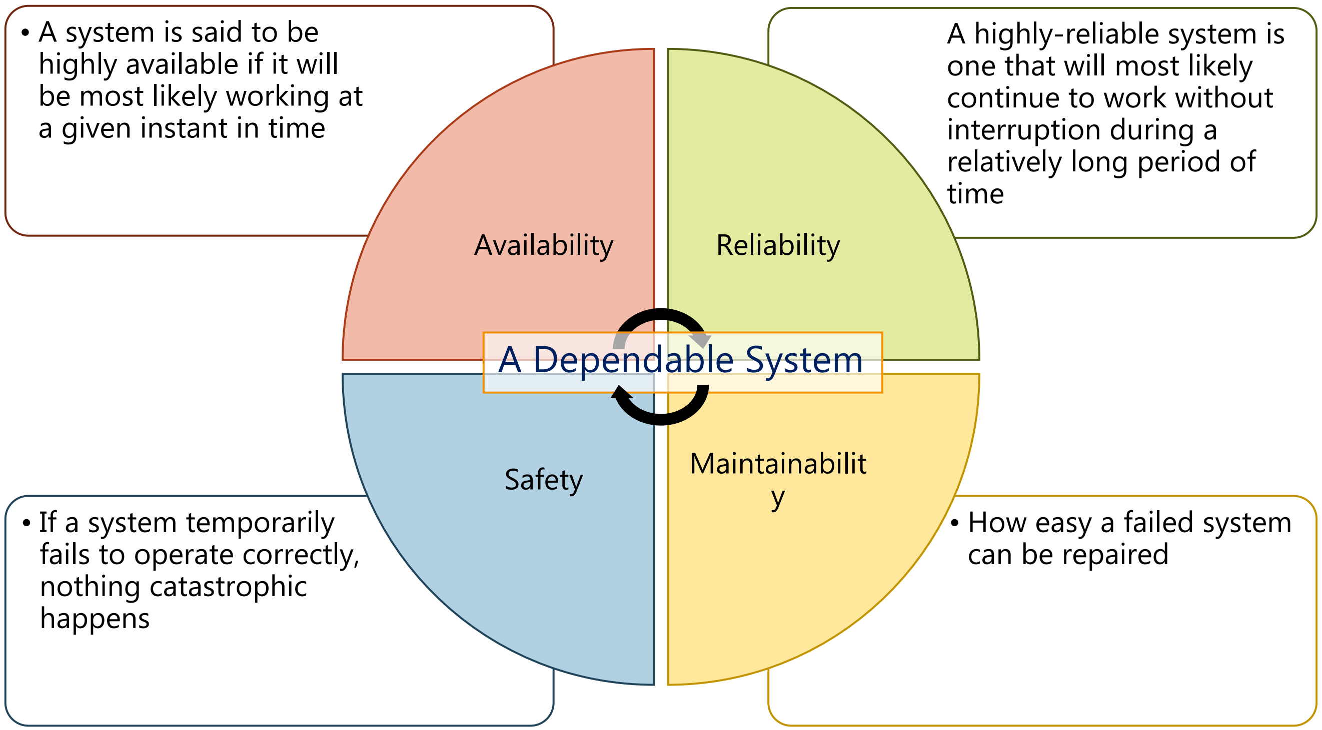
**Slide set 6**
**Distributed Systems**

Graduate Level


K. N. Toosi Institute of Technology
Dr. H. Khanmirza
h.khanmirza@kntu.ac.ir

# Concepts

- Being fault tolerant is strongly related to what is called a dependable system



# Concepts

- Reliability is defined in terms of a time interval instead of an instant in time

  - A system goes down randomly 1ms every hour
    - System is not reliable
    - System is available 99.999% of times

  - A system is shutdown two weeks a year
    - System is reliable
    - System is available for 96%

## Concepts

- A system is said to fail when it cannot meet its promises
  - If a distributed system is designed to provide a number of services, it has failed when one or more services cannot be (completely) provided

- An error is a part of a system's state that may lead to a failure
  - Example: receiver receives a erroneous packet

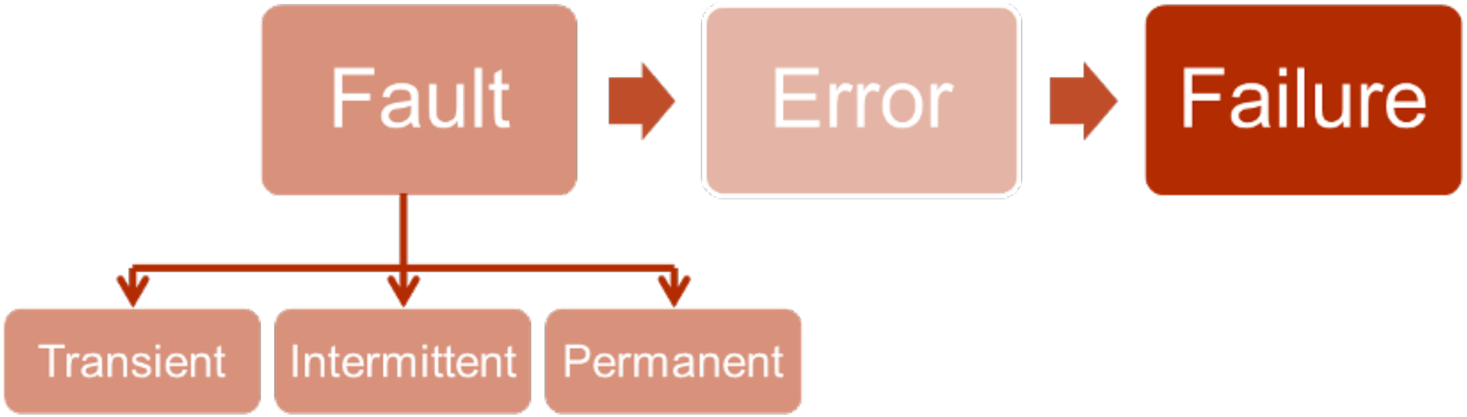
## Concepts

- The cause of an error is called a fault.
  - A crashed software is a failure which is crashes because of programming error. An uninitialized pointer is the fault of this error

- Building dependable systems relates to controlling faults
  - Preventing
  - Tolerating
  - Removing
  - Forecasting

# Concepts

- Fault tolerance
  - A system can provide its services even in the presence of faults
    - For erroneous packet, receiver can
      - Request the correct packet from sender
      - Use coding techniques to recover errors
- Fault Types



# Concepts

- Transient faults
  - Occur once and then disappear. If the operation is repeated, the fault goes away
  - Example: losing packet, but sending it again works fine

- Intermittent fault
  - Randomly repeating faults, difficult to diagnose
  - Example: concurrency and thread-interleave issues

- Permanent fault
  - Continues to exist until the faulty component is replaced
  - Example: Burnt-out chips, software bugs

## Concepts

- Partial failures
  - Specific for distributed systems
  - A partial failure may happen when a component in a distributed system fails

- An overall goal in distributed systems is to construct a system in such a way that it can automatically recover from partial failures

# Failure Models

- Crash failure
  - Server prematurely halts, but was working correctly until it stopped
  - First solution is reboot!
- Omission failure
  - When a server fails to respond to a request
  - Two types
    - Receive-omission failure: Fail to receive incoming messages
    - Send-omission failure: Fail to send outgoing messages
  - Omission failure: fails to take an action that it should have taken
  - Commission failure: takes an action that it should not have taken.

# Failure Models

- Timing failure
  - When the response lies outside a specified real-time interval
    - Sending data faster than what the client can absorb
    - Server responds too late to a request due to overload known as performance failure
- Response failure
  - Server response is incorrect, A serious failure
    - Value failure: a server simply provides the wrong reply to a request.
      - A search engine systematically returns web pages not related to any of the search terms
    - State-transition failure: when the server reacts unexpectedly to an incoming request
      - If a server receives an unknown message, a state-transition failure happens if no measures have been taken to handle such messages.

# Failure Models

- Arbitrary failures (Byzantine failures)
  - Produce arbitrary responses at arbitrary times
  - The most serious failure

# Failure Models

- Many of failure models deal with the situation that a process P no longer perceives any actions from another process Q

- Can P conclude that Q has indeed come to a halt?
  - It depends to the type of the distributed system: Synchronous, Asynchronous or Partial-Synchronous

# Halting Failures

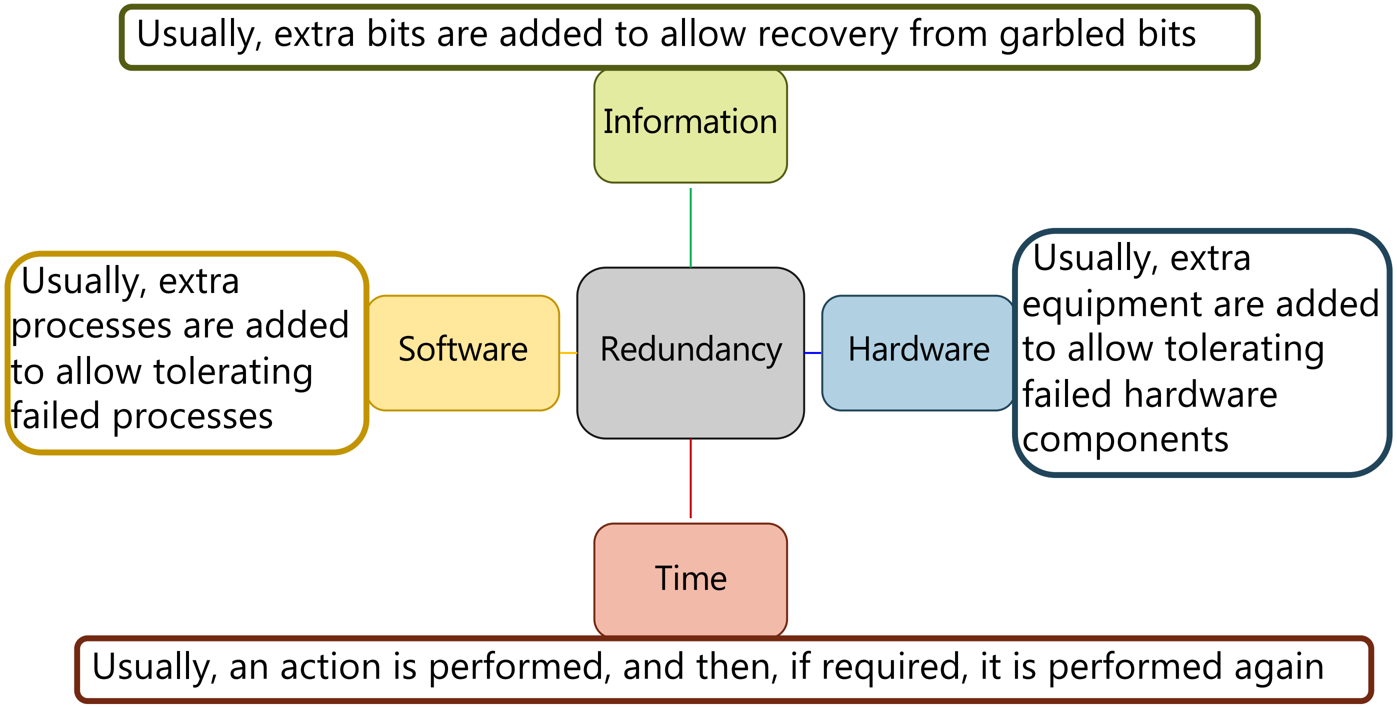
- Fail-stop
  - Crash failures that can be reliably detected
  - This may occur when assuming non-faulty communication links and when the failure-detecting process P can place a worst-case delay on responses from Q
- Fail-noisy
  - Like fail-stop, but P eventually come to the correct conclusion that Q has crashed
  - Some unknown time in which P’s detections of the behavior of Q are unreliable

# Halting Failures

- Fail-silent
  - Communication links are nonfaulty, but process P cannot distinguish crash failures from omission failures
- Fail-safe
  - Dealing with arbitrary failures by a process, but these failures are kind: they cannot do any harm
- Fail-arbitrary
  - Q may fail in any possible way; failures may be unobservable in addition to being harmful

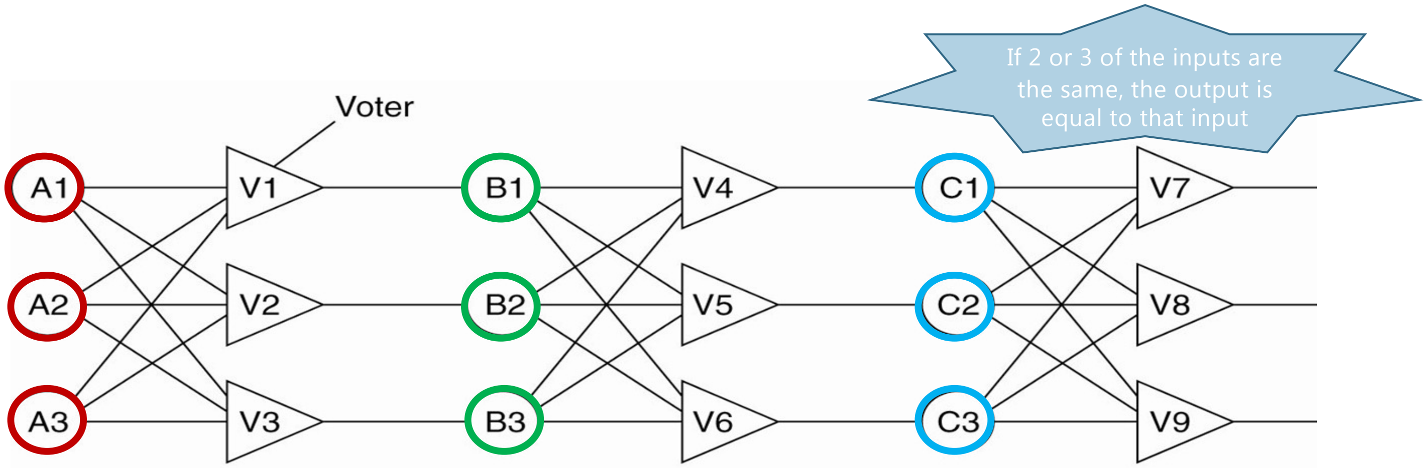
# Redundancy

- The key technique for masking faults is to use redundancy



# Triple Modular Redundancy Sample

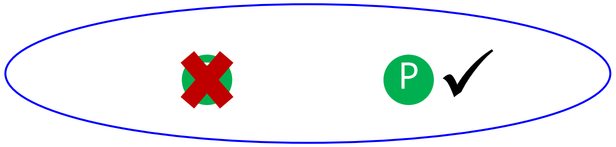

A circuit with signals passing through devices A, B, and C, in sequence




Each device is replicated 3 times and after each stage is a triplicated voter

# Process Fault Tolerance

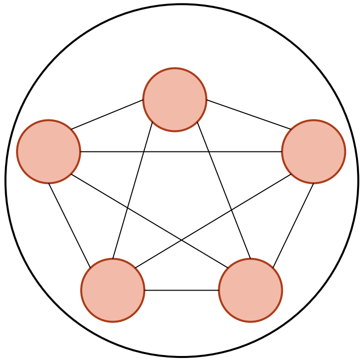
- Use Process Redundancy
  - Organize several identical processes into a group
    - Messages received by all members of the group
    - Failure of one to several processes in the group does not halt the whole system

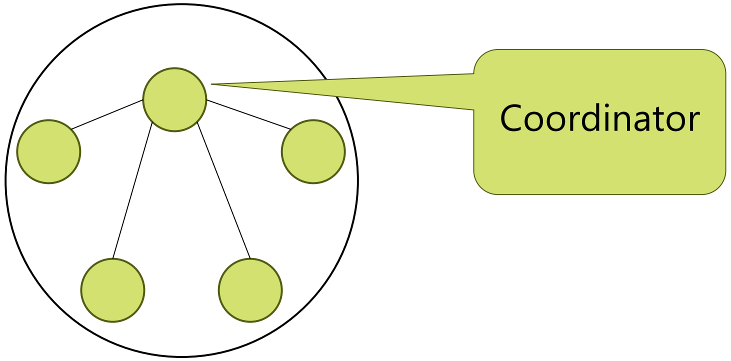



- Notes:
  - A process can join a group or leave one during system operation
  - A process can be a member of several groups at the same time
  - Mechanisms are needed for managing groups and group membership.

# Process Redundancy Models

- An important distinction between groups is their internal structure






To decide anything, a vote often has to be taken, incurring some delay and overhead

**Flat Group:**

(+) Symmetrical
(+) No single point of failure
(-) Decision making is complicated

**Hierarchical Group:**

(+) Decision making is simple
(-) Asymmetrical
(-) Single point of failure

# Group Membership Management

- Central group membership server
  - Easy to implement
  - Has single point of failure problem

- Distributed membership management
  - Servers are added to a multicast group
  - For join, a process can send a membership request to the whole group
  - To leave send a goodbye message to all members
  - In failures, member cannot commit a polite goodbye, other members will have to detect and report to other member groups

# Replication Protocols

- Primary-based protocols

- Replicated-write protocols

# Replication Protocols

- Primary-based replication
  - Hierarchical group
  - Primary-backup coordinates write operations
  - Needs election algorithms when primary backup fails

- Replicated-write protocols
  - Organize identical processes into a flat group
  - Needs voting for decision (quorum-based)

# Replication Protocols

- How much replication is sufficient?

## k-fault-tolerant system

If a system can survive faults in k components and still meet its specifications

## Masking k-failures (k-fault tolerancy)

- If faults are crash or omission, then k+1 components is enough

- If faults are arbitrary, 2k+1 components is needed. (Why?)

# Consensus Problems

Also known as **Agreement Problems**

## Consensus Problem

- In a fault-tolerant process group, all non-faulty processes execute the same commands, in the same order
- This means group members need to reach consensus on which command to execute
- Reaching consensus is easy when no failure happen

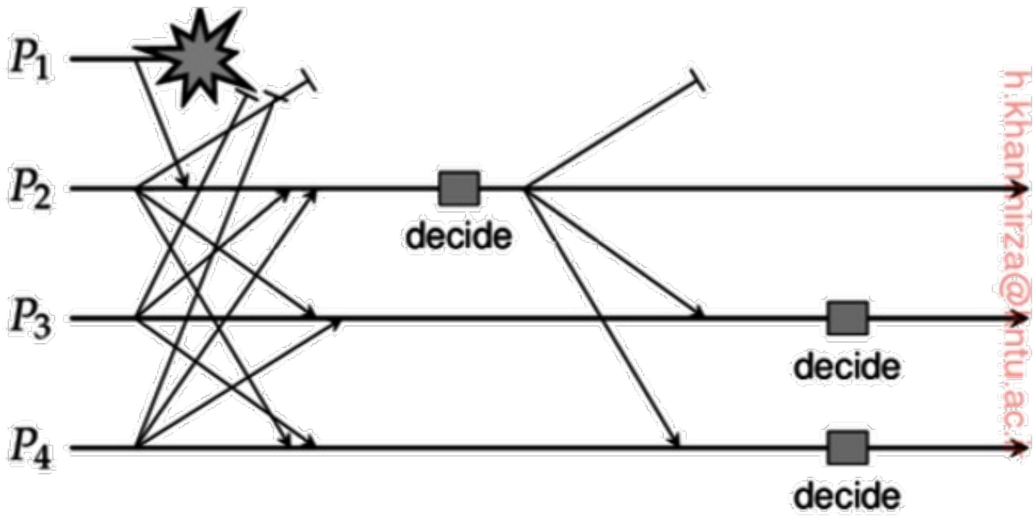
# Flooding Consensus

- Assumes Fail-Stop failures
- Algorithm operates in rounds

- Clients send their proposals to a group of processes $P=\{P_1, P_2,\ldots\}$
- At each round
  - Processes send their list of commands to all members
  - All processes merge all lists
    - All processes run a similar sorting algorithm, then all processes select the same command
  - Processes received commands from all others, broadcast their decision to others

# Flooding Consensus Example

- $P_1$ crashes, but before crash it sends its list. $P_2$ receives the list but $P_3, P_4$ do not receive the list
- $P_3$ detects $P_1$ failure but does not know if others have detected the failure or not
- $P_3$ knows that if other process has received $P_1$ list it will decide and send the decision to all
- $P_3, P_4$ do nothing, but $P_2$ do the decision and its decision to all
- In the next round, $P_3$ *and* $P_4$ can decide based on $P_2$ list



# Flooding Consensus

- This model works for fail-stop failures even with only one working process

- What if $P_3$ could not detect the failure of the $P_1$ for sure?

# Distributed Commit

# Distributed System Algorithms Properties

- Liveness
  - In all conditions algorithms reaches a steady state
- Safety
  - In all conditions with any input, algorithm does not violate initial assumptions

# Distributed Commit

- A set of operations should be performed by all group members or none at all
- All processes should execute operations in the same order
- The problem first was encountered in database systems
  - Suppose a database system is updating some complicated data structures that include parts residing on more than one machine.

- Assumptions:
  - Concurrent processes and uncertainty of timing, order of events and inputs (asynchronous systems)

  - Failure and recovery of machines/processors, of communication channels

# One-phase Distributed Commit

- A coordinator (= primary) sends an operation(s) to all participants (= backups)
- Each participant executes the operation
- The simplest Solution
- Problem:
  - No way to report back the failure of execution to coordinator!

# Two-Phase Commit (2PC)

- Phase 1: prepare phase
- Phase 2: commit phase

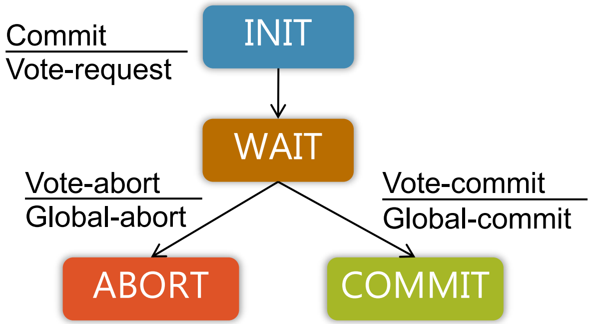
# Two-Phase Commit (2PC)

- Phase 1: prepare (voting) phase
  - A: Coordinator asks participants if they can execute the operation (VOTE-REQUEST)
  - B: Participants reply
    - VOTE-COMMIT: if they can execute operation
    - VOTE-ABORT: if they cannot execute the operation

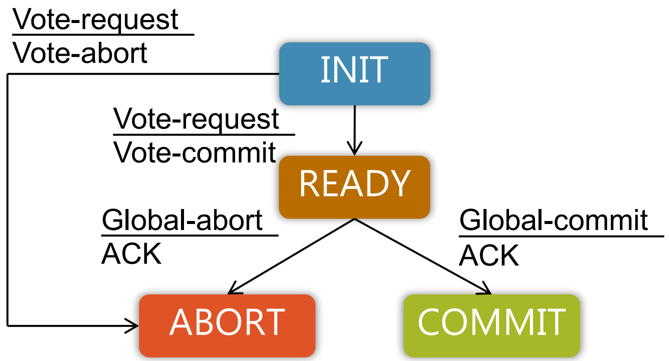
# Two-Phase Commit (2PC)

- Phase 2: commit (decision) phase
  - A: coordinator collects all votes
    - Sends GLOBAL-COMMIT: if all participants agree.
    - Sends GLOBAL-ABORT: if even one participant not agree.
  - B: Each participant
    - Commits locally if receive GLOBAL-COMMIT
    - Aborts transaction locally if receive GLOBAL-ABORT

# Two-Phase Commit (2PC)




The finite state machine for the **coordinator** in 2PC




The finite state machine for a **participant** in 2PC

- Looking to finite state machine of coordinator and participants, they have three waiting states.
- To avoid forever blocking, both use timeouts

# Two-Phase Commit (2PC)

- Blocking 1: participants waiting for receiving VOTE-REQUEST
  - After timeout, participants send VOTE-ABORT message

- Blocking 2: coordinator waits for vote replies
  - After some time if not all votes collected, it sends GLOBAL-ABORT message.

# Distributed Commit – 2PC

- Blocking 3: participants in ready state wait for coordinator reply
  - Participants cannot decide by themselves!

  - Simple Solution: Block until coordinator reboot and recover

  - Cooperative protocol: ask other participants
    - If the other is in COMMIT state → do commit
    - If the other is in ABORT state → do abort
    - If the other is in INIT state → do abort
      - The other node didn't receive vote-request message, or coordinator has crashed before sending to it
    - If the other is in READY state → contact another participant! (may block)

# Distributed Commit – 2PC

- Blocking 3: participants in ready state wait for coordinator reply (cont.)
  - Cooperative protocol: ask other participants (cont.)
    - If the other is in READY state → contact another participant! (may block)
    - If all are ready:
      - Since some of participants may crash which was received COMMIT/ABORT command
        - If it reboots and it will be in commit state
          - Other participants must wait for coordinator or this participant, but how long should they wait?
        - If it never restarted?

# Distributed Commit – 2PC

- To ensure recovery, coordinator and participants must log their state in disk.

- 2PC is blocking commit protocol
  - Blocking 3 scenario
  - If one or more machines fail (we need all to reply)

- 2PC is safe but not live.

## Three-Phase Commit (3PC)

- Avoids blocking processes in the presence of fail-stop crashes

- It is not applied often in practice as the conditions under which 2PC blocks occur rarely

- Phase 1: prepare phase
- Phase 2: pre-commit phase
- Phase 3: commit phase

# Three-Phase Commit (3PC)

- Phase 1: prepare phase
  - A: Coordinator asks participants if they can execute the operation VOTE-REQUEST
  - B: Participants reply
    - VOTE-COMMIT: if they can execute operation
    - VOTE-ABORT: if they cannot execute the operation

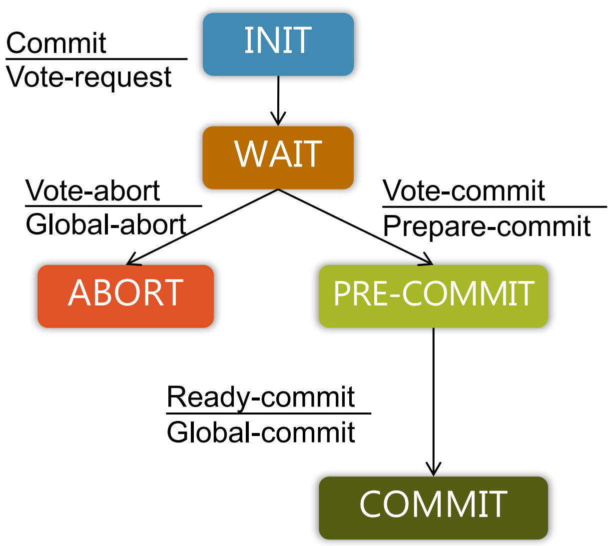
# Three-Phase Commit (3PC)

- Phase 2: pre-commit phase
  - A: coordinator collects all votes
    - Sends PREPARE-COMMIT: if all participants agree
    - Sends GLOBAL-ABORT: if even one participant not agree
  - B: Each participant
    - Send READY-COMMIT if receive PREPARE-COMMIT

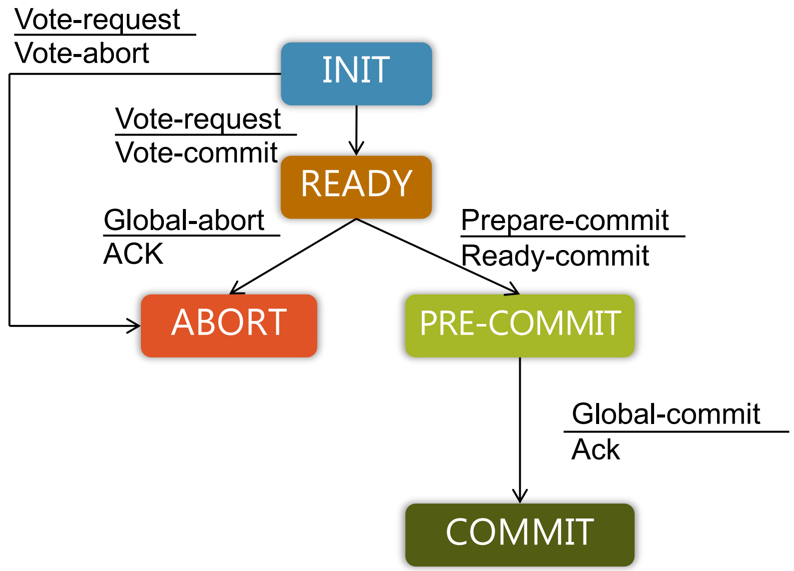
# Three-Phase Commit (3PC)

- Phase 3: commit phase
  - A: coordinator collects all READY-COMMIT messages
    - Sends GLOBAL-COMMIT: if all participants are prepared
  - B: Each participant
    - Commit locally if receive GLOBAL-COMMIT

# Three-Phase Commit (3PC)




The finite state machine for the **coordinator** in 3PC




The finite state machine for a **participant** in 3PC

# Three-Phase Commit (3PC)

- We skip the blocking scenarios similar to 2PC

- Blocking 1: coordinator is blocked in PRE-COMMIT
  - One or more processes have crashed, but they have voted for commit → coordinator sends GLOBAL-COMMIT

  - Crashed participator can be recovered by a recovery protocol, later.

# Three-Phase Commit (3PC)

- Blocking 2: Participant blocked in READY or PRE-COMMIT.
  - This means coordinator has failed, then contacts with other participants
    - If in COMMIT → commit
    - If in PRE-COMMIT → commit
    - If in INIT → abort (because no participant reached to PRE-COMMIT)
    - If (majority) in READY → abort
      - A participant is crashed but no one knows what was the state of the crashed participant
      - If it recovers to INIT then it was aborted
      - If it was recovered to PRE-COMMIT, nothing harmful may be done
    - This situation is the major difference with 2PC. In 3PC no crashed participant may recover to COMMIT, thus they can come to agreement

# Three-Phase Commit (3PC)

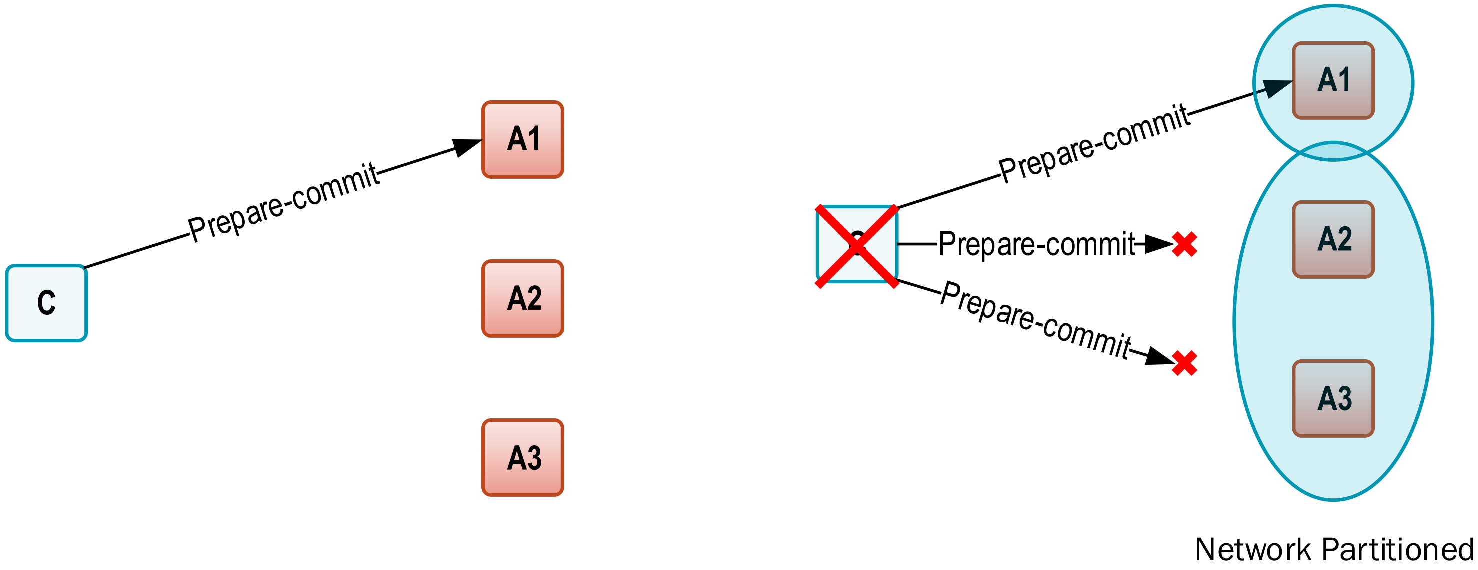
- Liveness: it always makes progress

- Safety: No!
- 3PC results in inconsistent state between replicas when network is partitioned.

- 3PC trades safety for liveness

# Three-Phase Commit (3PC)

- Safety:
  - C after sending prepare-commit, crashes and network is partitioned
  - A1 commits but, A2 and A3 will abort



# FLP Impossibility

- Impossibility of distributed consensus with one faulty process
  - FLP Fischer-Lynch-Paterson (FLP) • M.J. Fischer, N.A. Lynch, and M.S. Paterson, Journal of the ACM, 1985.

- What FLP says: you cannot guarantee both safety and progress when there is even a single fault at an inopportune moment to reach a consensus.

- What FLP does not say: in practice, how close can you get to the ideal.

# Paxos

# Paxos

- Solving 2PC Problems
  - We should not rely on all participants → we can use majority vote
  - Having one coordinator is a real issue → Add more coordinators
- This makes Paxos algorithm!

## Paxos

- 1979, 2PC, Gray
- 1981, 3PC, Stonebraker
- 1989, 42-page tech report
- 1990, “Part-time Parliament”
  - Paper rejected, ACM Transactions on Computer Systems
  - It was not considered a useful algorithm
- 1996, First implementation
- 1997, Used in Frangipani Distributed Lock
- 1998, Paper resubmitted and accepted TOCS
  - Won ACM SIGOPS Hall of Fame Award in 2012!
- 2001, ”Paxos Made Simple”, Lamport
- 2007, “Paxos Made Live”, Chandra
- 2014 RAFT appears

# Paxos

- Paxos is everywhere
  - Yahoo's ZooKeeper (Now an Apache project)
  - Google's Chubby (Distributed Lock)
  - Frangipani (Distributed lock service)
  - Amazon Web Services uses Paxos
  - Windows Fabric, used by many of the Azure services, make use of the Paxos algorithm for replication between nodes in a cluster
  - Neo4j HA graph database implements Paxos, replacing Apache ZooKeeper used in previous versions.
  - Apache Mesos uses Paxos algorithm for its replicated log coordination
  - …

# Paxos

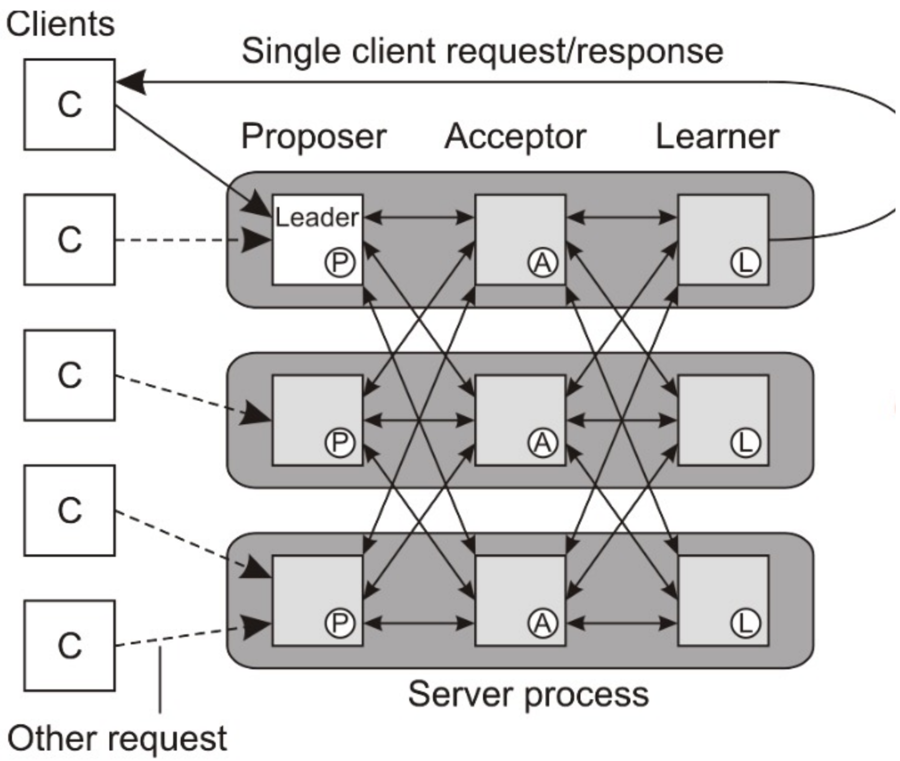
- Assumptions
  - The distributed system is partially synchronous
  - Communication is unreliable, messages may be lost, duplicated, or reordered
  - Corrupted messages can be detected
  - All operations are deterministic, once an execution is started, it is known exactly what it will do.
  - Processes may exhibit fail-noisy failures, but not arbitrary failures, nor do processes collude

# Paxos

- Paxos lets all nodes agree on an operation despite node failures, network failures and delays

- Paxos works correctly when less than N/2 nodes fail

# Paxos




- A single machine (server) has three components
- Proposer
  - Handles clients' request
  - Suggest proposals for acceptors
- Acceptor
  - Receives proposals
  - Accepts or reject proposals
- Learner
  - Learns the operation chosen by majority
- When one of these components crashes, server is considered as crashed

# Paxos

- A clients requests an operation
- Proposers receive and handle requests of clients one at a time
- A Proposer creates a proposal and sends to acceptors
- If majority of acceptors accept the same proposal, it is said to be chosen.

# Paxos

- Multiple proposers and Multiple acceptors
- It is possible proposers never get the vote of majority

- Solution: Let several proposals is accepted by acceptors

- How acceptors distinguish proposals from each other?
- Tag each proposals with a unique number (=N)
  - Each proposer generates a unique number never has generated before

- Proposals are tagged with (proposerID, N) pair as Proposal Number

# Paxos

- The proposer with the highest proposal number is the leader proposer
  - Paxos embeds a Distributed Leader Election process with proposal numbers that are, in fact, Lamport logical clock

- The proposal with the highest proposal number will have the majority
- Or
- Acceptors always choose (agree with) the operation with the highest proposal number

- If another proposer transmit a proposal with any higher number than the current chosen proposal, becomes the leader or its proposal is chosen.

# Paxos

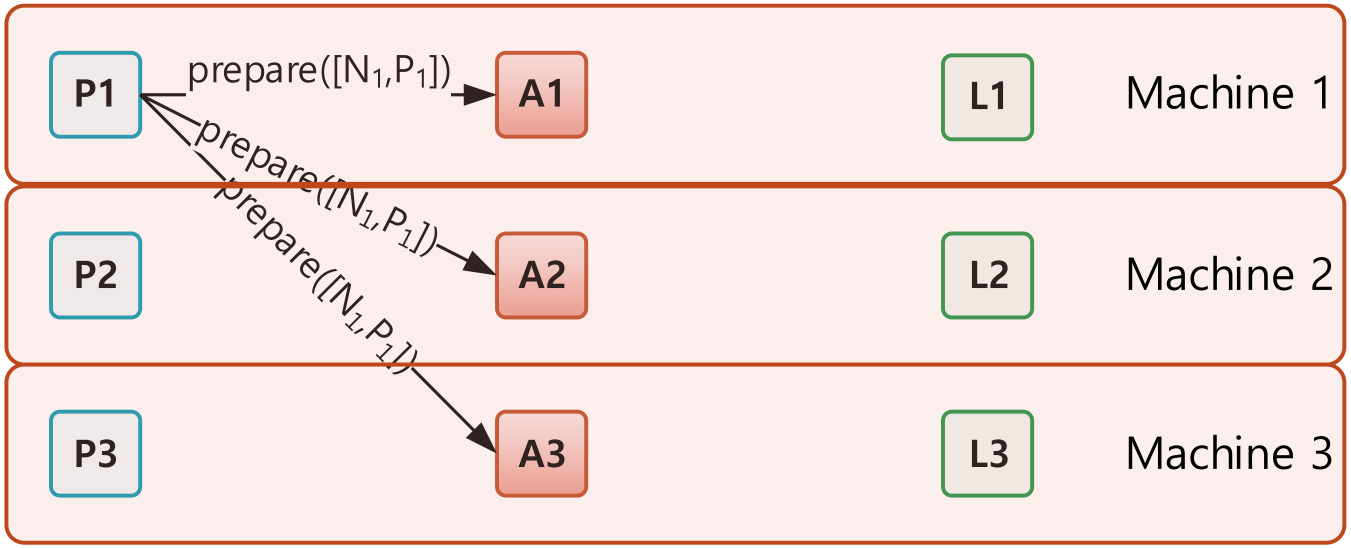
- Detailed Algorithm
- Phase 1a - prepare phase
- Phase 1b - promise phase
- Phase 2a - accept phase
- Phase 2b - accepted phase

# Paxos

- Phase 1a - prepare phase

- A proposer, P selects a proposal number N and sends a prepare request with number $(N_i, P_i)$ to majority of acceptors

# Paxos

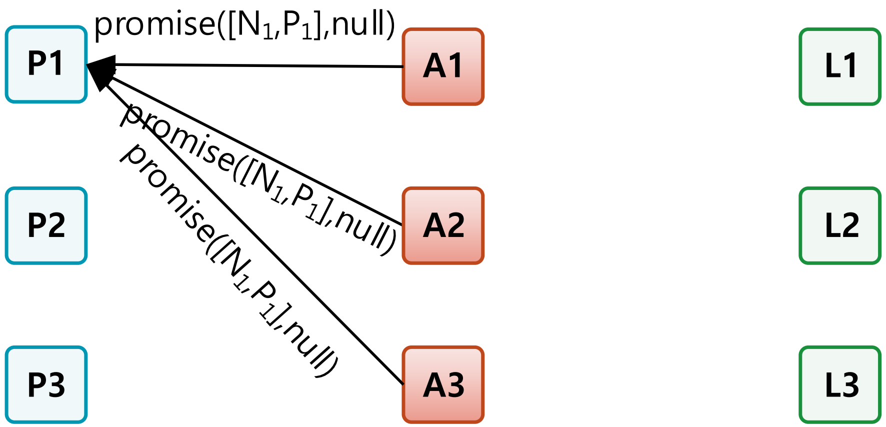

P1
prepare([N1,P1])
A1
L1
Machine 1
prepare([N1,P1])
prepare([N1,P1])
P2
A2
L2
Machine 2
P3
A3
L3
Machine 3

# Paxos

- Phase 1b - promise phase
- If an acceptor receives $(N_i,P_i)$
  - If this is the first proposal, sends promise to the proposer
  - Has promised to proposal $(N_j,P_j)$
    - If $(N_i,P_i) < (N_j,P_j)$, acceptor sends promise with number $(N_j,P_j)$, the highest-numbered proposal it has accepted so far
    - For optimization no of messages, acceptor may not reply
    - If $(N_i,P_i) > (N_j,P_j)$, acceptor promises $P_i$, sends a promise with number $(N_i, P_i)$
    - It promises not to accept any proposal with number less than $(N_i, P_i)$

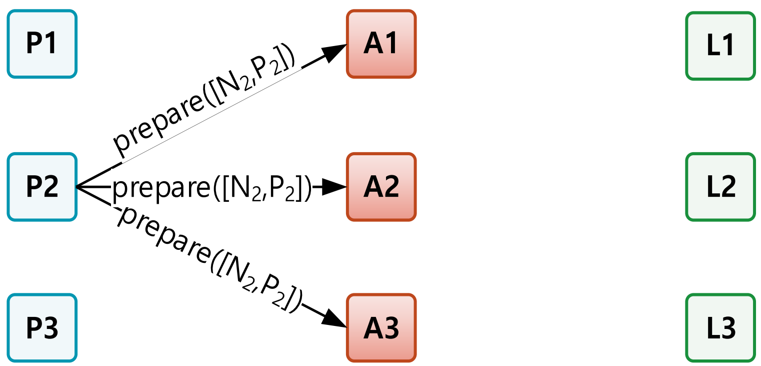
# Paxos

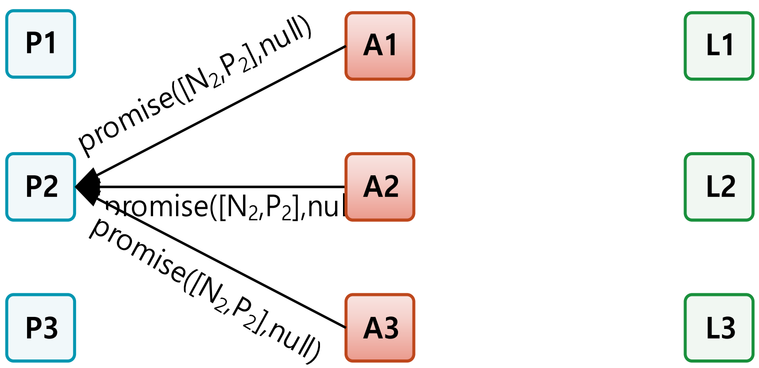
- Phase 1b - promise phase
  - There is no accepted operation to be announced, they return null.
  - $P_1$ is leader



# Paxos

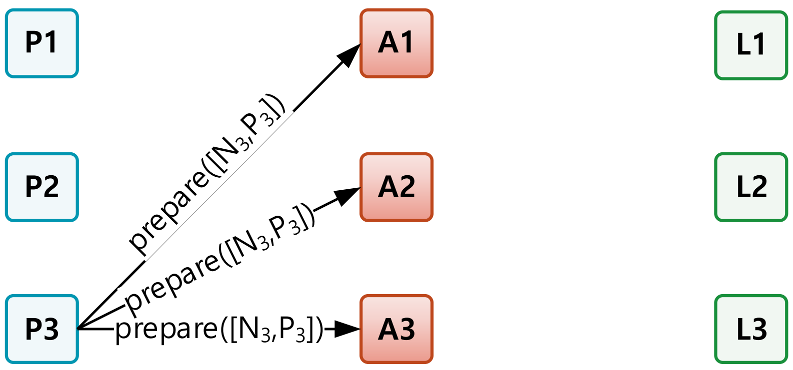
- $N_2 > N_1$ ➔ Now P2 is Leader

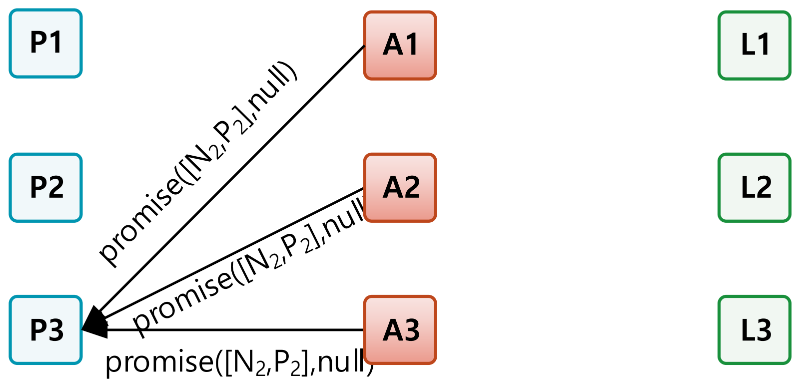


# Paxos

- $N_3 < N_2$


P1
P2
P3
A1
A2
A3
L1
L2
L3
prepare([N3,P3])
prepare([N3,P3])
prepare([N3,P3])


P1
P2
P3
A1
A2
A3
L1
L2
L3
promise([N2,P2],null)
promise([N2,P2],null)
promise([N2,P2],null)

# Paxos

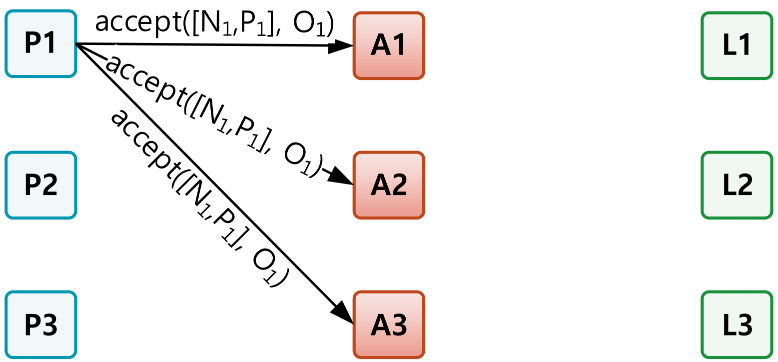
- Phase 2a - accept phase

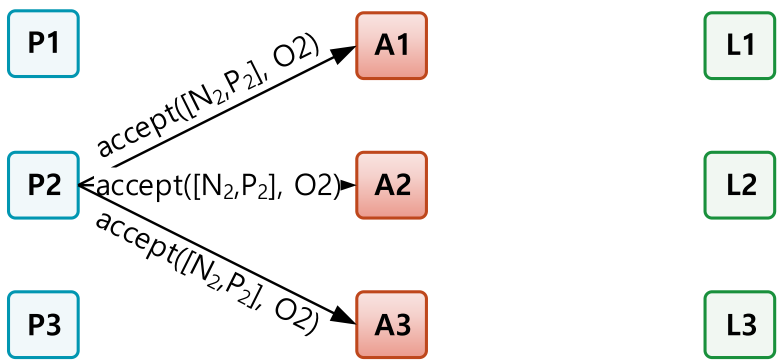
- If a proposer receives corresponding promises from a majority of acceptors:
  - It sends an accept request to each of those acceptors for a proposal numbered $(N_i, P_i)$ with operation P, which is the operation of the highest-numbered proposal among the responses

- Otherwise, aborts and starts again with a new proposal number

# Paxos

- Phase 2a - accept phase



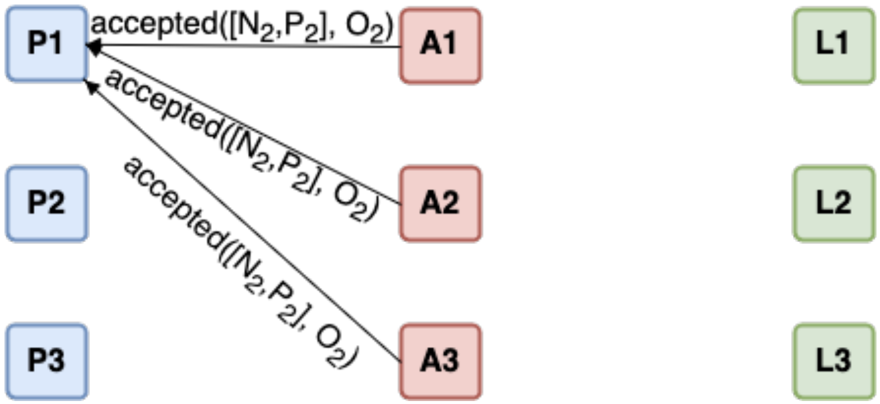



# Paxos

- Phase 2b - accepted phase
- An acceptor has promised to proposal $(N_i, P_i)$
  - If receives an accept request numbered $(N_i, P_i)$ sends accepted message
  - If receives an accept request numbered $(N_j, P_j)$ sends accepted with the accepted number and its operation
- Acceptor accepts the accept request if its proposal no. is the highest proposal no. it have agreed to

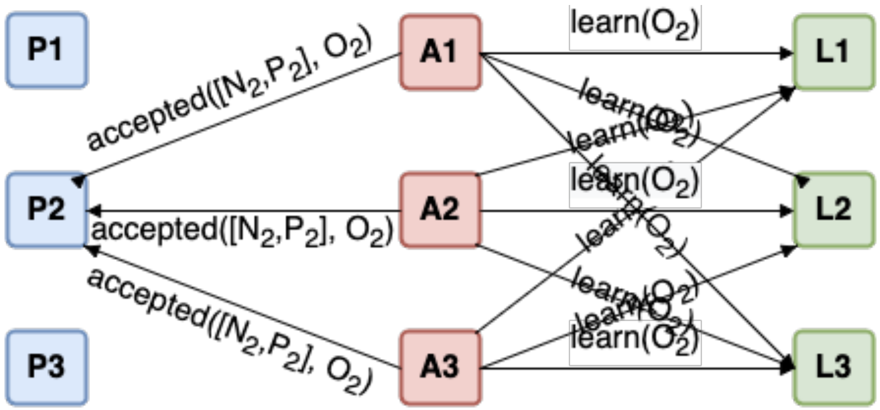
# Paxos

- Phase 2b - accepted phase
  - Acceptors tell $P_1$, they have accepted another proposer
  - $P_1$ may retry



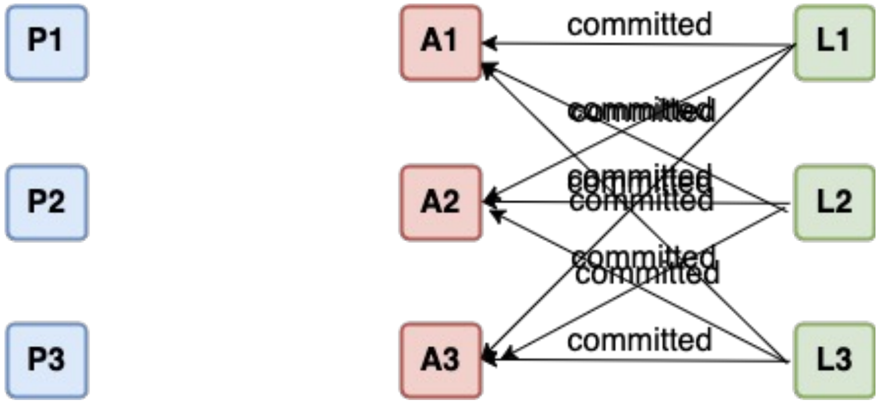
# Paxos

- Phase 2b - accepted phase
  - After sending accepted, acceptors talk with learners about their decision
  - Learners commit if they receive the same operation from majority of acceptors



# Paxos

- Phase 2b - accepted phase
  - Learners acknowledge acceptors about their commit
  - When Acceptors received enough committed, start a new cycle



# Paxos

- Phase 2b - accepted phase

- What if a new proposal received with higher number than what has accepted, before learn?

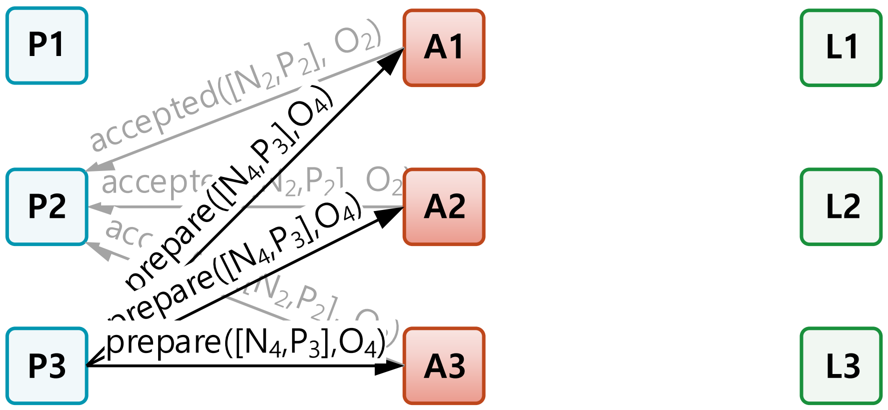
- The proposal is accepted if the operation is the same as any previously accepted proposal!

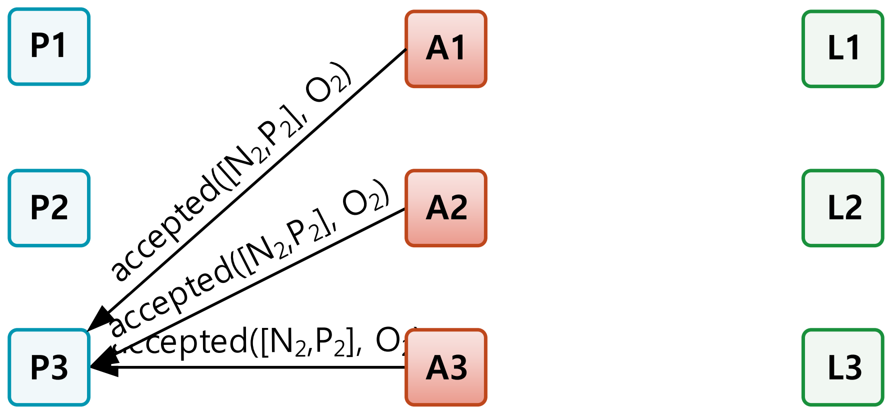
# Paxos

- Once a proposal with operation P is chosen by majority,
  - No new operation will accepted, until the chosen operation get completed
  - If the operation on the highest-numbered proposal has not completed, no new operation can be proposed
  - Every higher-numbered proposal that is chosen also must propose P
- The goal is reaching a consensus, it is not important which value is eventually accepted.

# Paxos

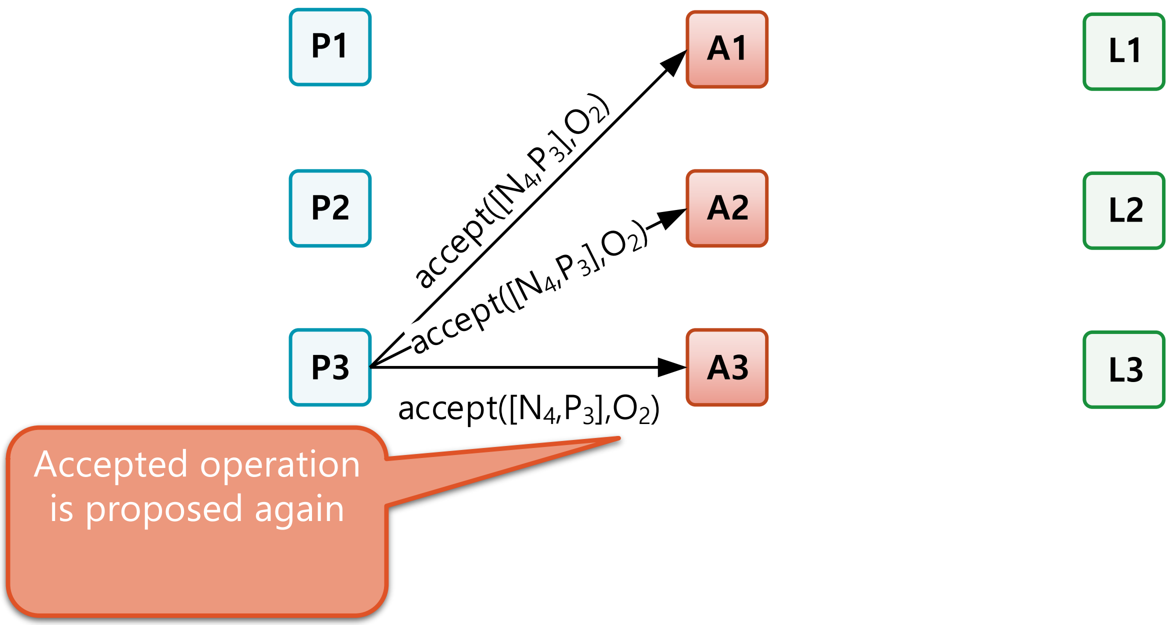
- Phase 2b - accepted phase
  - What if new proposal received by acceptors before completion of learn with higher proposal no.?
  - $N_4 > N_2$



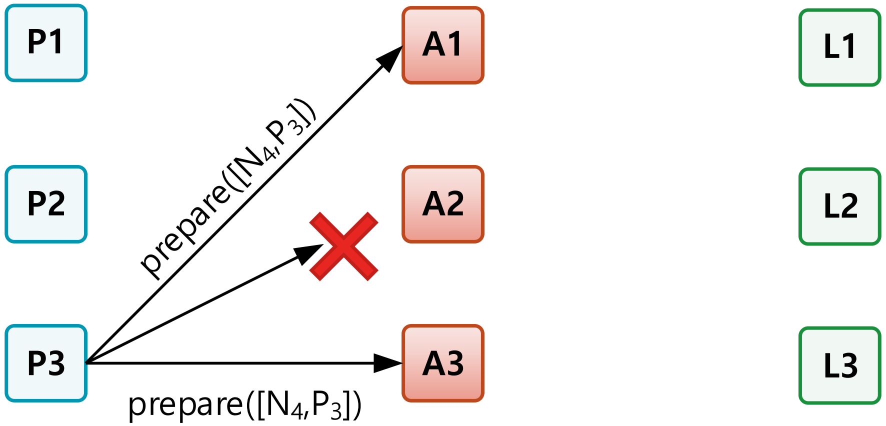
# Paxos

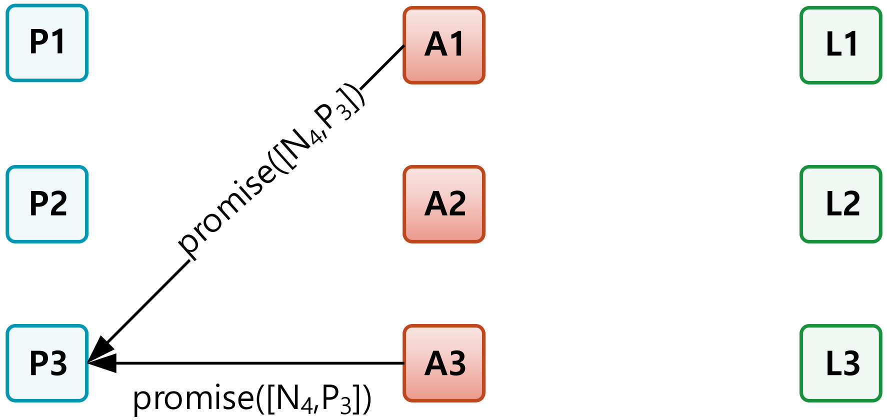
- Phase 2b - accepted phase
  - What if new proposal received by acceptors before completion of learn with higher proposal no.?
  - $N_4 > N_2$



# Paxos

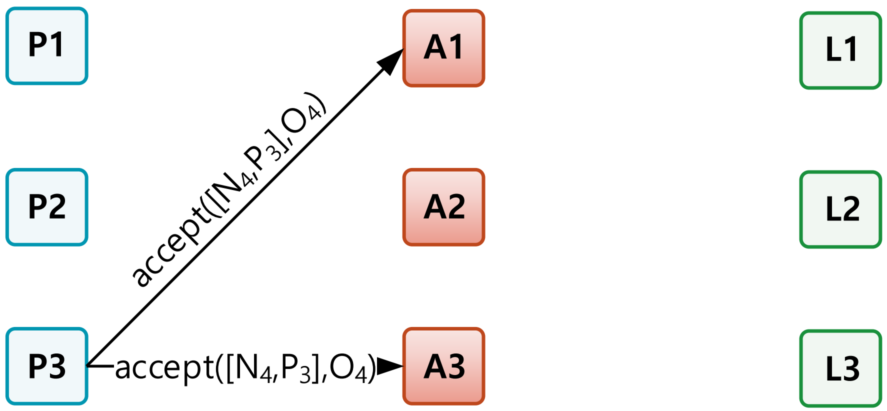
- A faulty scenario (1)

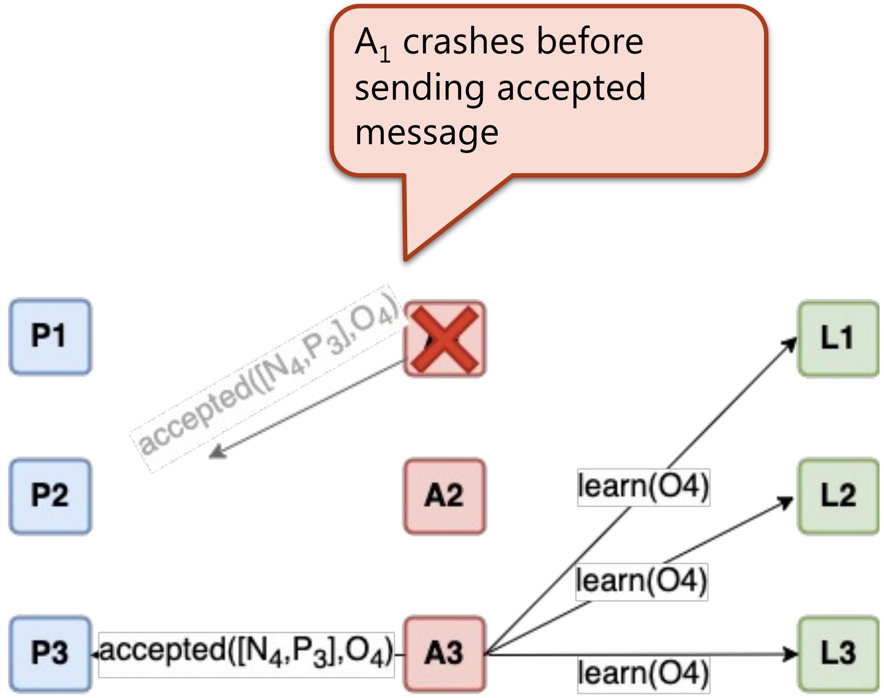


# Paxos

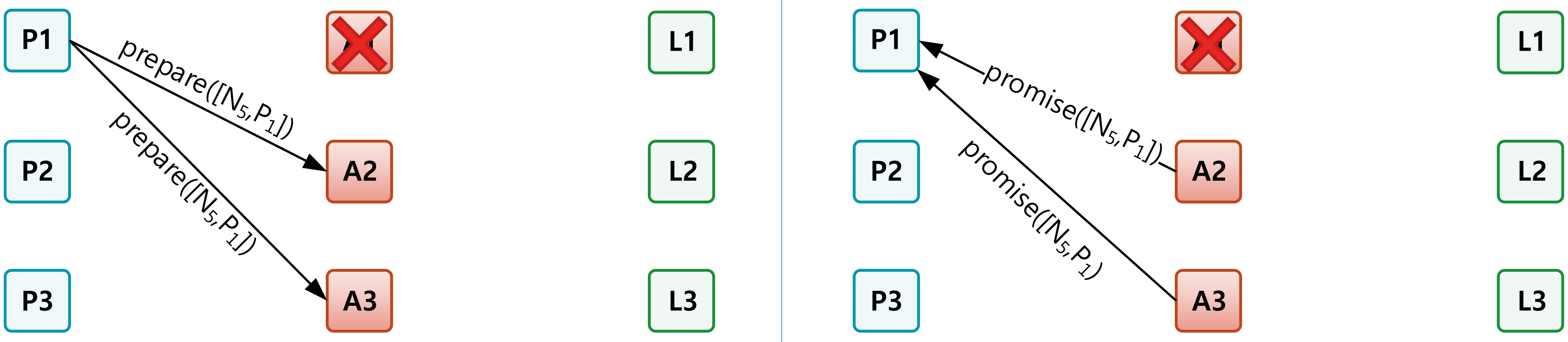
- A faulty scenario (2)



# Paxos

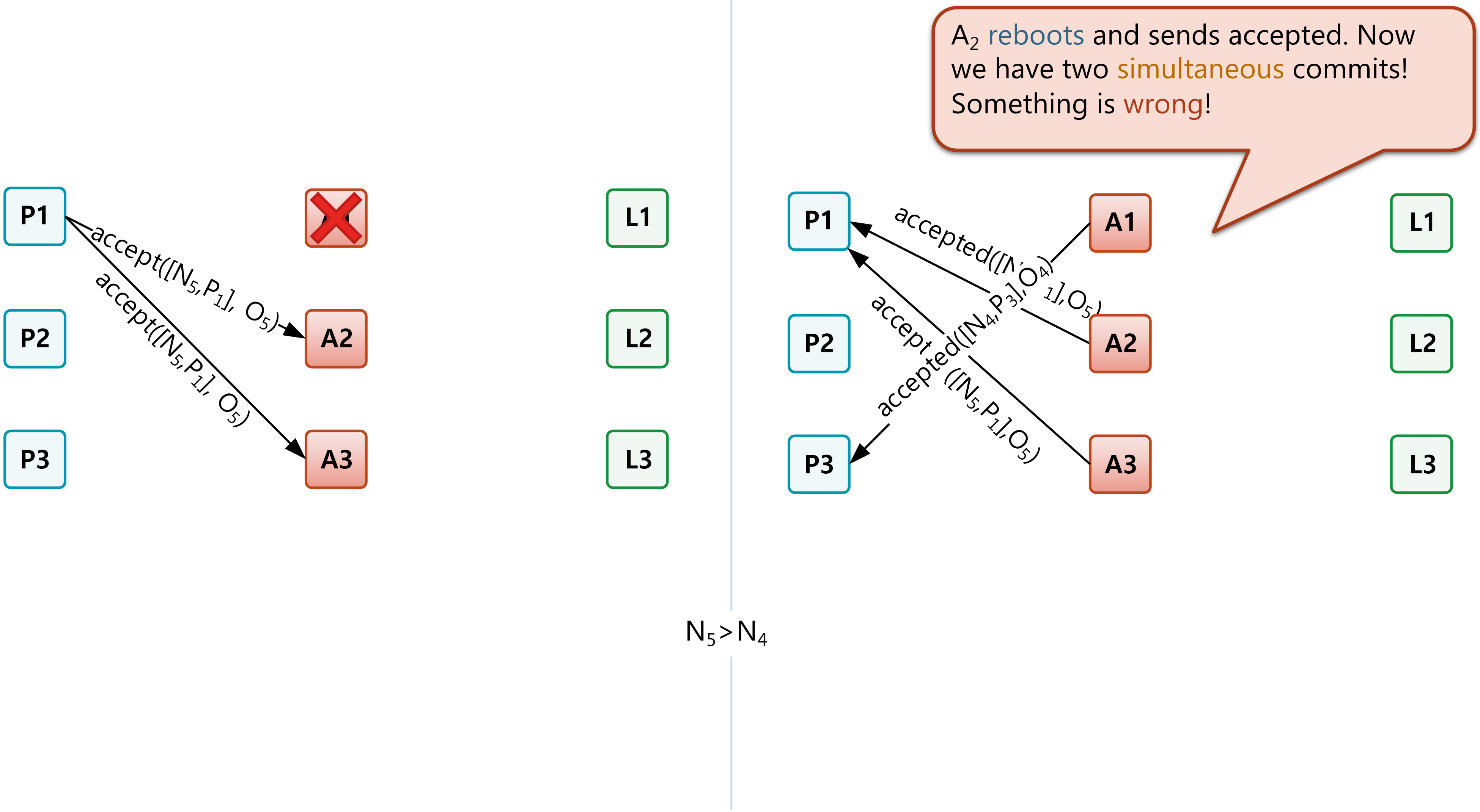
- A faulty scenario (3)




$N_5 > N_4$

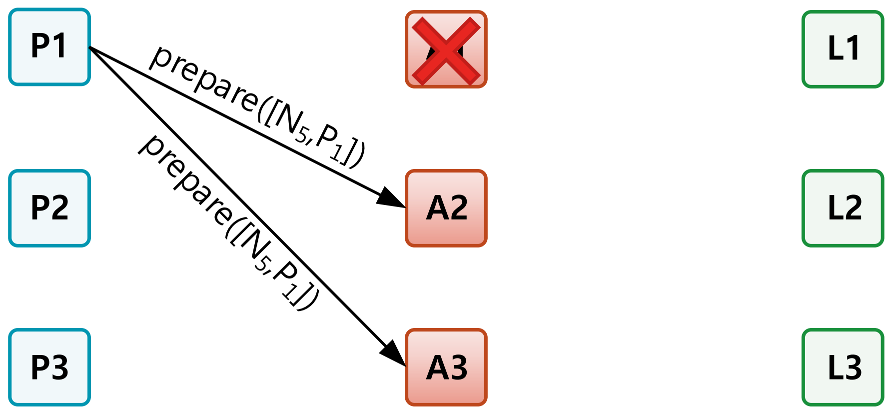
# Paxos

- A faulty scenario (4)

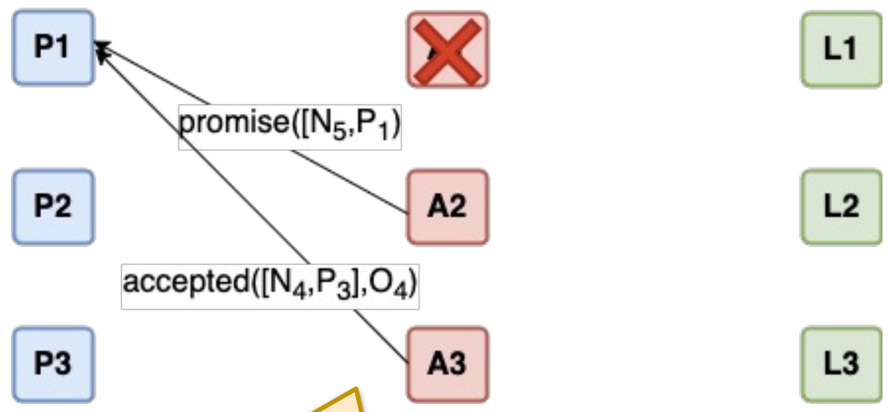


# Paxos

- A faulty scenario (3-2)
  - correct version
- After a while $A_1$ reboots and sends accepted and operation continues




$N_5 > N_4$

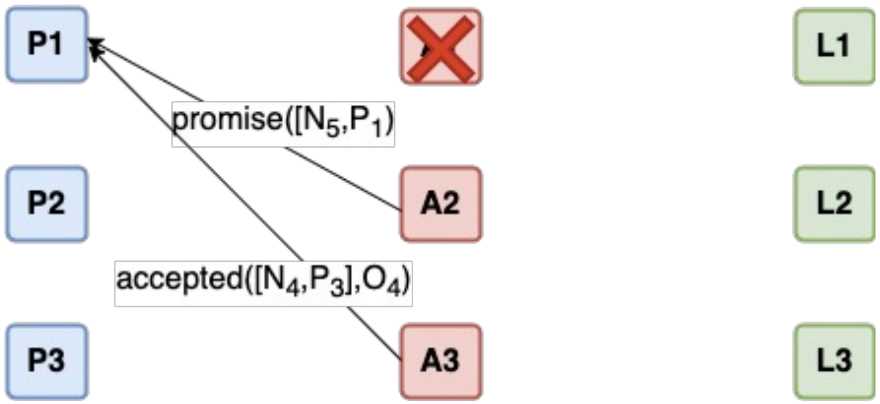



$A_3$ never sends promise if $O_4$ has not been committed.
(step 3 is wrong)

No new operation will accepted, until the chosen operation get completed

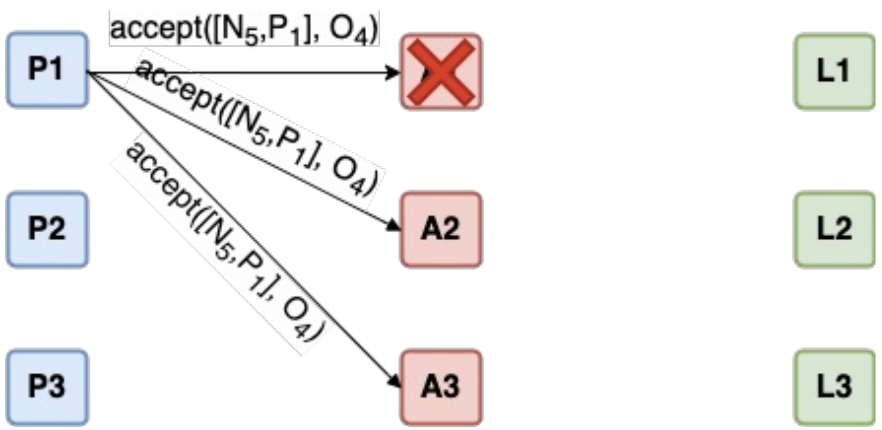
# Paxos

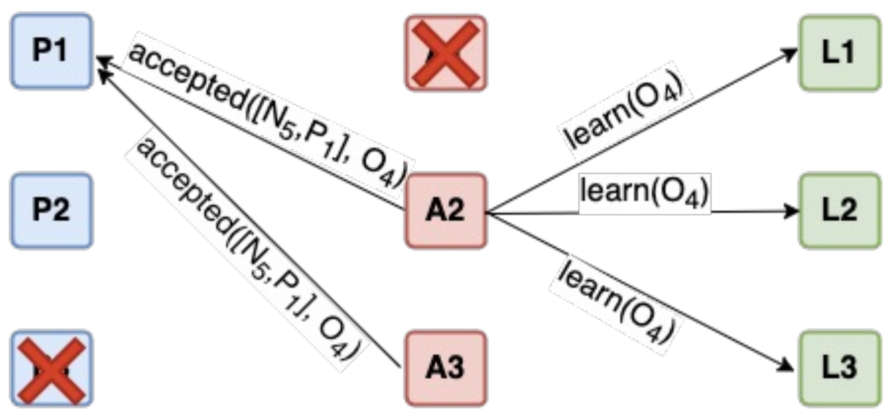
- A faulty scenario
- What if $A_1$ crash and does not reboot again or it doesn't't remember its previous choice?



# Paxos

- A faulty scenario
- P1 retries with the $O_4$ operation!!



# Paxos

Think about other bad conditions:

- What if leader fails
  - Before sending accept
  - After sending accept
  - To send accept to majority of acceptors
  - To send accept to some of acceptors (not majority)

- What if a node fails after receiving accept?
  - If it doesn't restart ···
  - If it reboots ···

- What if a node fails after sending promise?
  - If it reboots ···

# Paxos

- Safety Property
  - Only an operation that has been proposed may be accepted.
  - Only a single operation is chosen
  - An learner learns an operation that has been chosen

# Paxos

- Liveness Property (= Termination)
  - If two or more proposers race to propose new values, they might step on each other toes all the time.
    - P1: prepare(n1 )
    - P2: prepare(n2 )
    - P1: accept(n1 , v1 )
    - P1: prepare(n3 )
    - P2: accept(n2 , v2 )
    - P2: prepare(n4 ) , ···.
    - n1 < n2 < n3 < n4 <···
  - With randomness, this occurs exceedingly rarely.

# Paxos

- To read a client must ask several nodes and choose the value of majority

## Paxos Issues

- Difficult to understand
- *" The dirty little secret of the NSDI* community is that at most five people really, truly understand every part of Paxos ;-)."* – NSDI viewer

- Very difficult to implement
- *"There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system…the final system will be based on an unproven protocol." – Chubby Authors*

# Designing for Understandability: The Raft Consensus Algorithm

## Replicated State Machine

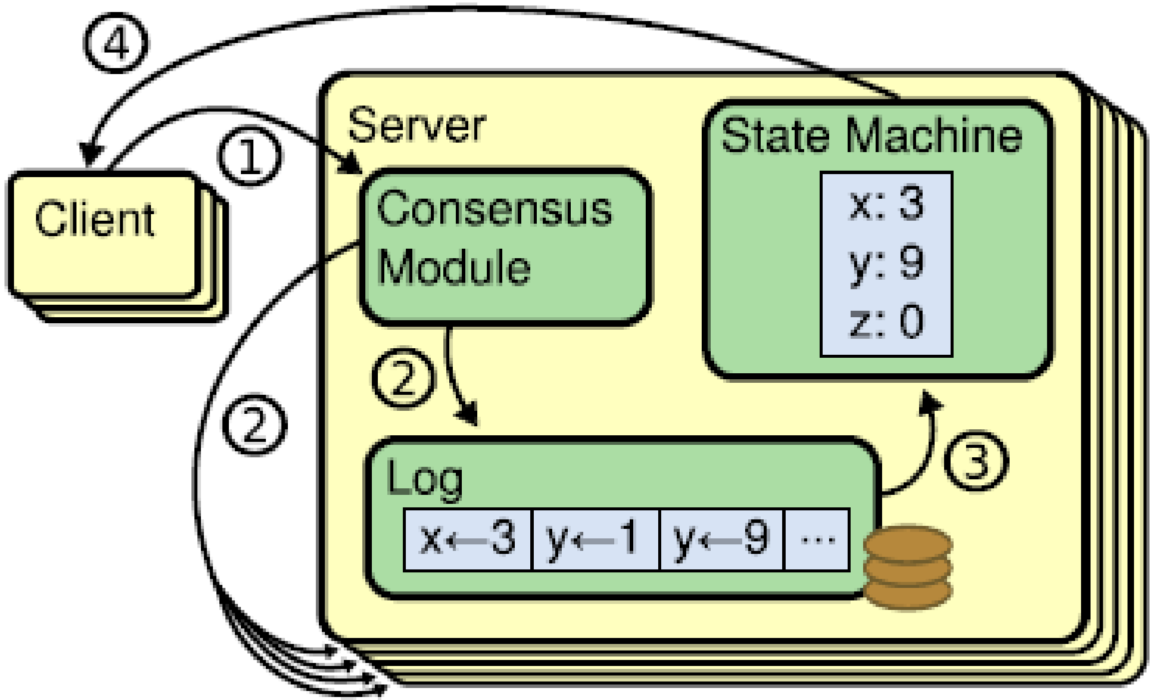
- Each command from a client changes the state of a replica
- Each replica maintains a log of events
- Replicas apply events in the log to update their state

- Log Consensus
- All replicas must agree on the order of events in the log
- Consensus algorithm (i.e. Paxos) ensures that all logs contain the same commands in the same order

- Replicated log => Replicated State Machine

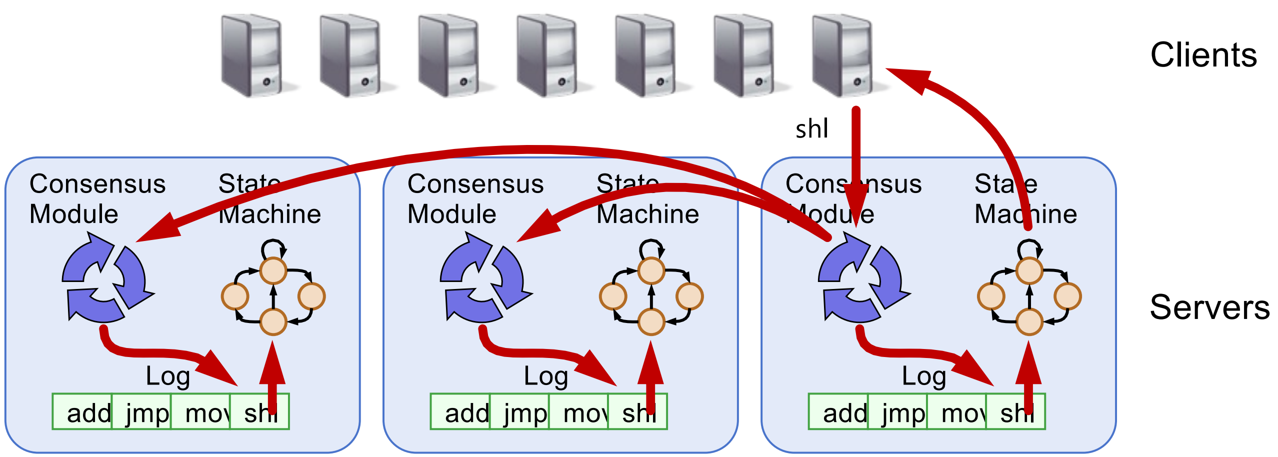
# Distributed Log

- State machines always execute commands in the log order
- They will remain consistent as long as command executions have deterministic results



# Overview

- Client sends a command to one of the servers
- Server adds the command to its log
- Server forwards the new log entry to the other servers
- Once a consensus has been reached, each server state machine process the command and sends it reply to the client



## Leader Election

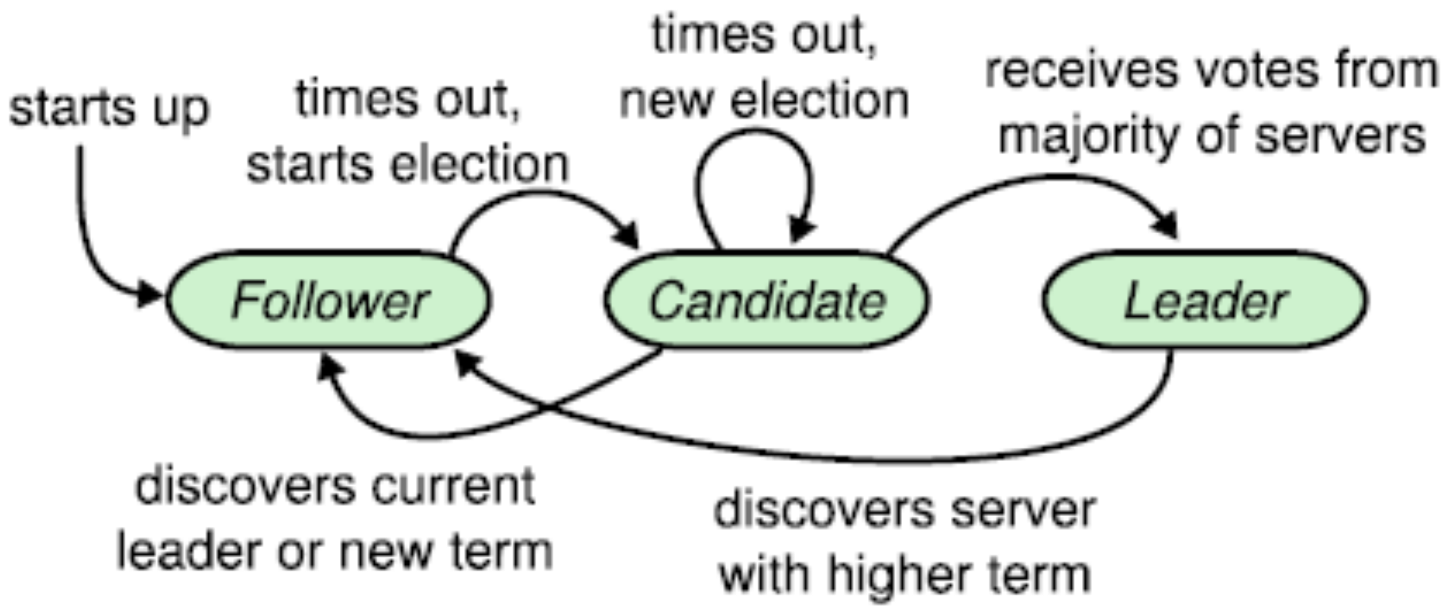
- RAFT assumes starts with electing one leader

- Each server can be in one of three states
  - Leader
  - Follower
  - Candidate (to be the new leader)

- Raft guarantees at a given time only one leader exists

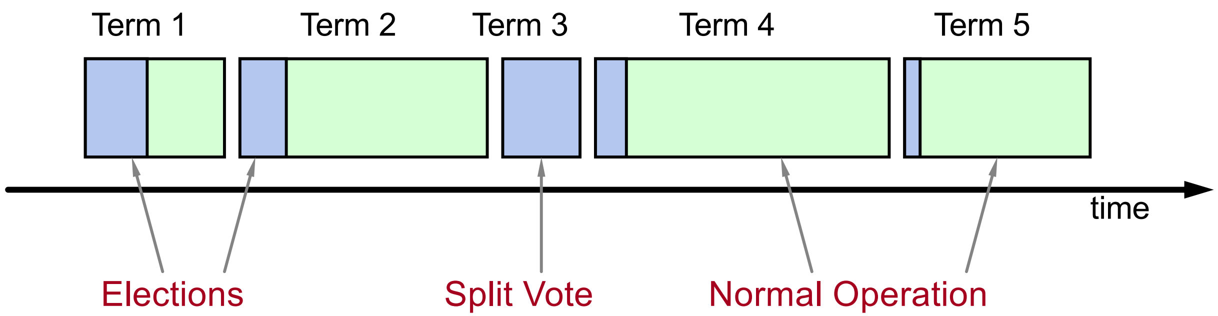
# Leader Election

- Leader transmits heartbeats
- If Election-Timeout elapses followers start the election process



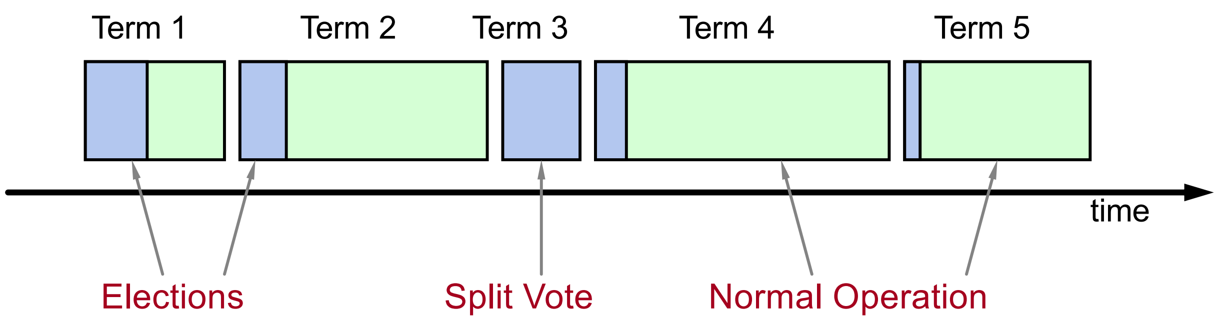
# Leader Election

- Time is divided into Terms
- A term may has
  - no leader → election / split vote
  - one leader → normal operation



# Leader Election

- Term is like logical clock
- Followers
  - maintain current Term, to identify obsolete info
  - include in all messages
  - update the term if receive a higher value



# Leader Election Process

- When a follower starts an election, it
  - Increments its current term
  - Transitions to candidate state
  - Votes for itself
  - Issues RequestVote RPCs to all the other servers in the cluster.

- A candidate remains in that state until
  - It wins the election
  - Another server becomes the new leader
  - A period of time goes by with no winner, backs off with random interval

- Candidate receive the majority of votes become leader
- Each server will vote for at most one candidate in one term
- Winner sends heartbeat messages to all others

# Log Replication

- Leaders
  - Accept client commands
  - Append them to their log (new entry)
  - Issue AppendEntry RPCs in parallel to all followers

- Followers record the log and acknowledge the leader

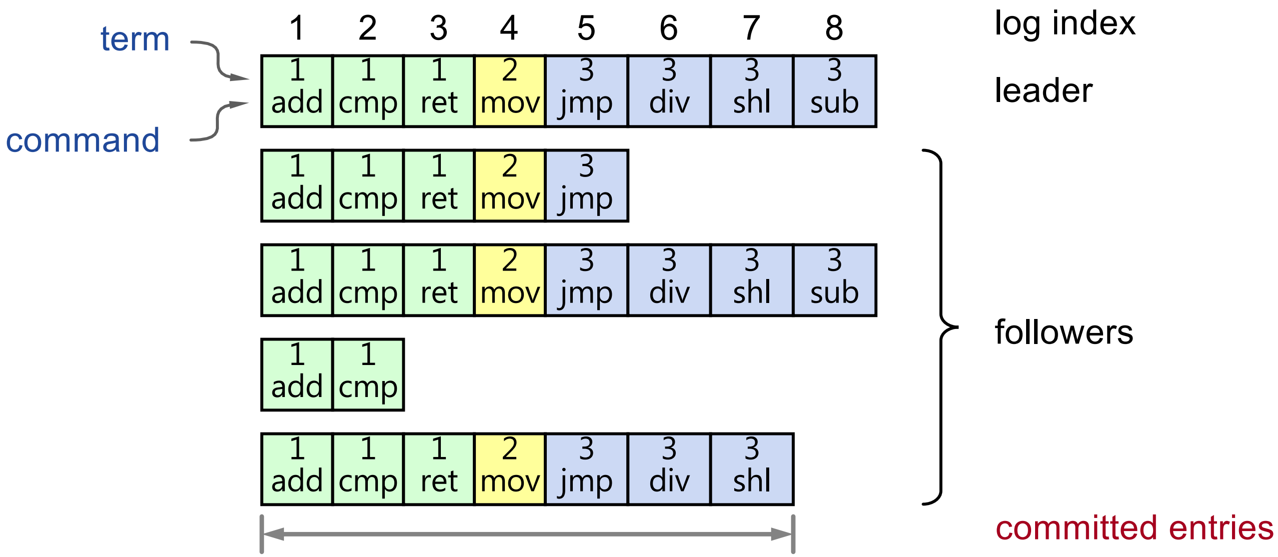
- Leader commits (updates the state machine) if majority acknowledged
  - Re-issue the command for slow servers, no problem!

- Heartbeats and subsequent messages include the index of last committed log
- Committing an entry also commits all previous entries

# Log Structure

- Log entry = { index, term, command }
- Log stored on stable storage (disk); survives crashes

- Entry committed if known to be stored on majority of servers
  - Durable & stable, will eventually be executed by state machines



# Log Structure

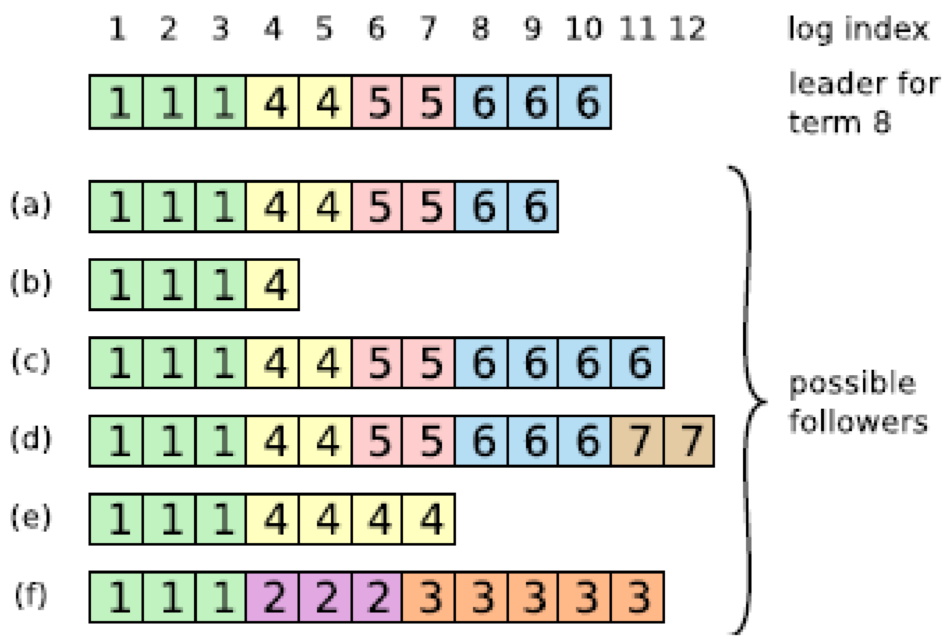
- Raft commits entries in strictly sequential order
  - No gap is accepted

- If log entries on different server have same index and term:
  - Store the same command
  - Logs are identical in all preceding entries

- Entry committed if known to be stored on majority of servers
  - Durable & stable, will eventually be executed by state machines

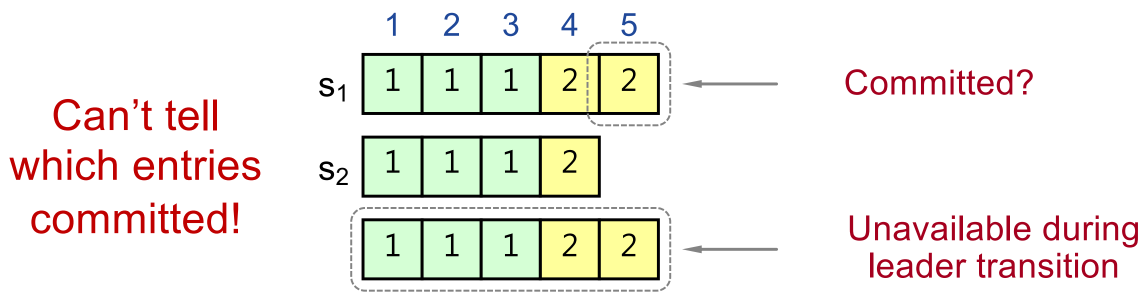
# Handling Leader Crash

- Can leave the cluster in a inconsistent state if the old leader had not fully replicated a previous entry
- Some followers may have in their logs entries that the new leader does not have
- Other followers may miss entries that the new leader has



## Handling Leader Crash

- Elect candidate most likely to contain all committed entries



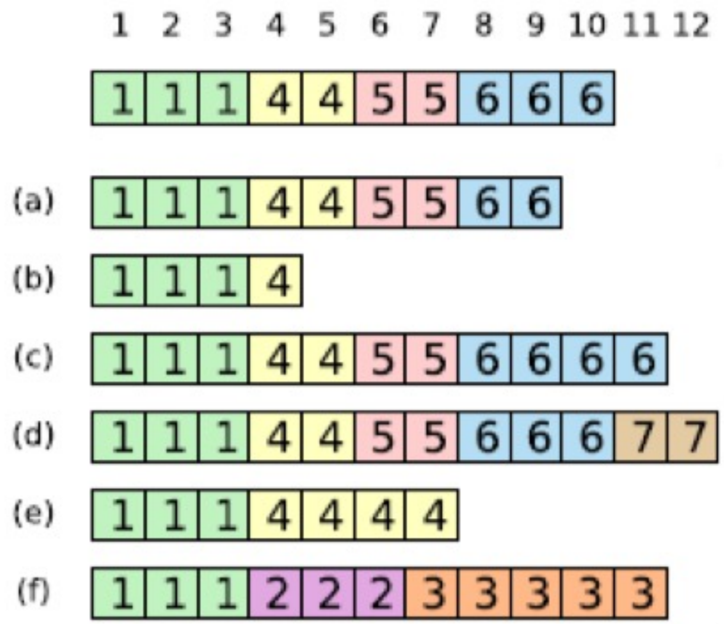

- In RequestVote, candidates include {index,term} of last log entry
- Vote for candidate unless
  - Their own log is more "up to date" (higher term-longer log)
  - They have already voted for another server

- Leader will have "most complete" log among electing majority

# Handling Leader Crash

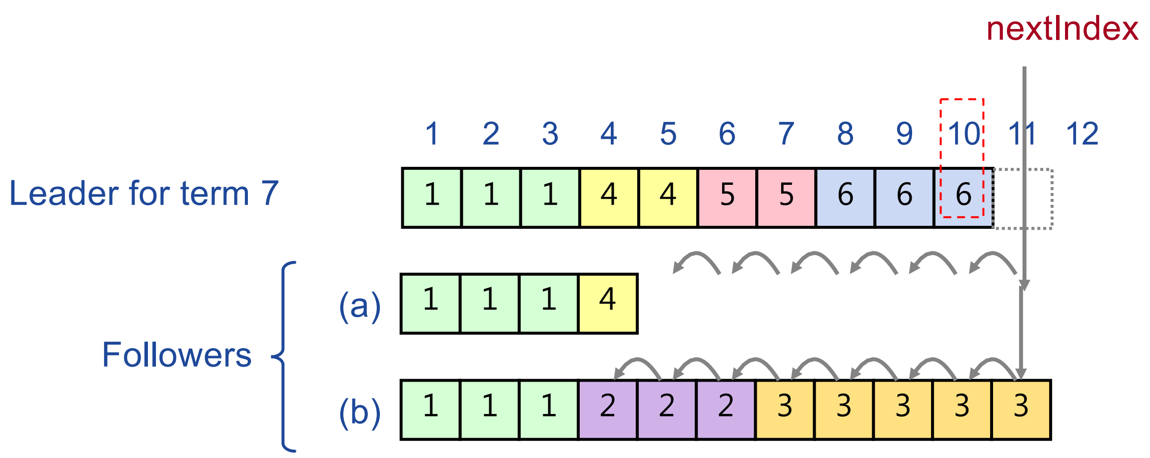
- New leader forces followers' log to duplicate its own
- Conflicting entries in followers' logs will be overwritten

- New leader sets its nextIndex to the index just after its last log entry (11 in the example)
- Broadcasts it to all its followers



# Handling Leader Crash

- Leader maintains a nextIndex for each follower
  - Index of entry it will send to that follower
- Followers that have missed some AppendEntry calls will refuse all further AppendEntry calls
- Leader will decrement its nextIndex for that follower and redo the previous AppendEntry call



## References


- Slides of Dr. Payberah: https://www.slideshare.net/payberah/paxos-43900572
- Neat Algorithms - Paxos: http://harry.me/blog/2014/12/27/neat-algorithms-paxos/
- Kirsch, Jonathan, and Yair Amir. "Paxos for system builders: An overview." Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. ACM, 2008.
- Lamport, Leslie. "Paxos made simple." ACM Sigact News 32.4 (2001): 18-25.
- Princeton Distributed Systems course, https://www.cs.princeton.edu/courses/archive/spring22/cos418/schedule.html