THE UNIVERSITY OF CHICAGO

Class 6: Fault Tolerance/Consensus

MPCS: 52040 Distributed Systems

Kyle Chard

chard@uchicago.edu

Course schedule (* things might change a little)

Date	Lecture discussion topic	Class	Assessment
January 10	1. Introduction to Distributed Systems	Intro/Logistics	
January 17	2 Distributed architectures 3 Processes and virtualization	1 Docker	Homework 1 due
January 24	4 Networks and Communication	2 RPC/ZMQ/MPI	Homework 2 due
January 31	5 Naming	3 DNS/LDAP	Homework 3 due
February 7	6 Coordination and Synchronization		**Mid term exam**
February 14	7 Fault tolerance and consensus	Raft Project description	Homework 4 due (Project released)
February 21	8 Consistency and replication	4 FaaS	
February 28	9 Distributed data	5 Distributed data	
March 7	10 Data-intensive computing	6 Streaming	**Project due (March 9)** **Final exam (March 14)**

Agenda

- Part 1: Clock Synchronization
- Part 2: Logical clocks
- Part 3: Mutual exclusion
- Part 4: Election algorithms

Distributed Systems: Chapter 6

Election algorithms

- Many distributed algorithms require one process to act as a coordinator, initiator, or another role
  - E.g., mutual exclusion algorithms, time servers, consensus

- It doesn’t matter which process, but we need to guarantee one will do it

Election algorithms

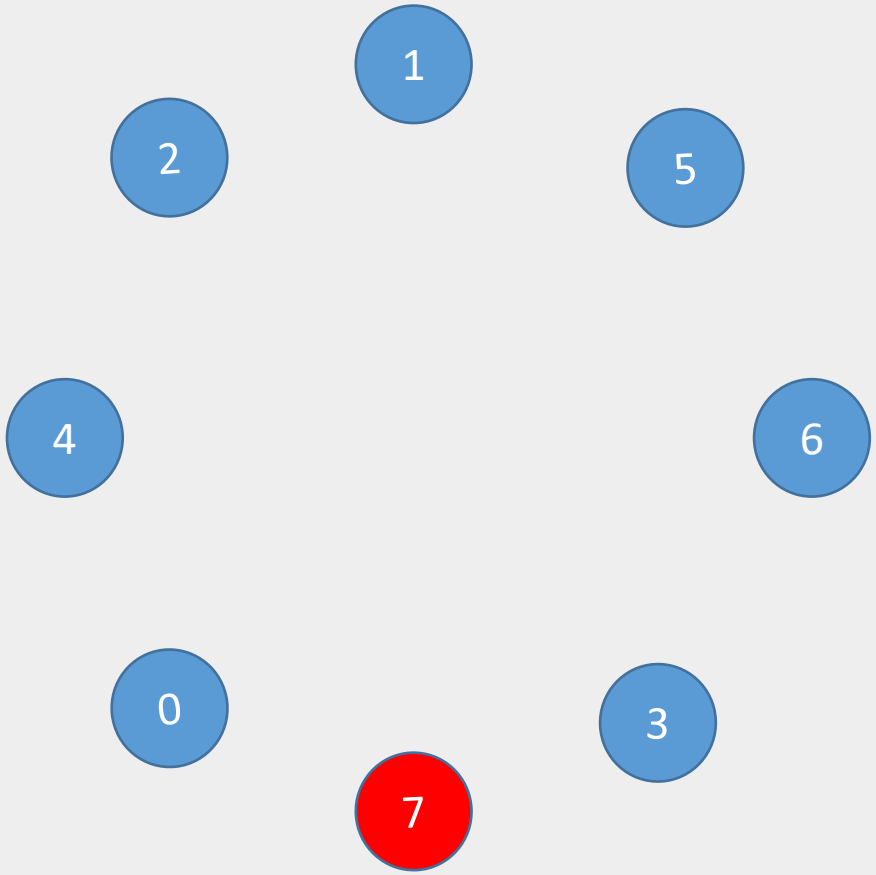
- Assumptions
  - All processes have a unique ID
  - All processes know the IDs of all other processes in the system
  - Processes do not know if another process is up or down
  - Our aim is to elect one of these processes
    - (often simplified as choosing the highest ID that is up)

- Bully: send election messages to all processes with higher identifiers, if no one replies, node wins election and becomes coordinator
- Ring: send election messages in a ring, after message goes around the entire ring, the one with the highest ID becomes coordinator

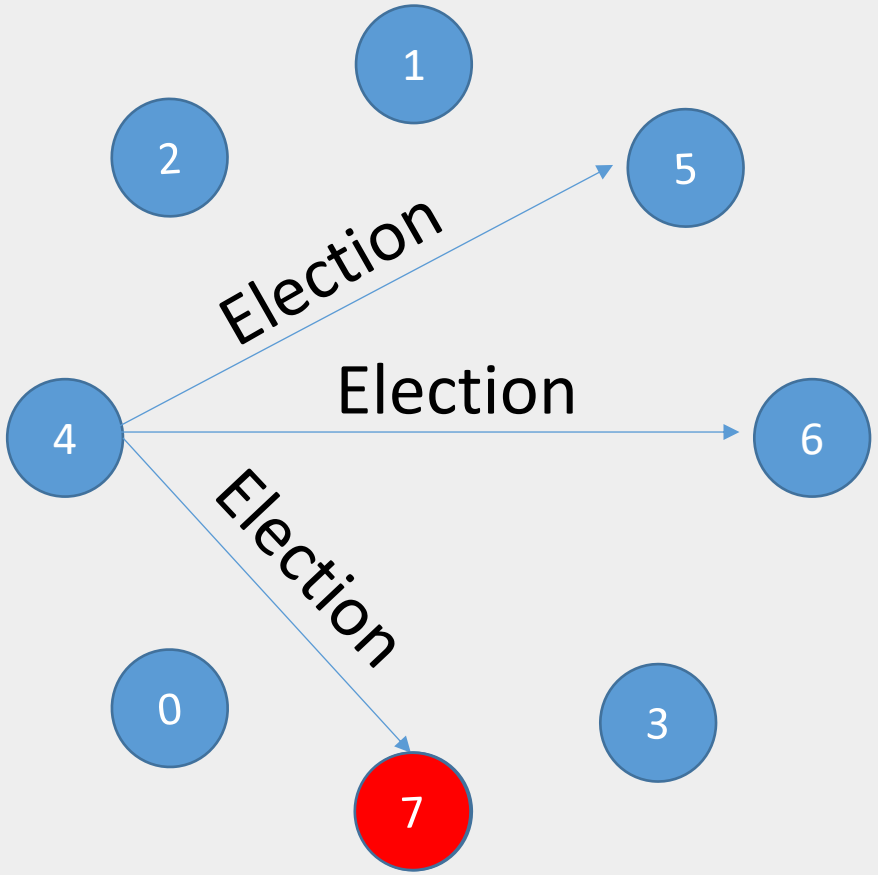
Bully algorithm

- Consider N processes {P0, … Pn-1} where id(Pk) = k
- When a process Pk notices that the coordinator is no longer responding to requests, it initiates an election:
  1. Pk sends an ELECTION message to all processes with higher identifiers:
     - Pk+1, Pk+2, …, Pn-1
  2. If no one responds, Pk wins the election and becomes coordinator
  3. If a process with higher ID answers, it takes over and Pk's job is done
- Winning process sends COORDINATOR message to all other processes
- If a process that was down comes back up, it holds an election
  - If it's the highest process it will win the election and take over as coordinator

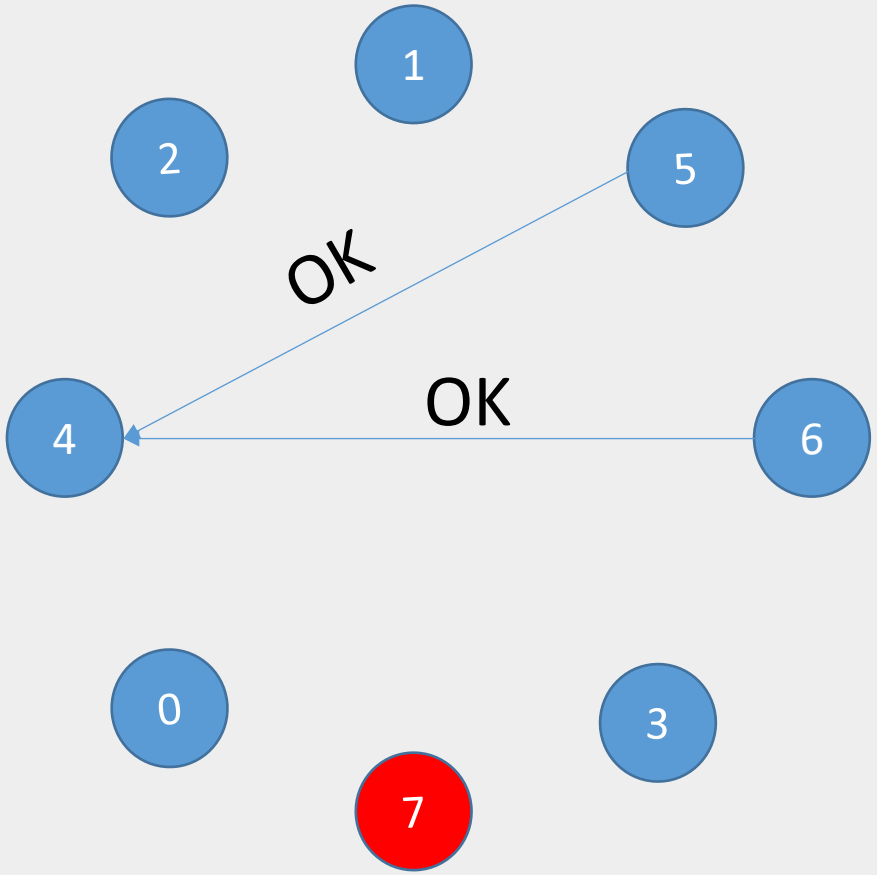
Bully algorithm



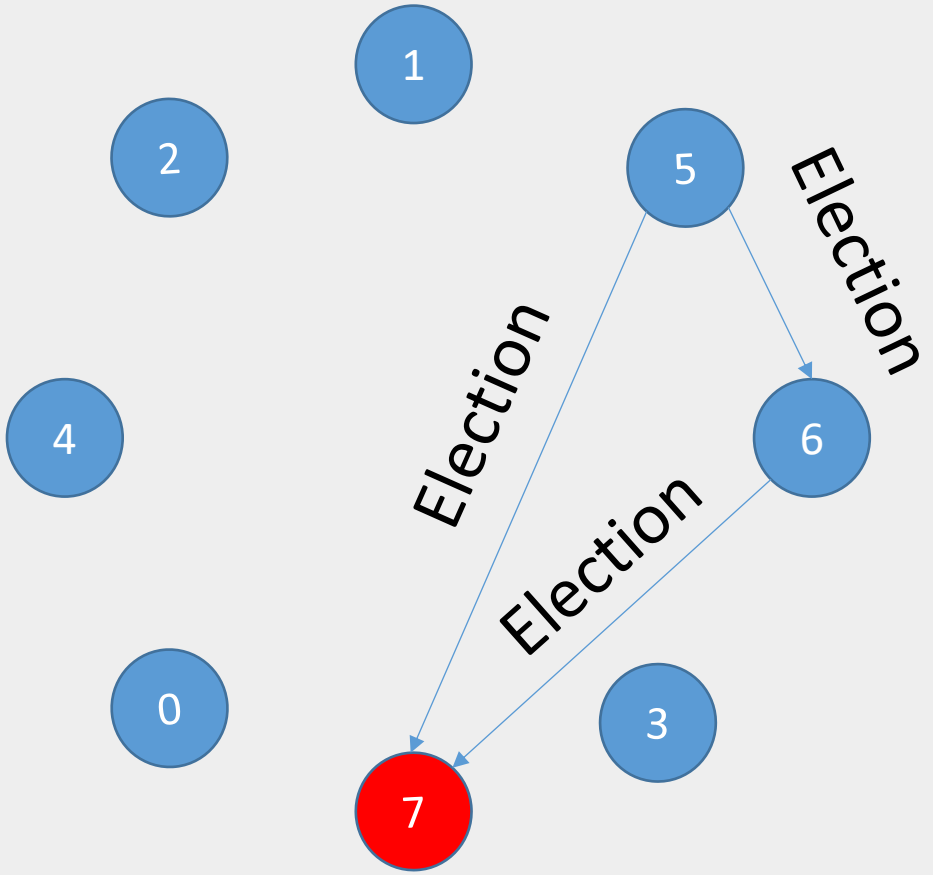
Bully algorithm



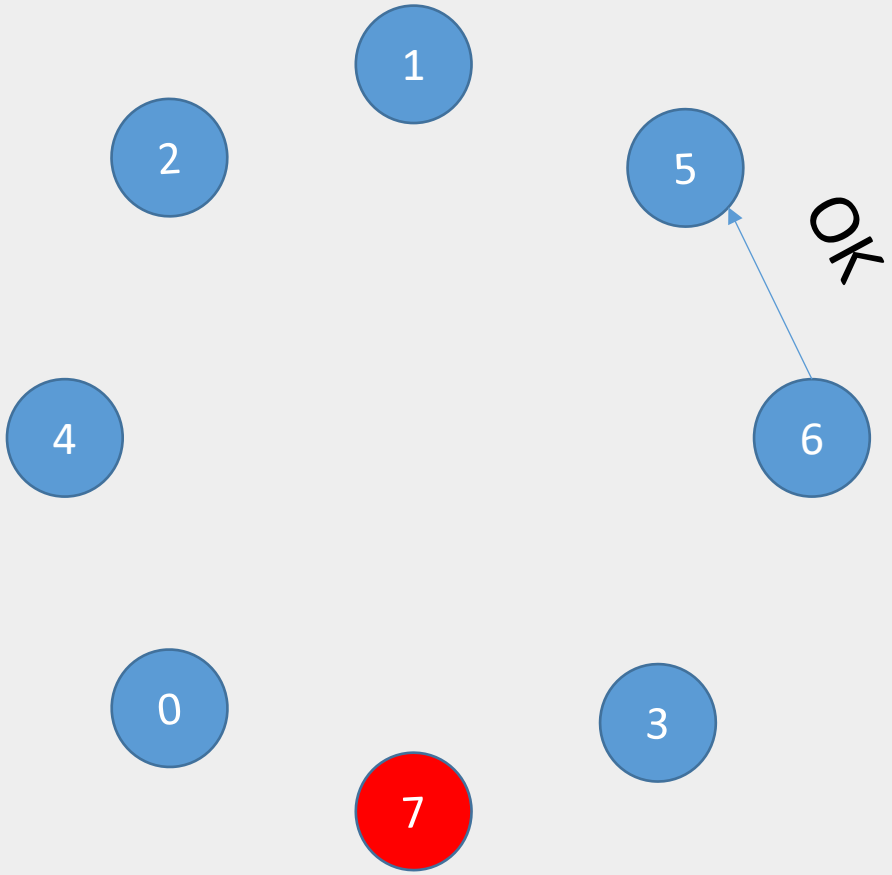
Bully algorithm



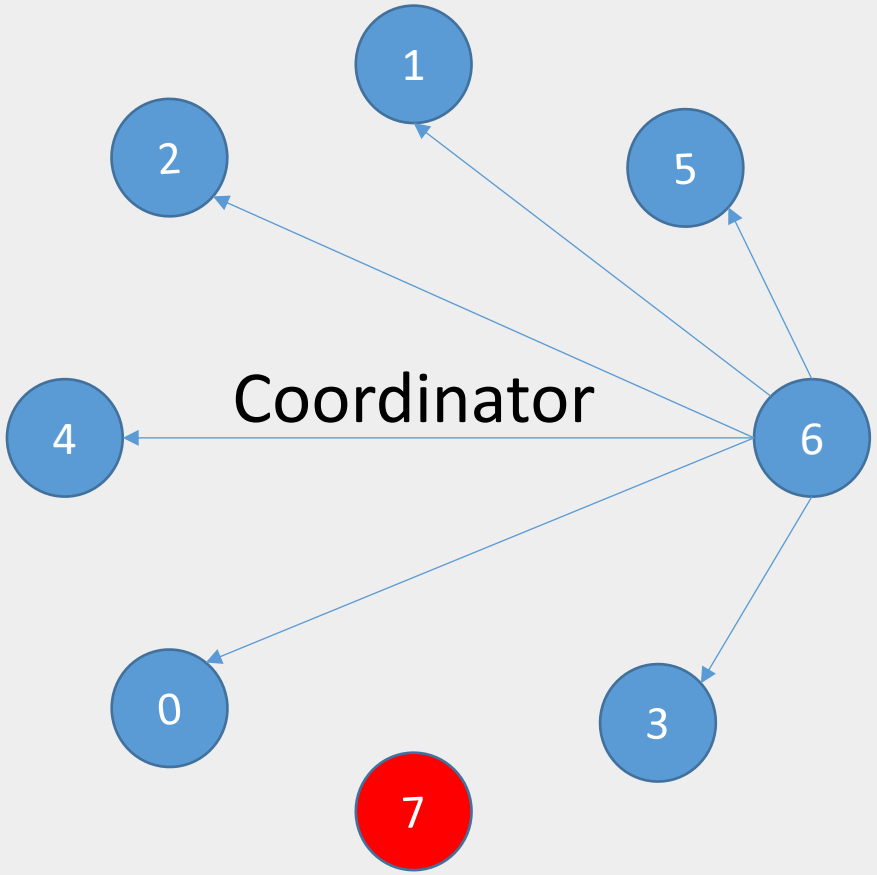
Bully algorithm



Bully algorithm



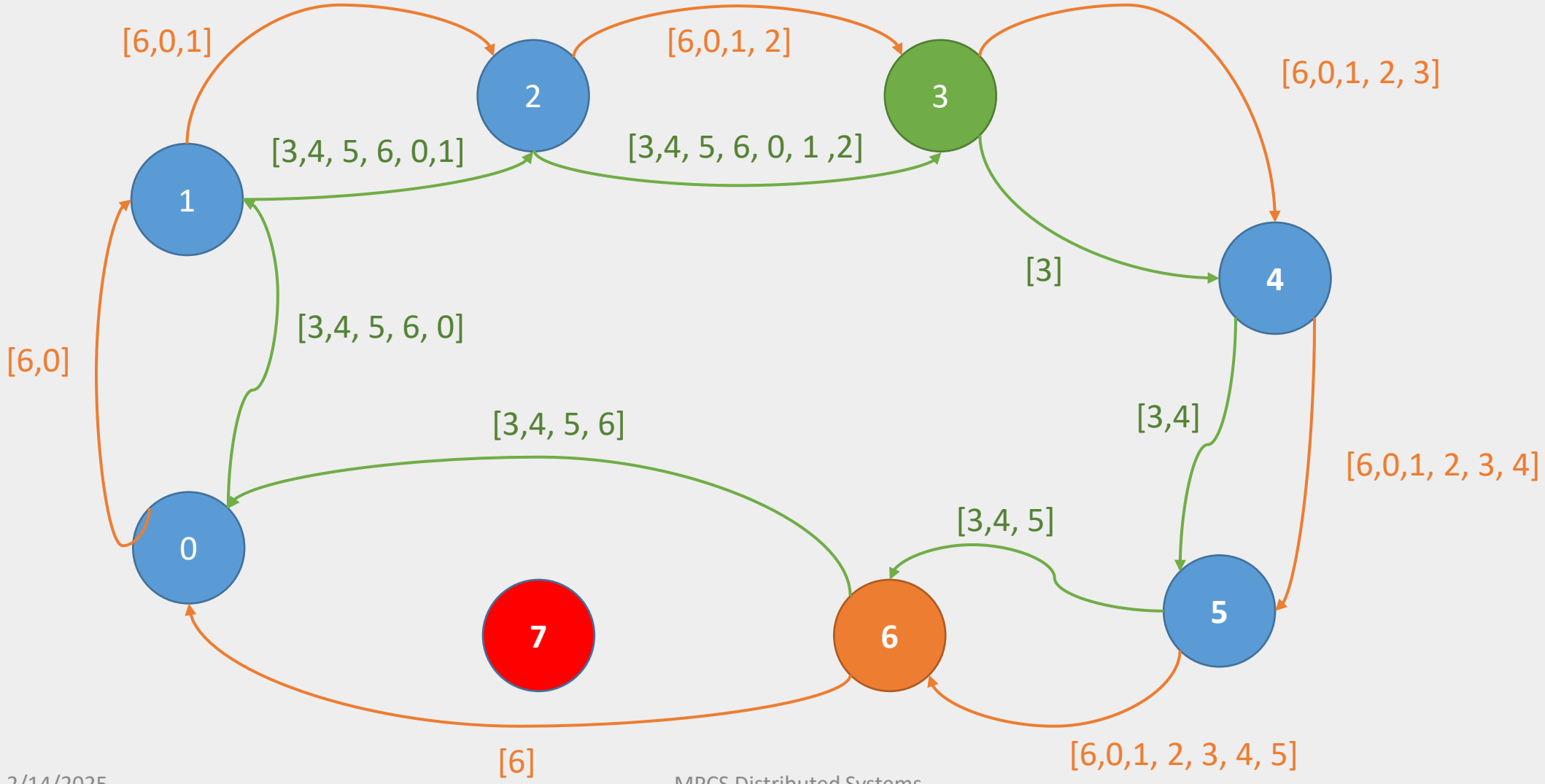
Bully algorithm



Ring algorithm

- Process priority is obtained by organizing processes into a (logical) ring and the process with the highest priority should be elected as coordinator.
  - Any process can start an election by sending an election message to its successor
    - If a successor is down, the message is passed on to the next successor
  - If a message is passed on, the sender adds itself to the list
    - When it gets back to the initiator, everyone had a chance to make its presence known
  - The initiator sends a coordinator message around the ring containing a list of all living processes
    - The one with the highest priority is elected as coordinator

Example of ring election



Agenda

- Part 1: Failures and Resilience
- Part 2: Distributed Consensus
- Part 3: Paxos
- Part 4: Raft

Background: Faults

- Faults are inevitable in any systems
  - Node failure
  - Network failure
  - Software failure
  - Power failure
  - Data center failure
  - Client failure
  - …

Failures are even created intentionally

- Netflix developed chaos monkey to intentionally cause failures to increase application resiliency
- Chaos gorilla will take down an entire data center

https://netflix.github.io/chaosmonkey

Faults in distributed systems

- Unlike centralized systems, distributed systems may often have partial failures:
  - Part of the system fails while the remaining part continues to operate
- Ideally we want to automatically discover/recover from partial failures without affecting performance

Dependable systems

- Availability
  - System is ready to be used immediately. The probability that a system is operating correctly at any given moment.
  - Often expressed in terms of X 9s (e.g., 4 9s == 99.99%)
- Reliability
  - Run continuously without failure. Defined in terms of a time interval
  - What is the difference with availability?
- Safety
  - Ensure no catastrophic event happens if the system temporarily fails
  - E.g., important for planes and power plants
- Maintainability
  - Ease by which a failed system can be repaired

Anatomy of failure

- A system fails when it cannot meet its promises
- An error is part of a system's state that may lead to failure
- The cause of an error is a fault
  - Transient faults occur once and disappear
  - Intermittent faults occur, then vanish, then reappear
  - Permanent fault continues to exist until the faulty component is replaced
- Fault tolerance means that a system can provide services in the presence of faults

Failures in more detail

- Crash failure: Halts, but is working correctly until it halts
- Omission failure: Fails to respond to incoming requests
  - Receive omission: Fails to receive incoming messages
  - Send omission: Fails to send messages
- Timing failure: Response lies outside a specified time interval
- Response failure: Response is incorrect
  - Value failure: The value of the response is wrong
  - State-transition failure: Deviates from the correct flow of control
- Byzantine/Arbitrary failure: May produce arbitrary responses at arbitrary times

Faults in synchronous and asynchronous systems

- Synchronous system: process execution speeds and message delivery times are bounded. This also means that when server shows no more activity when it is expected to do so, a client can conclude that it has crashed
- Asynchronous system: no assumptions about process execution speeds or message delivery times are made. When a client no longer sees any actions from a server it cannot conclude that the server has crashed → it may be slow or its messages may have been lost

Partially synchronous systems

- Pure synchronous systems exist only in theory
- Stating that every distributed system is asynchronous is not what we see in practice (would be overly pessimistic in designing systems under this assumption)
- Realistic to assume a system is **partially synchronous**: most of the time it behaves as a synchronous system, yet there is no bound on the time that it behaves in an asynchronous fashion.
- In other words, **asynchronous behavior is an exception, meaning that we can normally use timeouts to conclude that a process has indeed crashed, but that occasionally such a conclusion is false**.
- In practice, this means that we will have to design fault-tolerant solutions that can withstand incorrectly detecting that a process halted.

Failures in partially synchronous systems

- **Fail-stop failures** refer to crash failures that can be reliably detected. This may occur when assuming nonfaulty communication links and when the failure-detecting process P can place a worst-case delay on responses from Q
- **Fail-noisy failures** are like fail-stop failures, except that P will only eventually come to the correct conclusion that Q has crashed. This means that there may be some a priori unknown time in which P's detections of the behavior of Q are unreliable
- **Fail-silent failures**, we assume that communication links are nonfaulty, but that process P cannot distinguish crash failures from omission failures
- **Fail-safe failures** cover the case of dealing with arbitrary failures by process Q, yet these failures are benign: they cannot do any harm
- **Fail-arbitrary failures**, Q may fail in any possible way; failures may be unobservable in addition to being harmful to the otherwise correct behavior of other processes

Detecting failures

- How can we reliably detect that a process has actually crashed?
  - Either actively send “are you alive?” or passively wait for messages
- General model
  - Each process is equipped with a failure detection module
  - A process *P* probes another process *Q* for a reaction
  - If *Q* reacts: *Q* is considered to be alive (by *P*)
  - If *Q* does not react with t time units: *Q* is suspected to have crashed
- For a synchronous system a suspected crash is equivalent to a known crash

Masking failures

- The best we can do is try and hide failures from other processes

- Information redundancy
  - Add extra bits to recover from garbled bits (e.g., Hamming code to recover from noise in transmission line)
- Time redundancy
  - Action performed, and if needed it is performed again (e.g., transactions)
- Physical redundancy
  - Extra equipment or processes added to tolerate loss of component
  - Could be done at the hardware or software level

Resilience

- What we would like to do is:
  - A) detect failures
  - B) mask failures
  - C) recover from failures

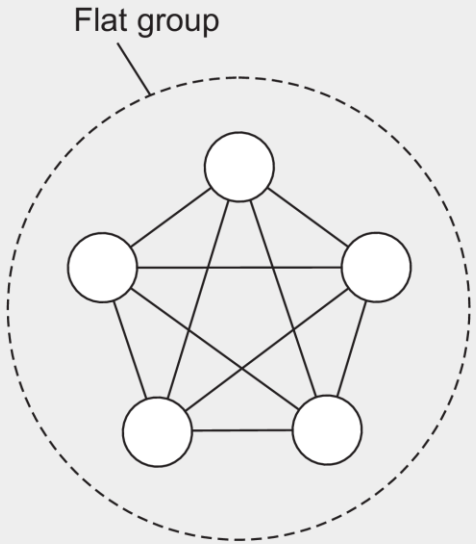
- Resilience is the ability to provide an acceptable level of service in the presence of failures

Resilience by process groups

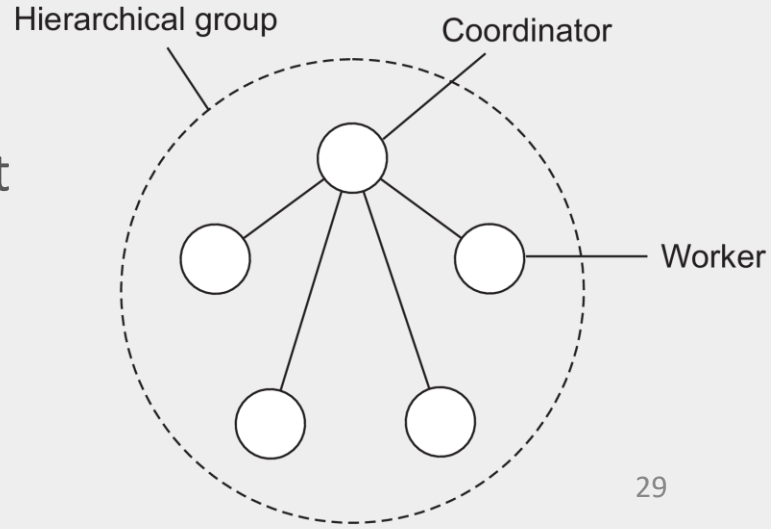
- Key approach is to organize several identical processes into a group
  - When a message is sent to the group, all members receive it
  - If one fails, another can take over
- Groups may be dynamic, processes can join/leave, processes may be members of different groups at the same time
- We need mechanisms to organize and manage process groups

Group organization

- Flat group:
  - All processes are equal and decisions are made collectively
  - Symmetrical/no single point of failure
  - Decision making is complicated and decisions can take time



- Hierarchical group:
  - One process is the coordinator and all others are workers
  - Loss of coordinator means we need to find a replacement
  - Decisions can be made quickly



How does this help us

- Groups allow us to mask faulty processes in the group
  - We replicate processes and organize into a group

- Need to consider how much replication is needed
  - A system is said to be **k-fault tolerant** if it can survive faults in **k** components
  - With silent failures then having **k+1** processes is sufficient
  - With byzantine failures we need at least 2k+1 **non-faulty processes**

Summary

- Failures should be expected: hardware, network, software, power..
- Distributed systems may often have partial failures
- Resilience means that we are able to detect, mask, and recover from partial failures without affecting performance
  - Process groups provide redundancy

Part 2: Distributed Consensus

Consensus

The process by which *nodes* in a distributed system reach agreement on something (value, lock, operation, etc.)

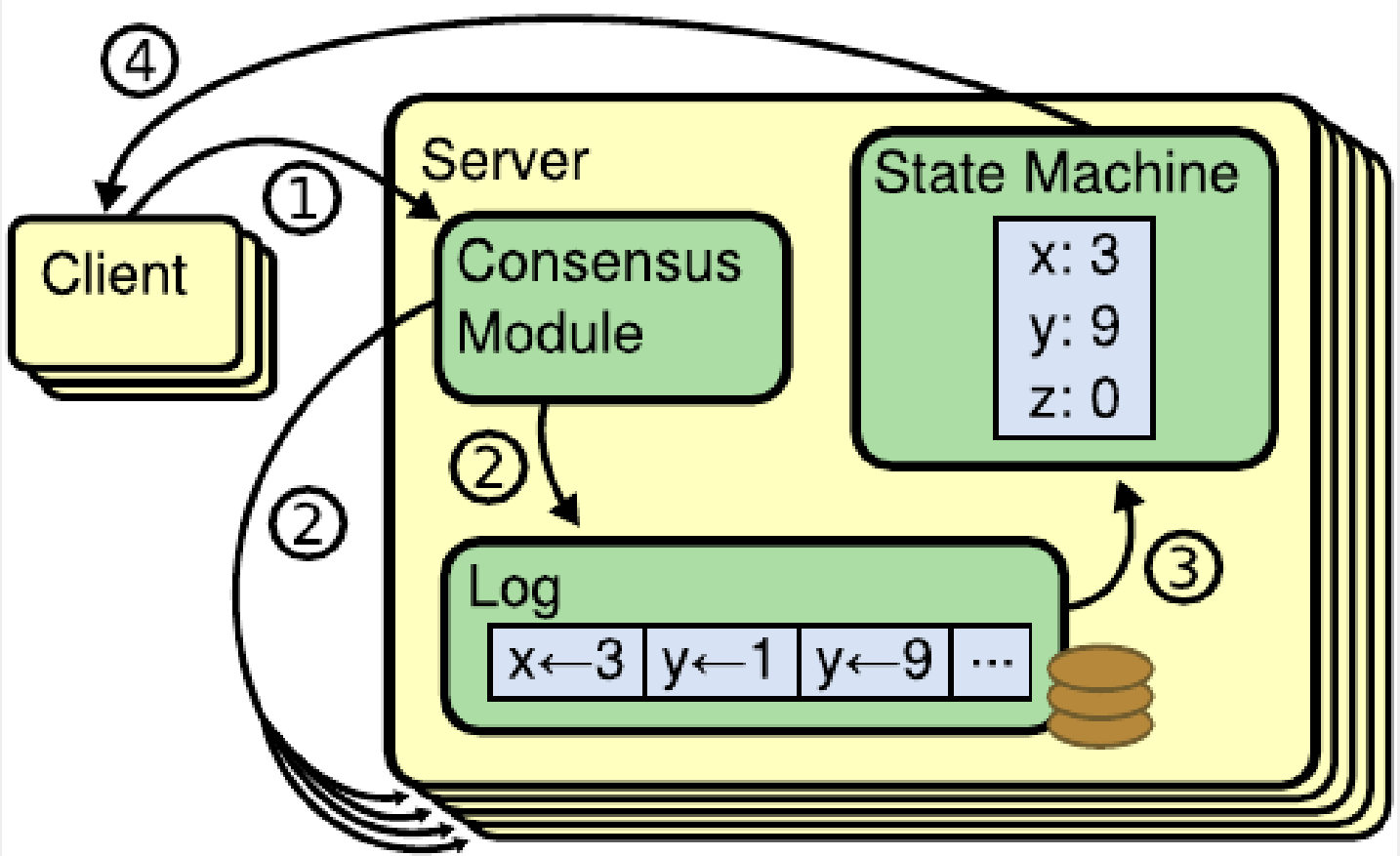
Background: important system properties

- Safety
  - Something bad will never happen
  - Consensus: no nodes will decide on different values

- Liveness
  - Something good will happen (eventually)
    - No specification on the time bound
    - System will make progress despite concurrent execution
  - Consensus: all nodes will eventually decide on a value

Background: replicated state machine model

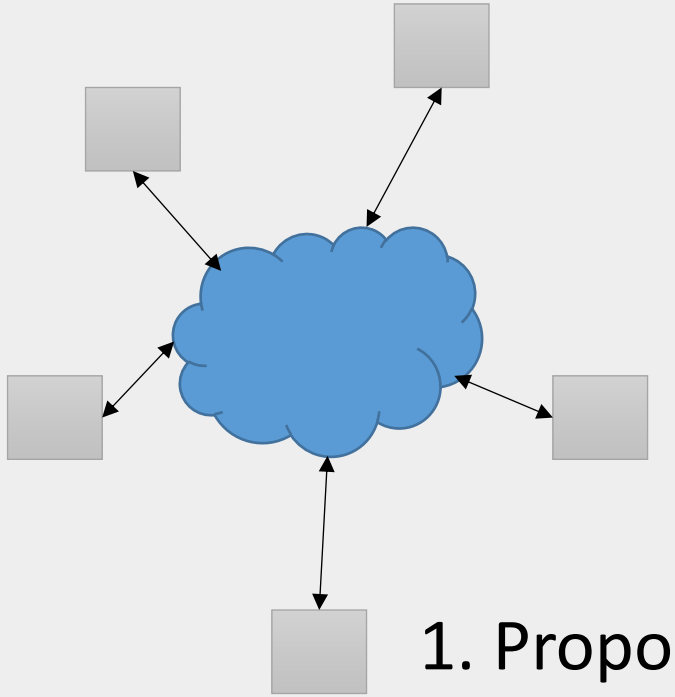
- General model for implementing fault-tolerant services by replicating servers
- Lamport: replicating a service over multiple nodes is *easy* if you present the same sequence of input commands to all replicas and they follow the same succession of states



Lamport "Time, Clocks, and the Ordering of Events in a Distributed System"

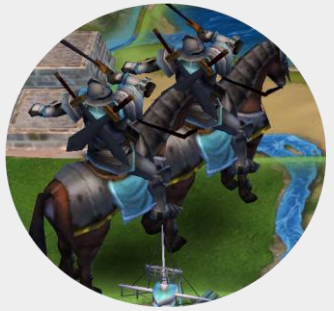
How do we achieve consensus?

- Agreement of many *nodes* on shared state (i.e., a data value)
- General model
  - *Proposes* a value by suggesting to others
  - *Decides* based on what it thinks everyone agrees on
- Relied on by:
  - Distributed databases, group membership systems, replicated state machines, data stores, message queues, anywhere we need mutual exclusivity …
  - Underlies most distributed systems: banks, cloud providers, online video services, retailers, really any distributed system with fault tolerance



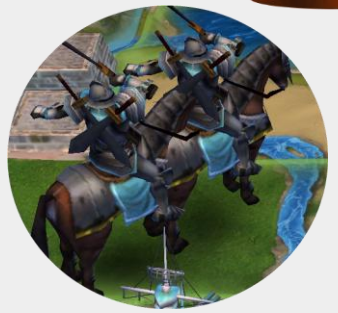
1. Propose
2. Decide

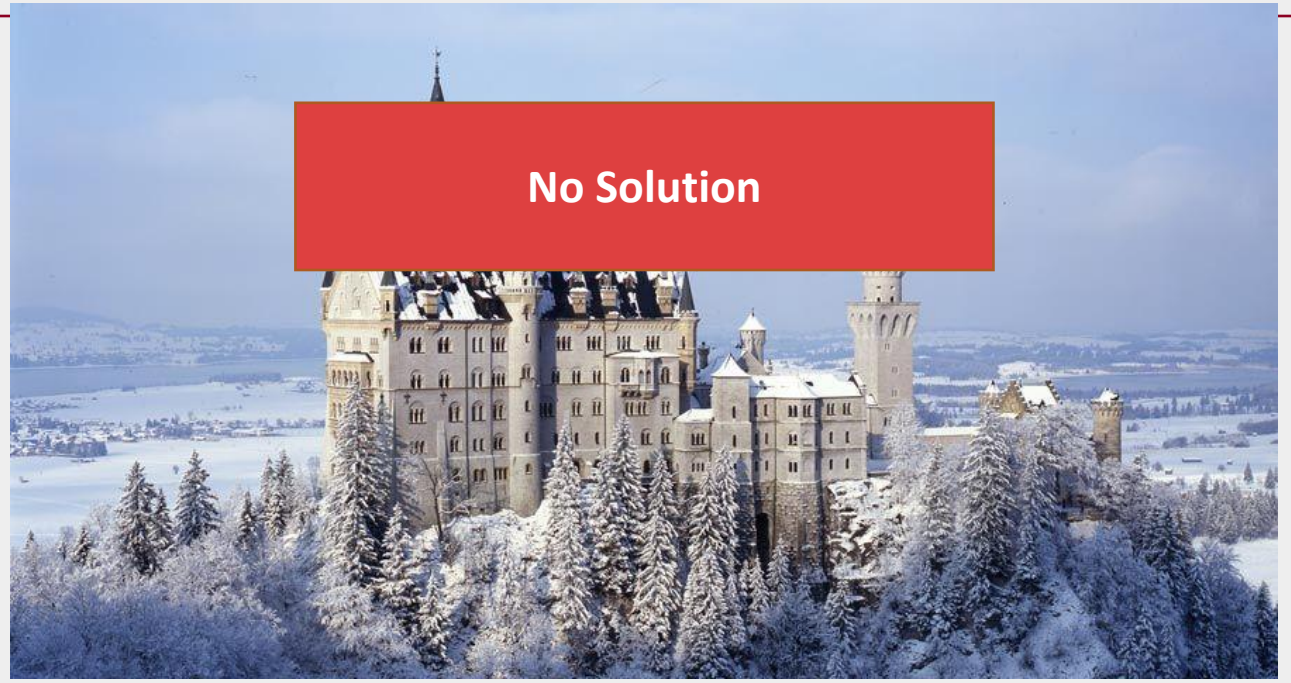
Two Generals Problem

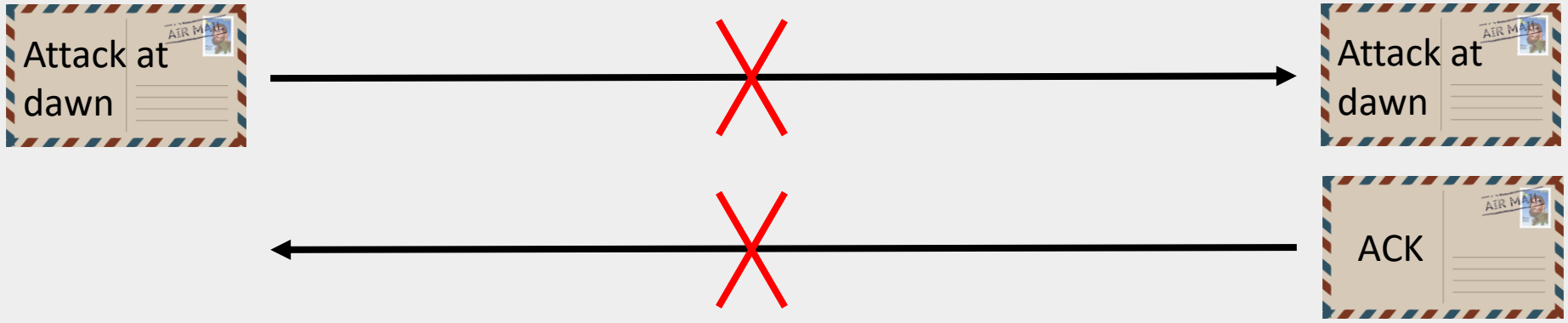


Do I know if A received my confirmation?

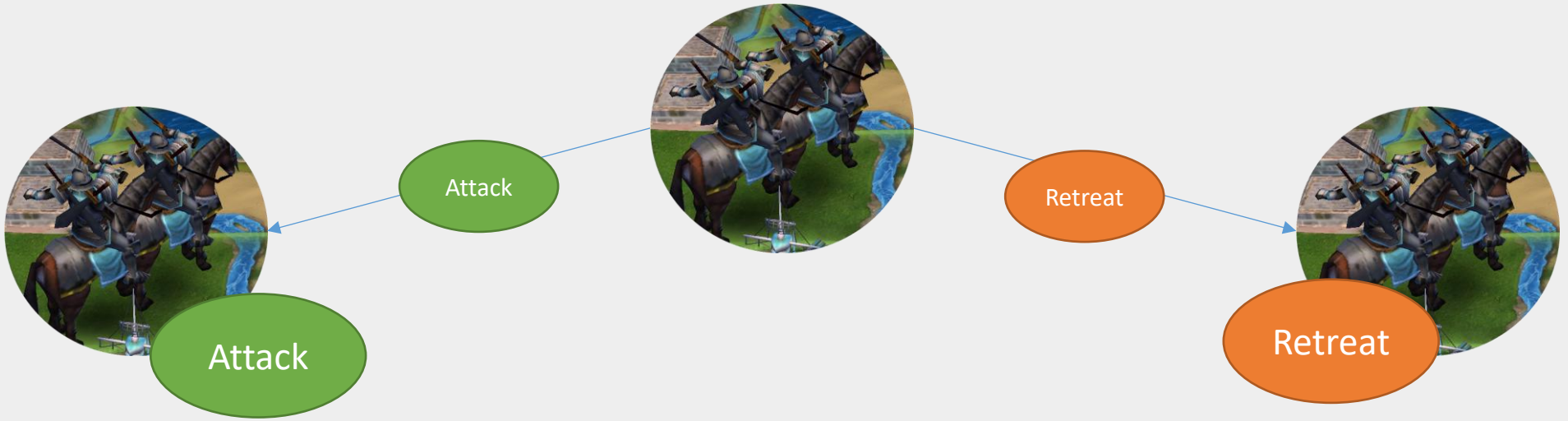
Two Generals Problem

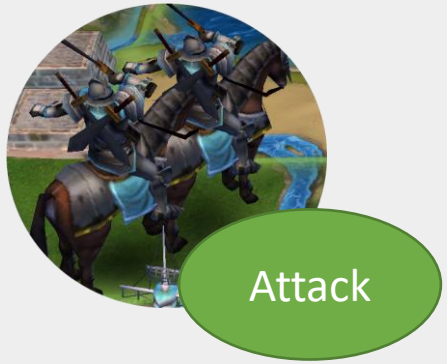






Byzantine General's Problem





Fischer-Lynch-Patterson (FLP result) - 1985

- One of the most important results in distributed systems

- Consensus was known to be solvable in synchronous systems..
- FLP: consensus cannot be guaranteed in an asynchronous distributed system within bounded time if a node *might* fail
  - We cannot reliably detect failure (between a crashed node and a slow node)

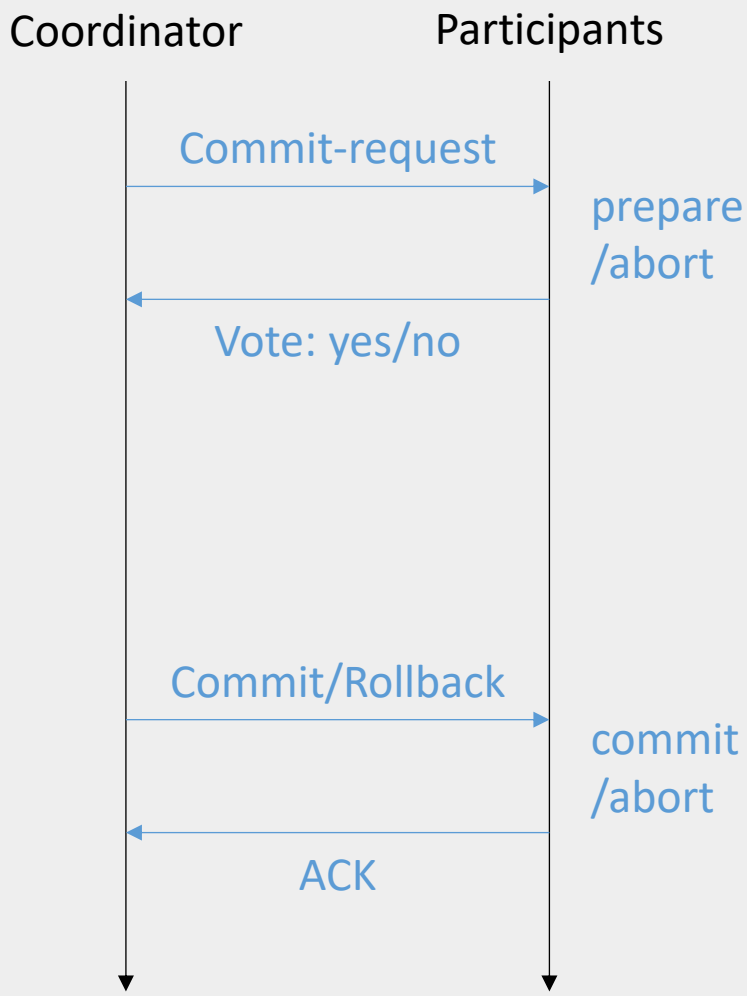
⇒ No asynchronous consensus algorithm can guarantee both safety and liveness

⇒ Safety: something bad will never happen

⇒ Liveness: something good will happen

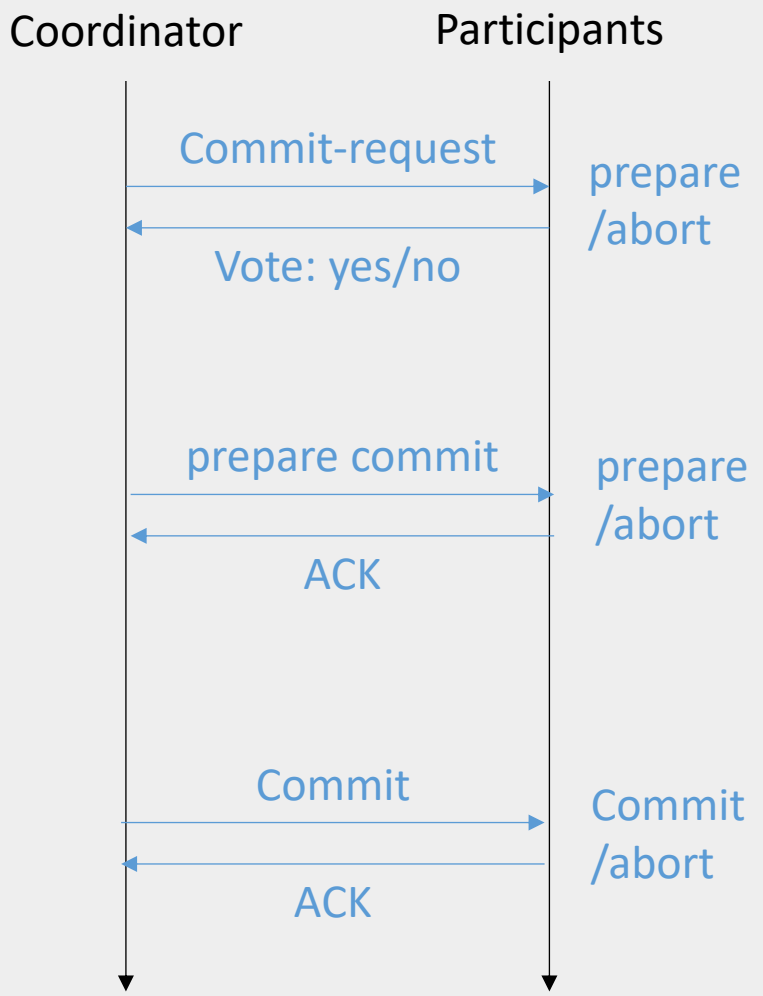
2 phase commit

- Phase 1: commit-request (or voting)
  - Coordinator suggests value to all nodes and gathers responses (yes/no)
  - Participants execute transaction up to commit
- Phase 2: commit
  - If everyone agrees, coordinator tells nodes the value is final, they commit, then ACK
  - If anyone disagrees, tell all nodes to rollback, then ACK
- Does this solve the consensus problem?
  - What happens if a node crashes?
    - Participant? Coordinator?

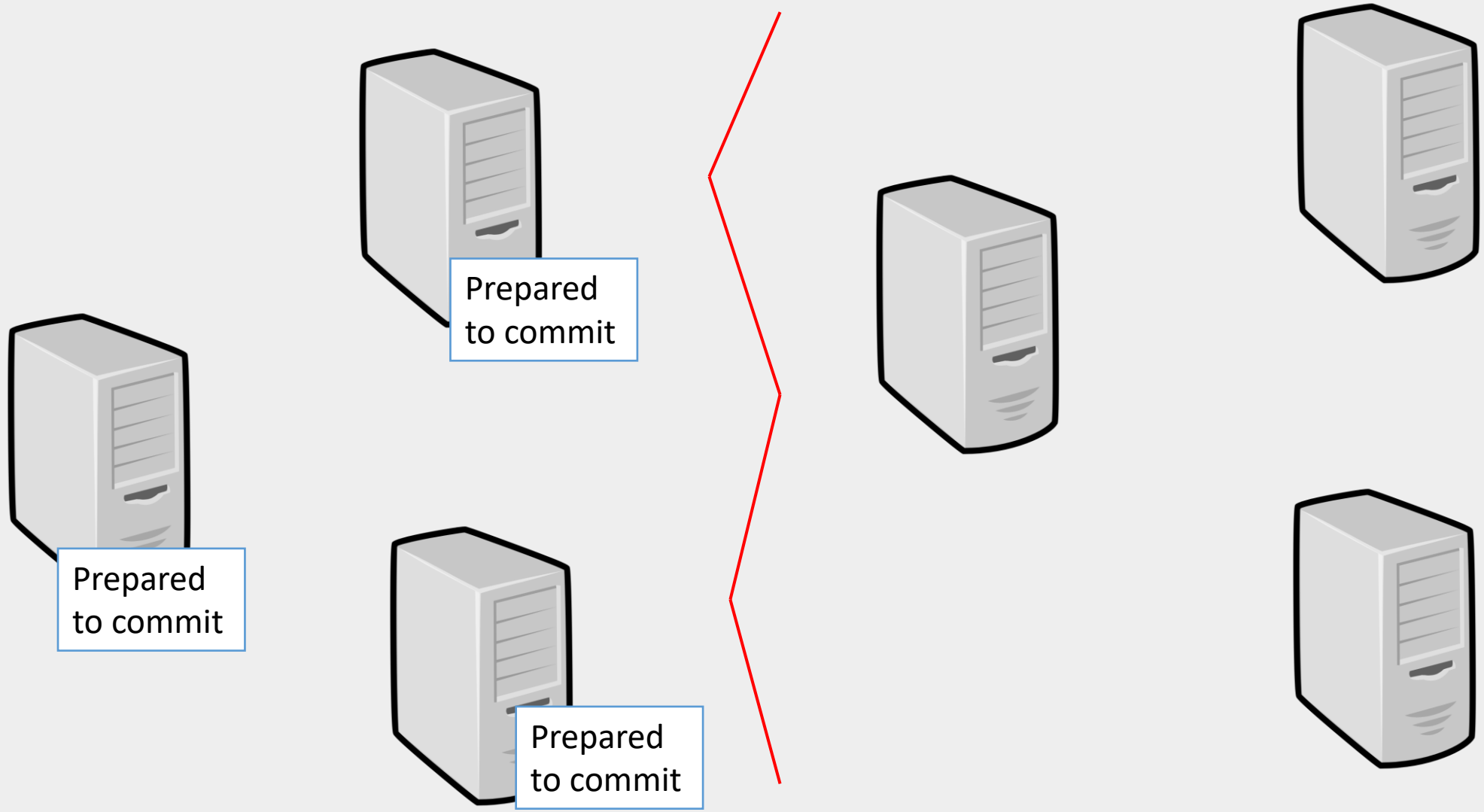


3 phase commit

- Phase 1: voting
  - Coordinator suggests value to all nodes and gathers responses (yes/no)
- Phase 2: prepare commit
  - If all reply yes then coordinator sends “prepare commit”
  - Each node executes up to commit and ACKs receipt of message
- Phase 3: commit
  - If coordinator receives ACK from all nodes it tells all nodes to commit
  - If any node says no it aborts



Resilience to network partition?



Part 3: Paxos

The part-time parliament

- Parliament to determine the law of the land
  - Represented by a sequence of decrees

155: *The olive tax is 3 drachmas per ton*

- No member is willing to remain in Chamber to record actions
  - Each Paxon legislator maintains their own ledger (written in indelible ink) to record decrees
- No two ledgers can have a different entry for that decree
  - Trivially met by leaving ledgers blank … need some way to guarantee progress (liveness)
- Legislators are willing to pass any decree that is proposed
  - Issue: if one group passes a decree and then another group enters and passes a conflicting decree

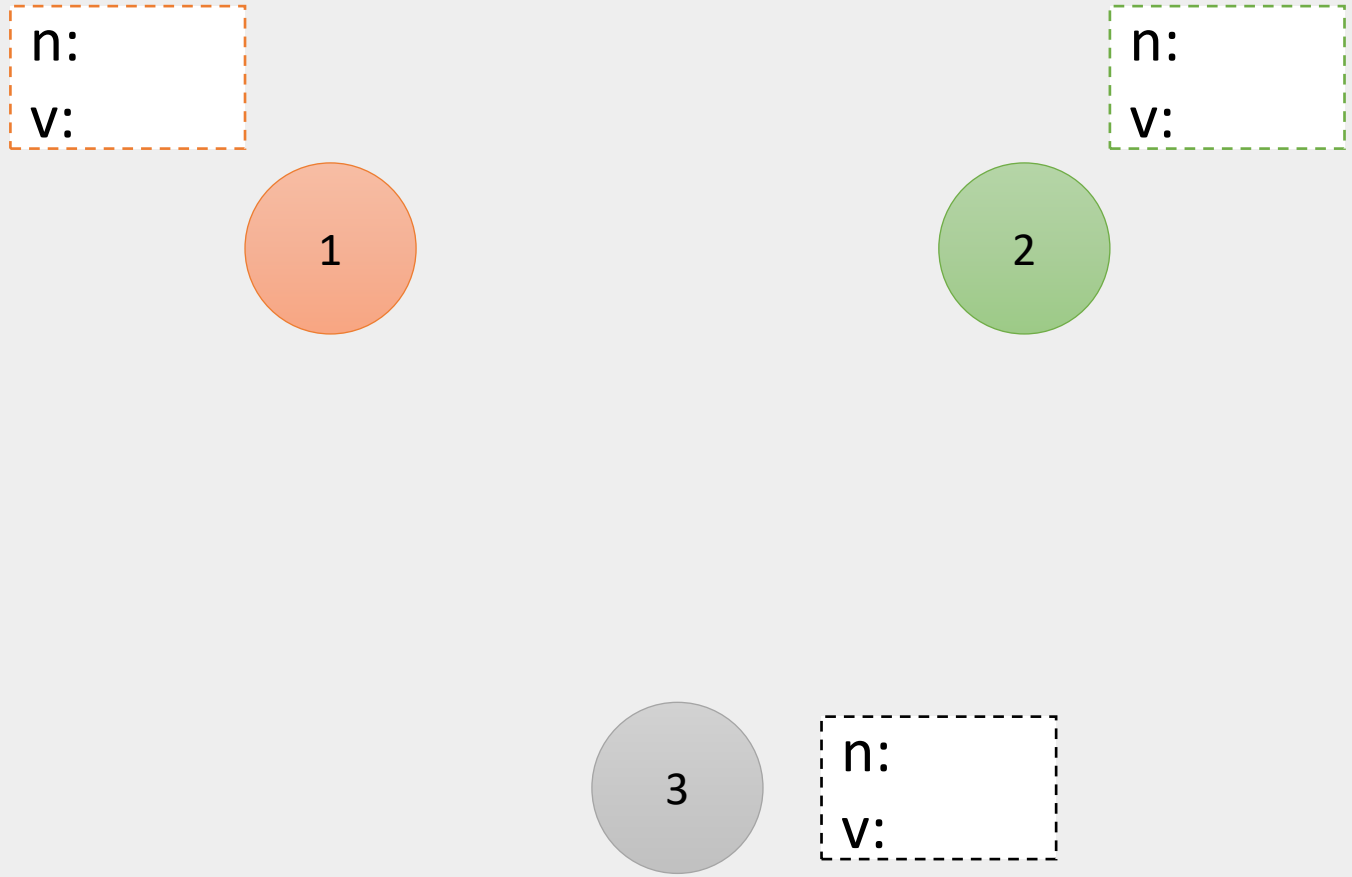
Basic protocol

- 1a: Prepare
  - A proposer selects a proposal number *n* and sends a *prepare* request with number *n* to a majority of acceptors (n > any number used by proposer)
- 1b: Promise
  - If an acceptor receives a *prepare* request with number *n* greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than *n* and with the highest-numbered proposal (if any) that it has accepted
  - Otherwise, the acceptor can ignore the proposal.

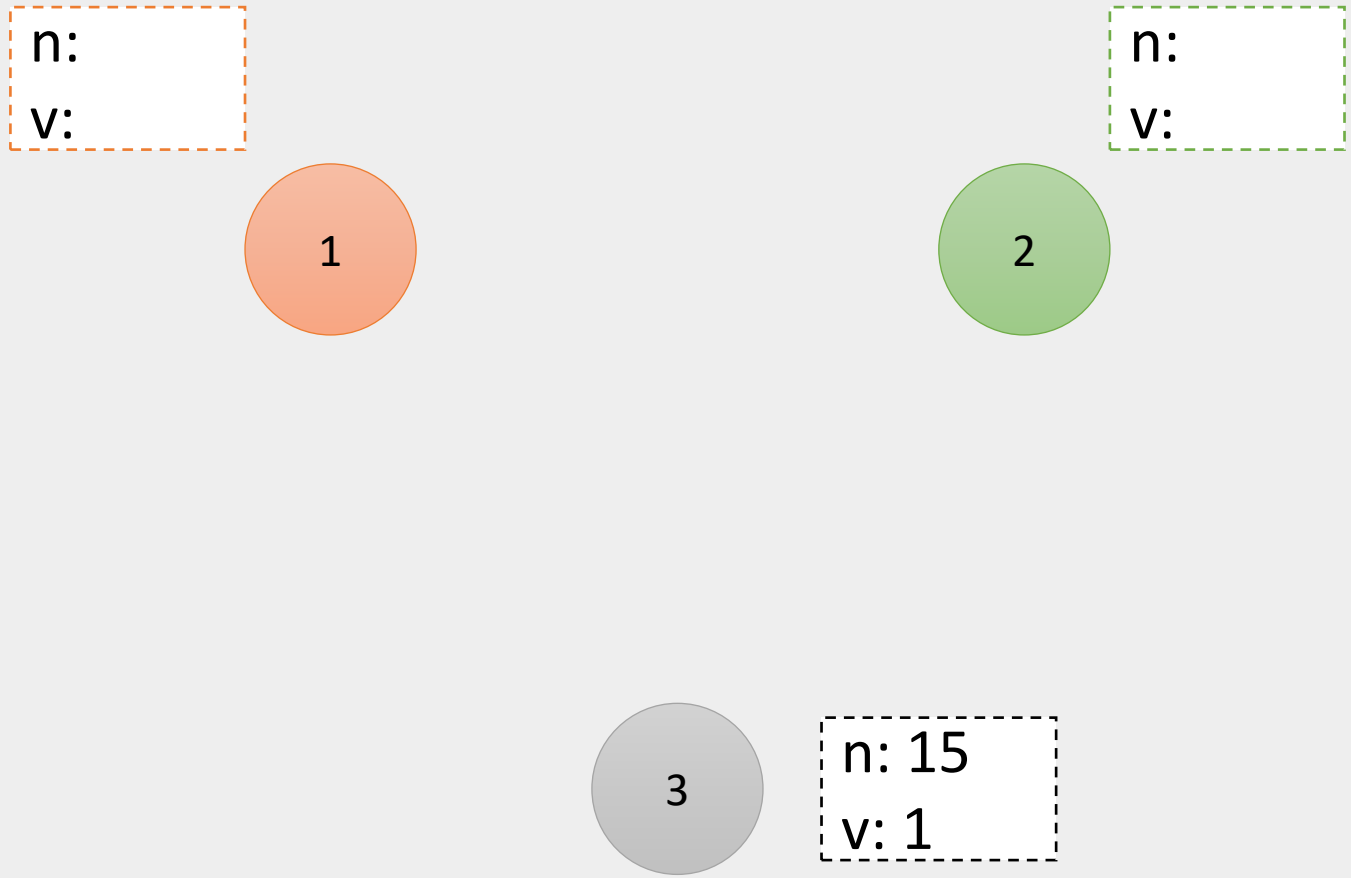
Basic protocol

- 2a: Accept
  - If the proposer receives a response to its *prepare* requests (numbered *n*) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered *n* with a value *v*, where *v* is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.
- 2b: Accepted
  - If an acceptor receives an *accept* request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than *n*. It should send an accept message to the proposer and every learner.

Paxos: No failure

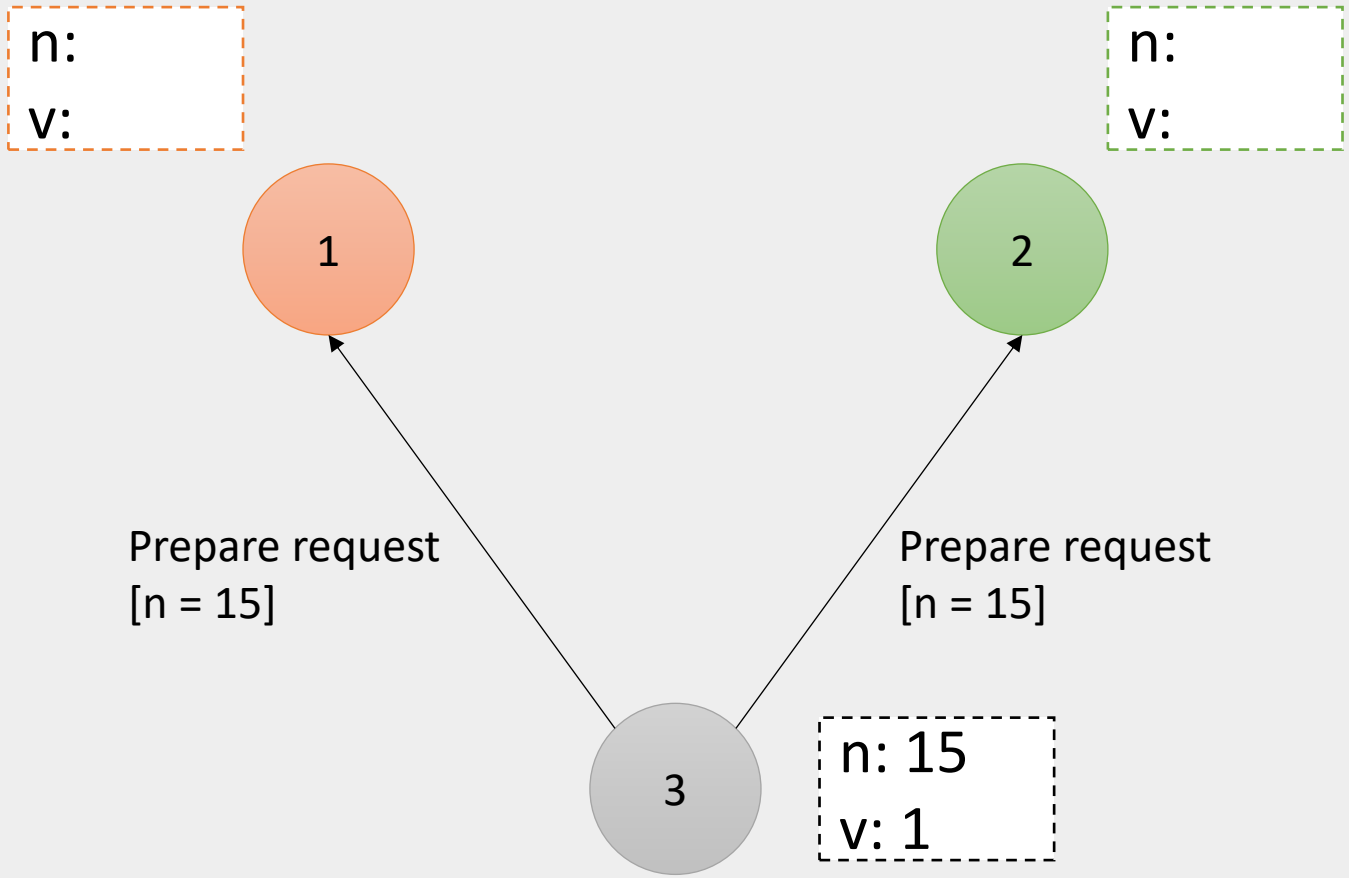


Paxos: No failure

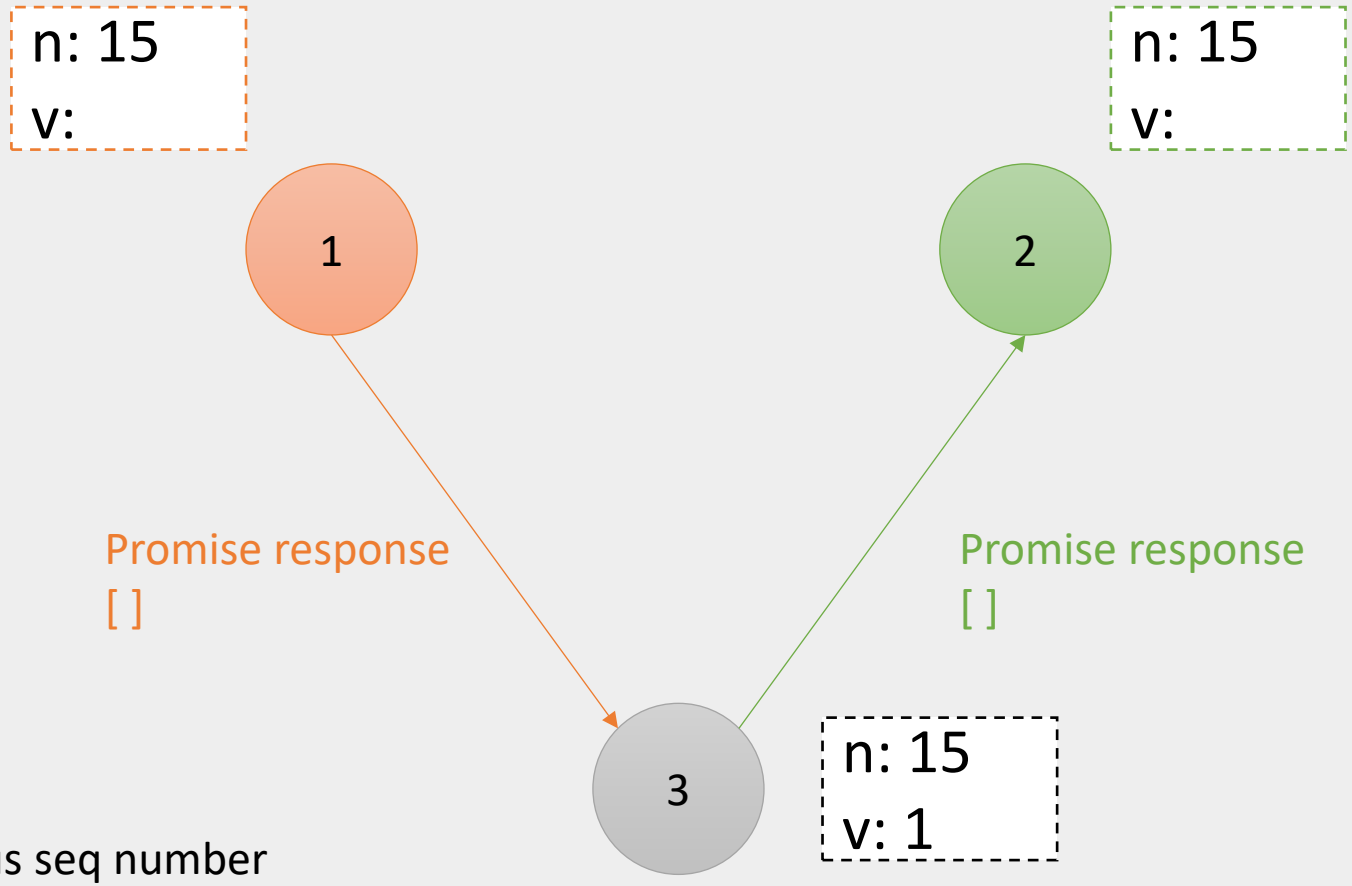


Triggered by a user action (e.g., update a value)
Pick n greater than we've used before

Paxos: No failure

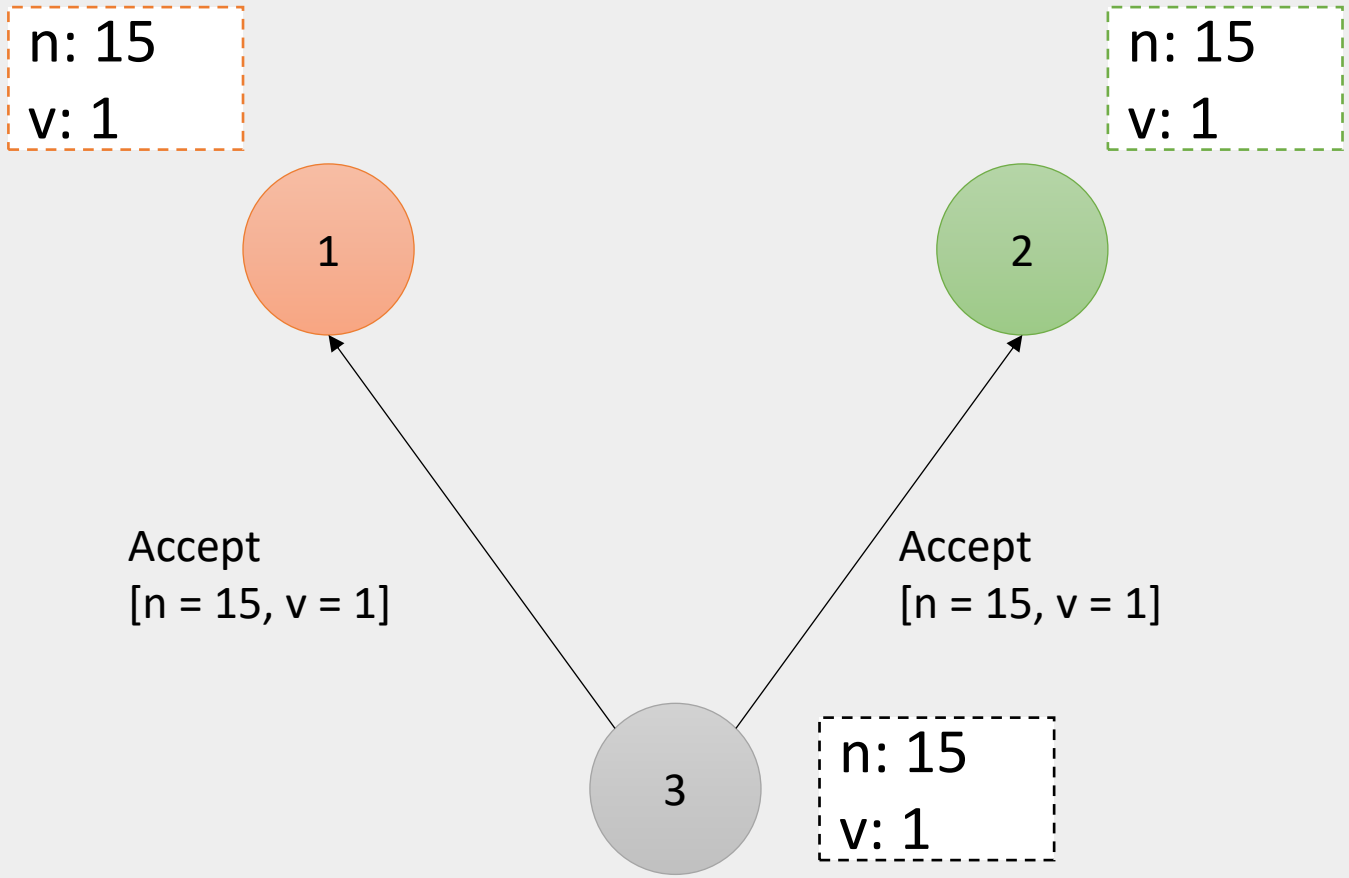


Paxos: No failure

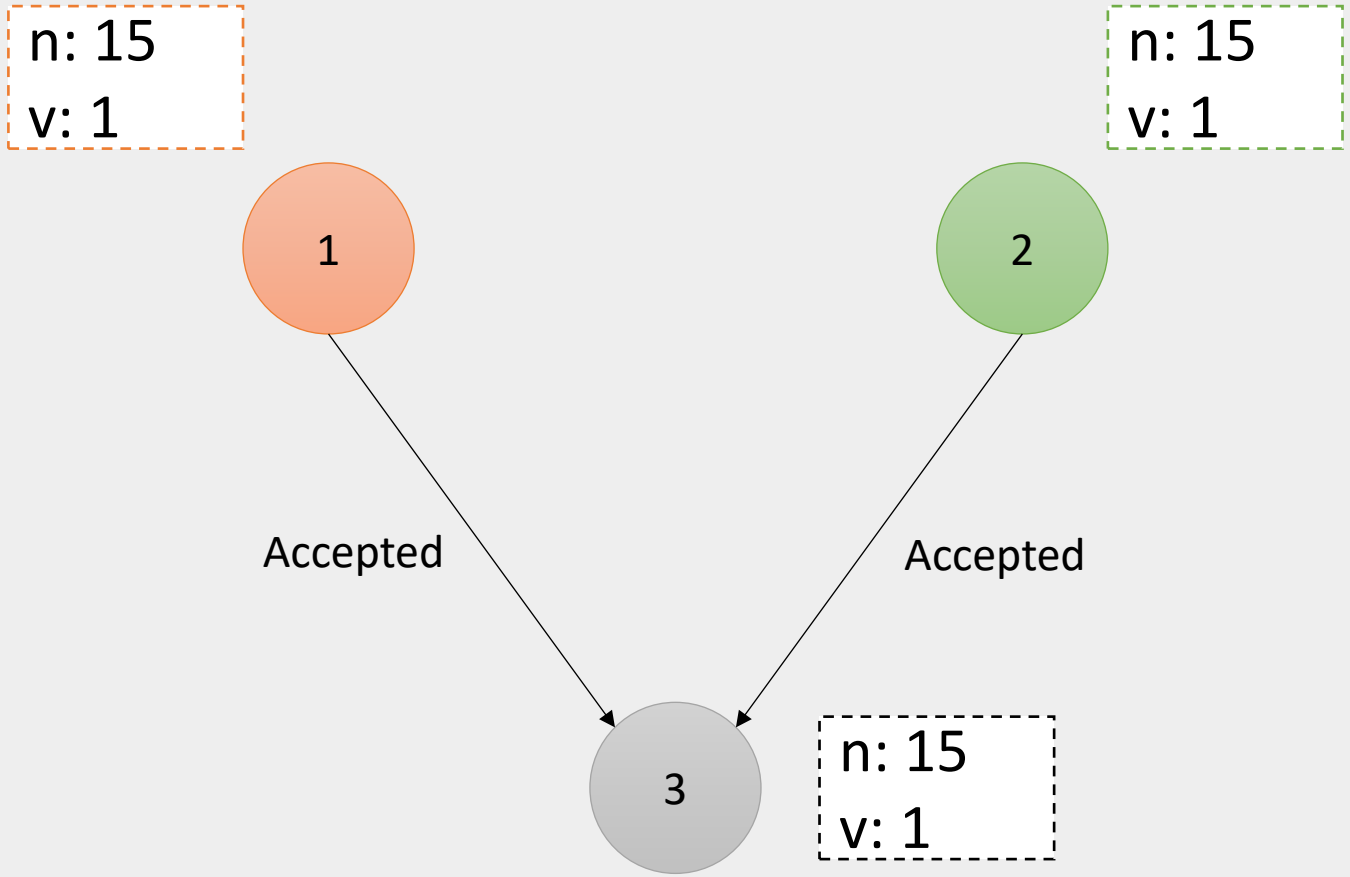


Check n greater than previous seq number and promise not to accept another proposal (if they have a higher sequence number reply with that value)

Paxos: No failure



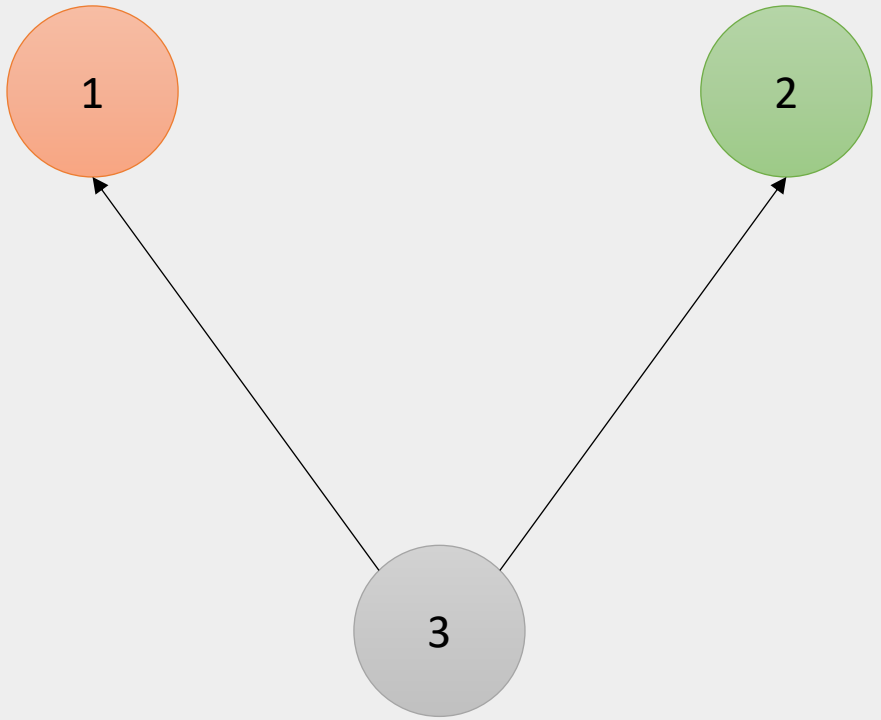
Paxos: No failure



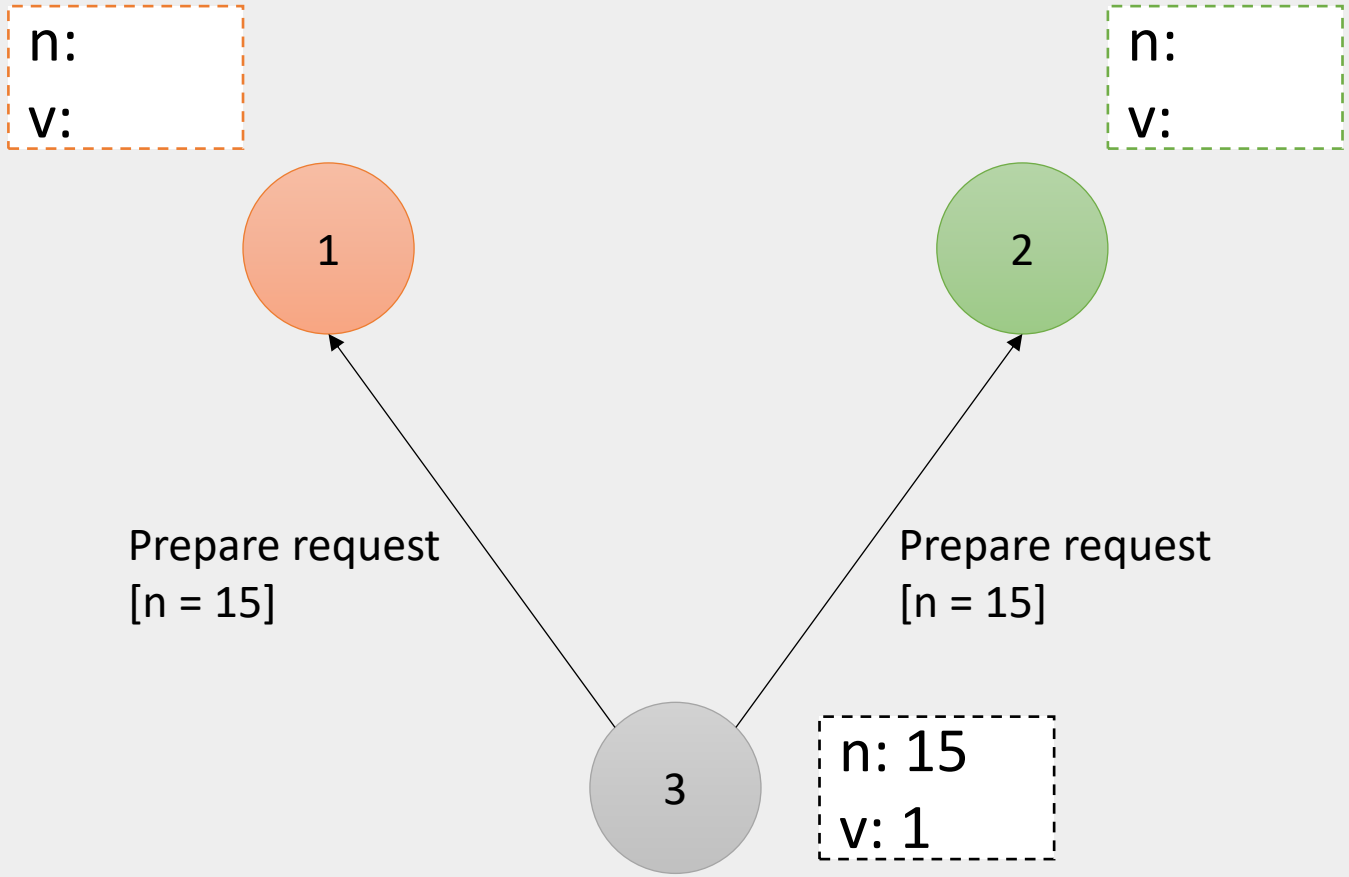
What if a node fails?

Node 1 or 2?

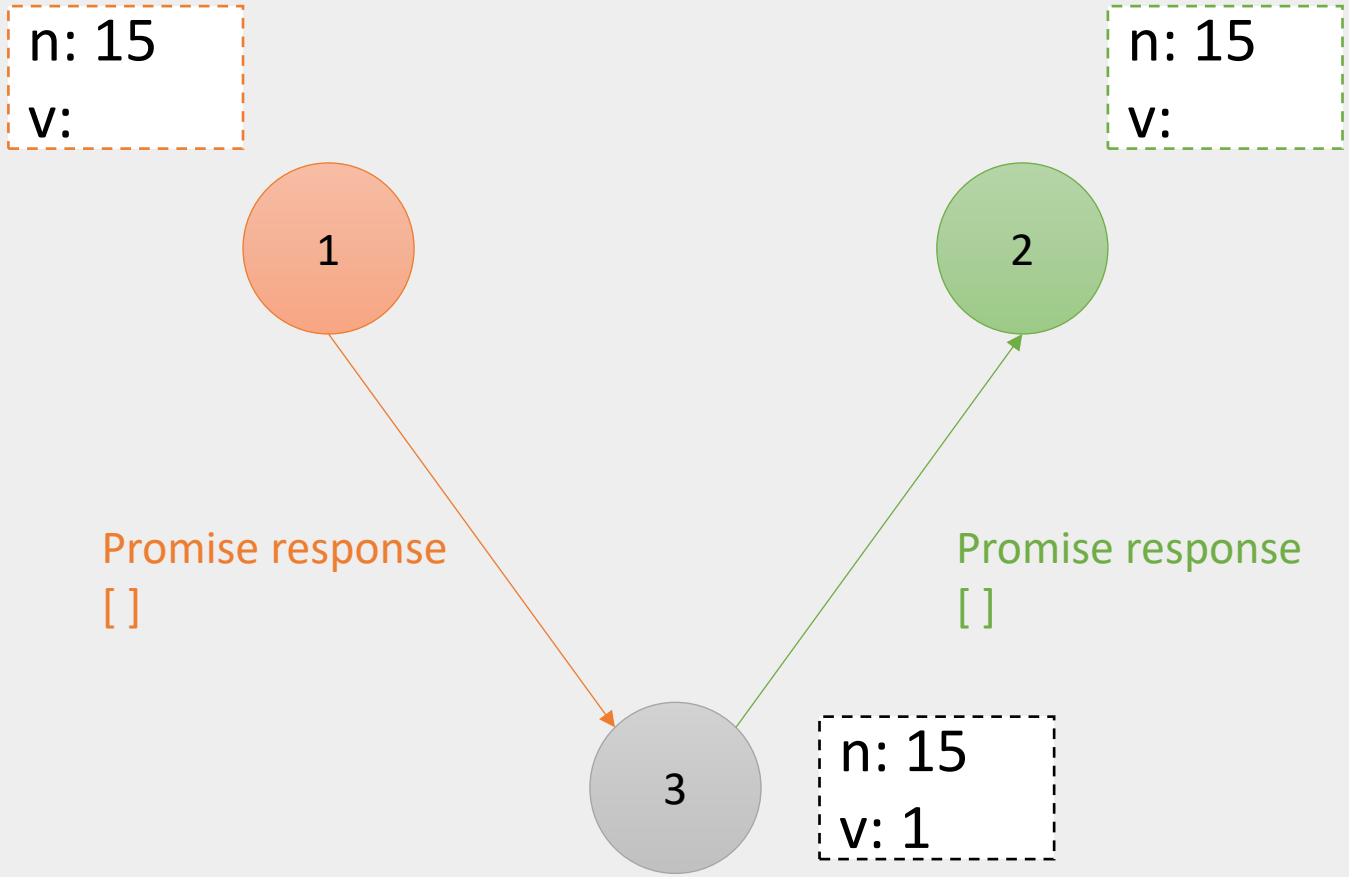
- Not a problem as we only need a quorum



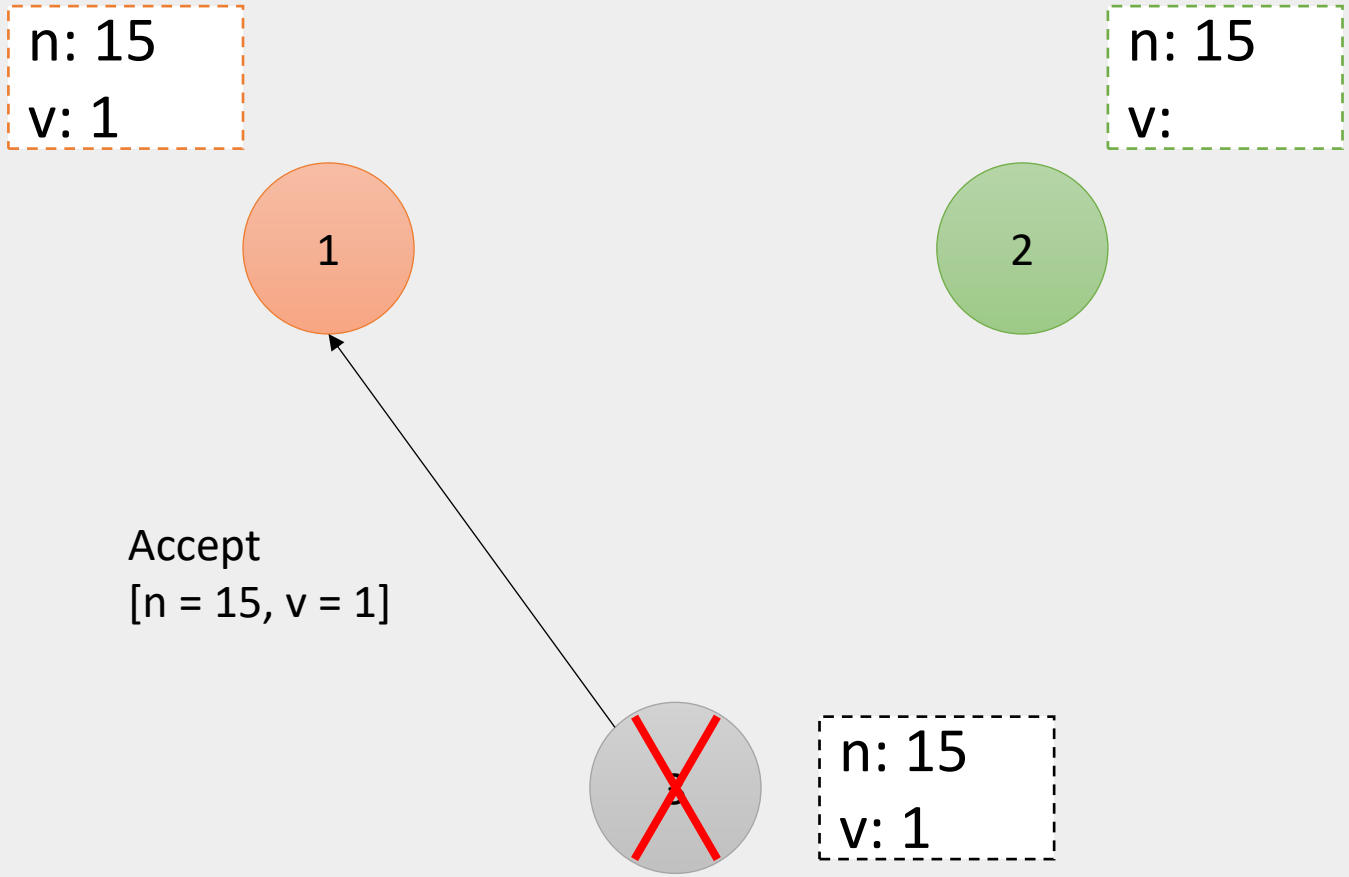
Paxos: Leader node failure (without conflict)



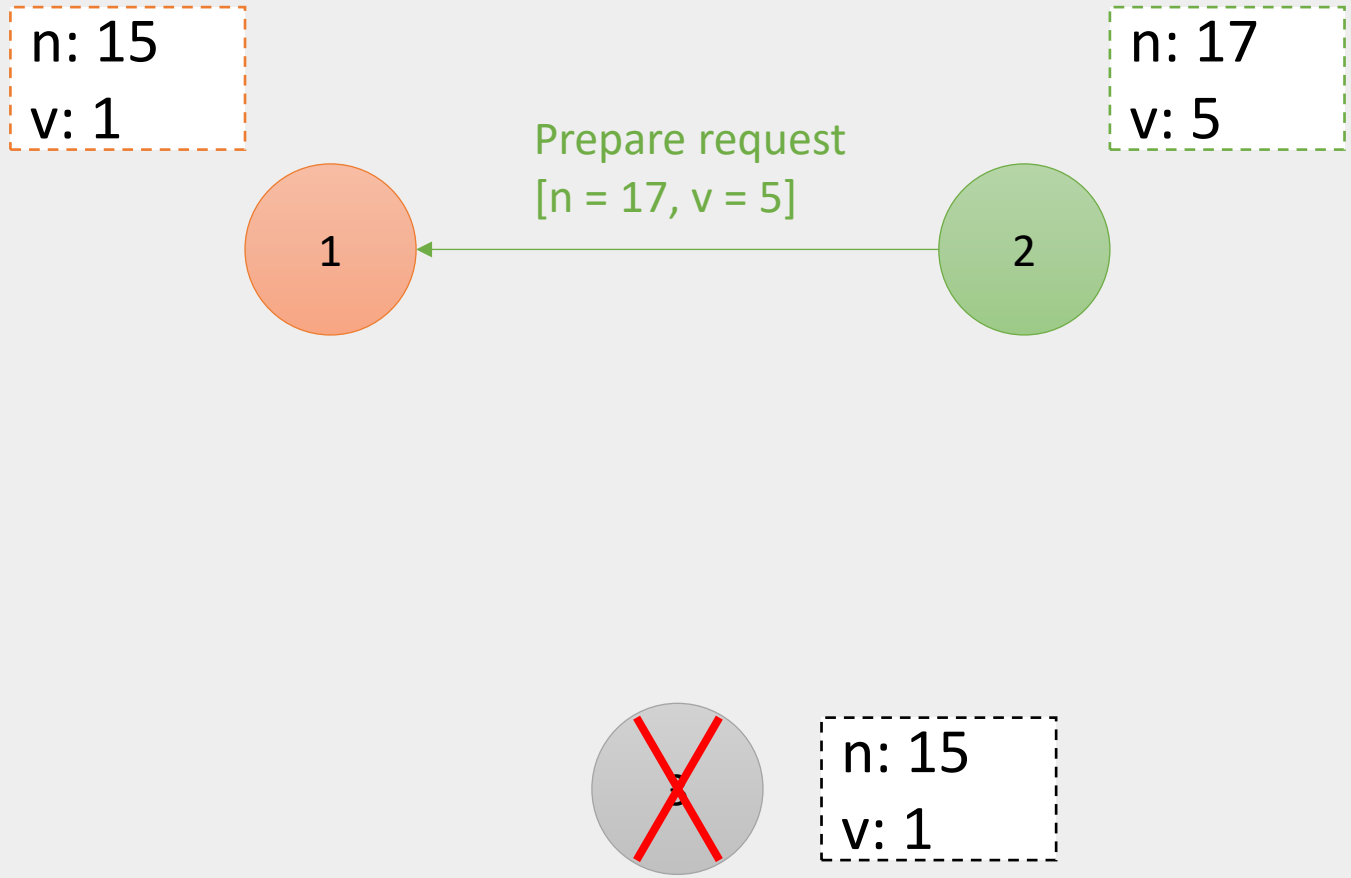
Paxos: Node failure (without conflict)



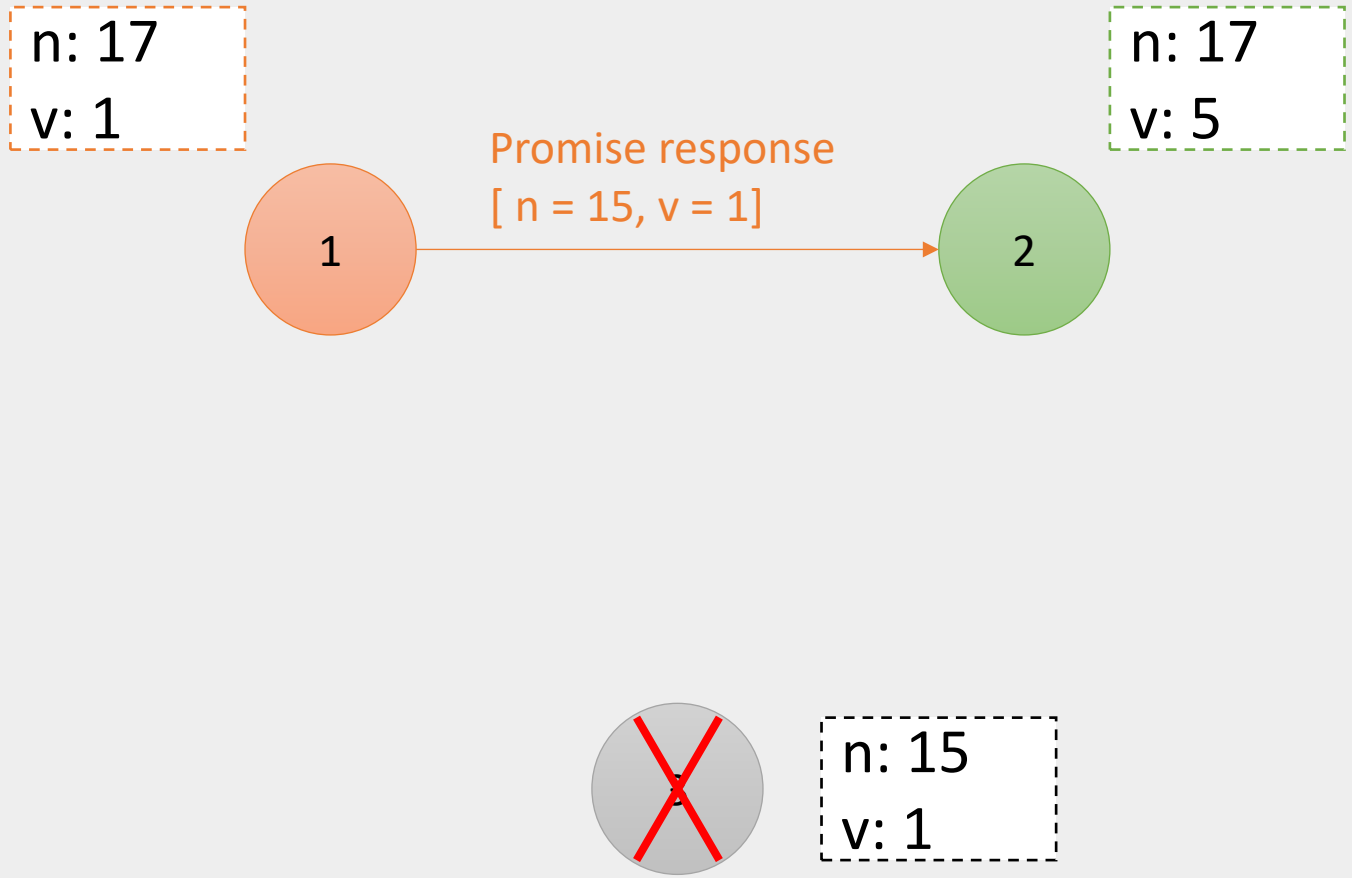
Paxos: Node failure (without conflict)



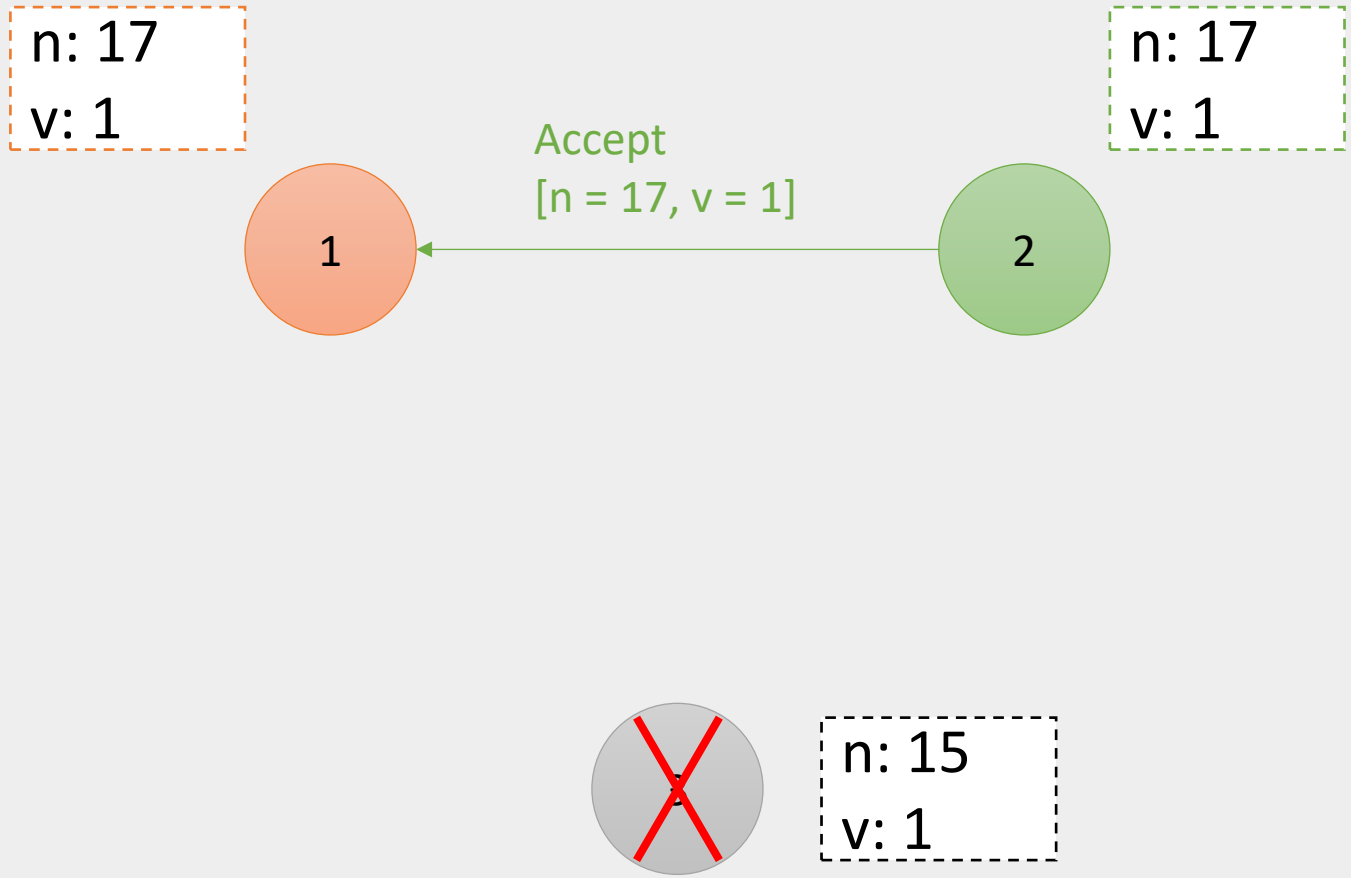
Paxos: Node failure (without conflict)



Paxos: Node failure (without conflict)



Paxos: Node failure (without conflict)



Failure modes

- Failure of an acceptor (with quorum alive)
  - No recovery needed (no additional messages required)
- Failure of a learner
  - No recovery needed (no additional messages required)
- Failure of proposer without conflict
  - As we just saw
- Failure of proposer with conflict
  - E.g., when leader recovers
  - Can lead to many unsuccessful rounds (protocol may stall)

Guarantees safety not liveness

Paxos adds two things over 2PC

- Paxos adds two important mechanisms to 2PC
- The first is ordering the proposals so that it may be determined which of two proposals should be accepted
- The second is to consider a proposal accepted when a majority of acceptors have indicated that they have decided upon that proposal

Summary

- Guarantees safety (consistency)
  - Can withstand f failures with $2f + 1$ acceptors
- Liveness?
  - No, but conditions for preventing progress are rare
- Difficult to implement
- Many variants and optimizations (with different degrees of specificity
- Any node can commit a value in 2RTT
  - In Multi Paxos the leader can do it in 1RTT
- Paxos + replicated state machine == fault tolerant services

Paxos in practice?

"We found few people who were comfortable with Paxos, even among seasoned researchers"

"We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year."

– Ongaro and Ousterhout. In search of an understandable consensus algorithm

"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."

– Google, "Paxos made live"

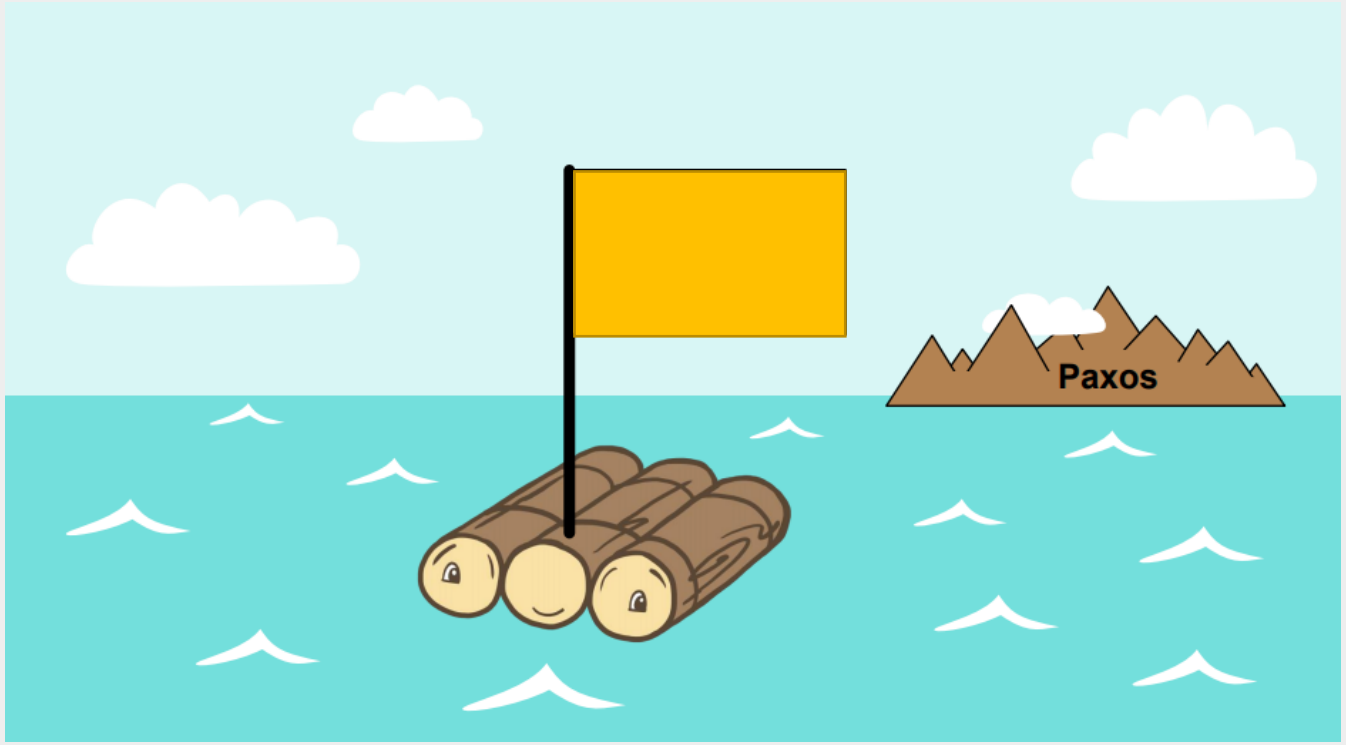
Part 4: Raft

Raft: an “understandable” consensus algorithm

An effort to make design decisions based on understandability

- Strong leader model

1. Problem Decomposition
2. Minimize state space
   - Multiple problems with single mechanism
   - No special cases
   - Minimize nondeterminism

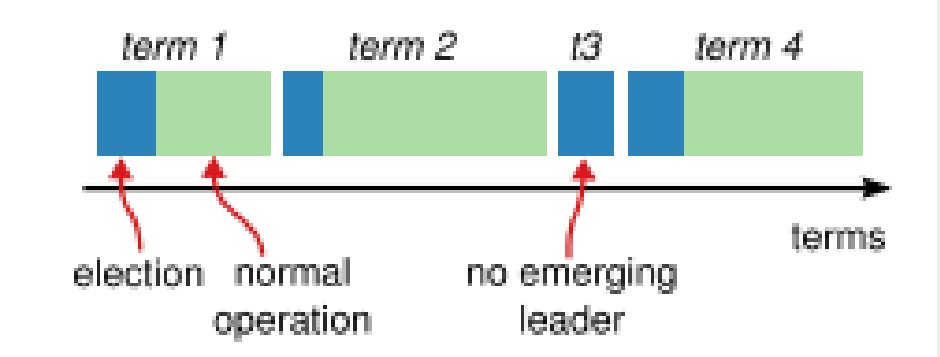


Problem decomposition: 3 core problems

1. Leader election:
   - Select one server to act as leader
     - All message passing initialized by leader (or node attempting to become leader)
     - Detect crashes, choose new leader
2. Log replication (normal operation)
   - Leader accepts commands from clients, appends to its log
     - Note: clients must communicate with leader
   - Leader replicates its log to other servers (overwrites inconsistencies)
3. Safety
   - Keep logs consistent
   - A few tricks e.g., only servers with up-to-date logs can become leader

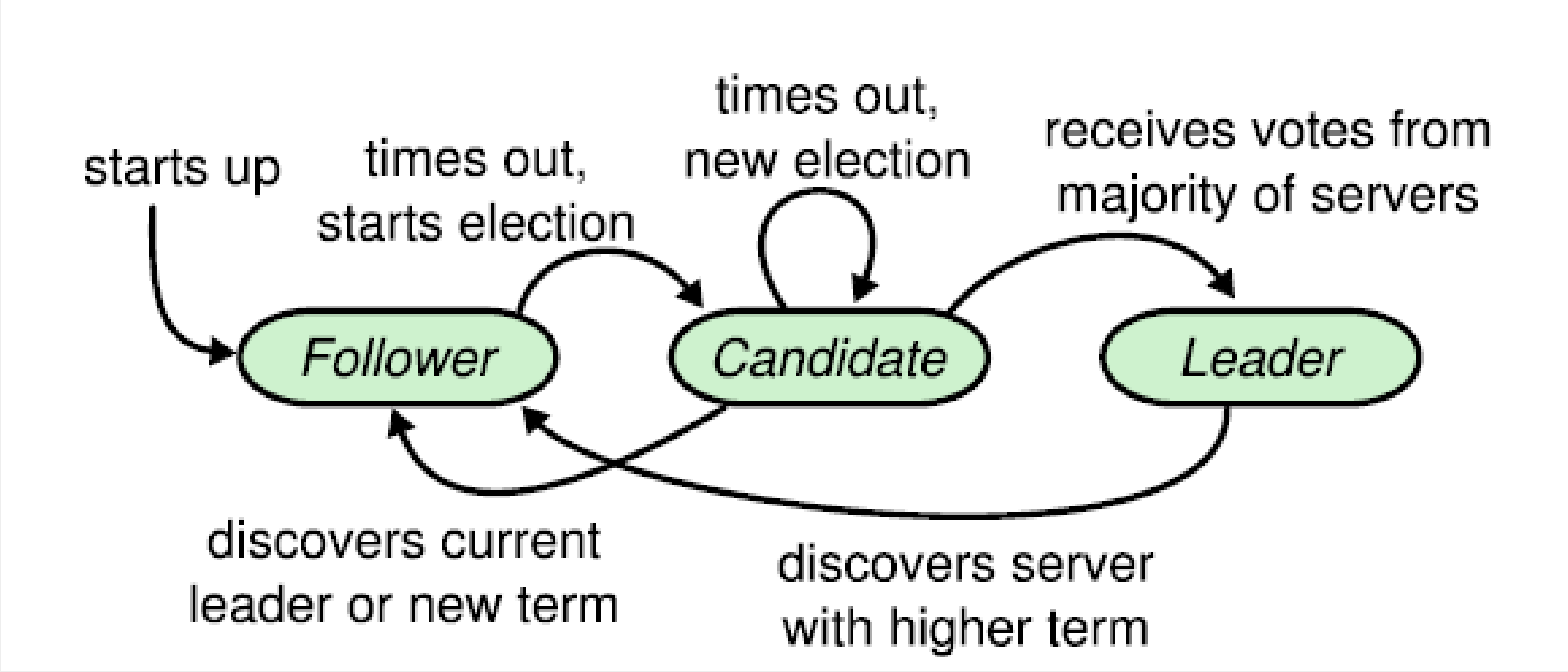
Terms

- Logical clock model
  - Each node has its own local view of time (its current term)
  - Current term increases monotonically over time



- Every Raft communication includes (and compares) current term
  - Only 1 leader per term
  - Some terms have no leader (failed election)
- Terms are updated when a node starts an election or when the current term communicated from another server is higher than its own
- Communication with a node with a higher term is rejected
  - If a candidate/leader learns of higher term it returns to being a follower

Raft protocol



Leader election

- Leader sends heartbeats, if a node does not get a heartbeat (or another message) from the leader within a timeout period it initiates an election

1. Follower increments current term and transitions to candidate state
2. Votes for itself
3. Issues *RequestVote* to each other node
4. Waits until:
   - Wins the election (majority of nodes vote)
   - Another server establishes itself as leader
   - A period of time passes without a winner (restarts election)

- Other nodes will vote for at most one candidate in a term using FCFS

Log replication

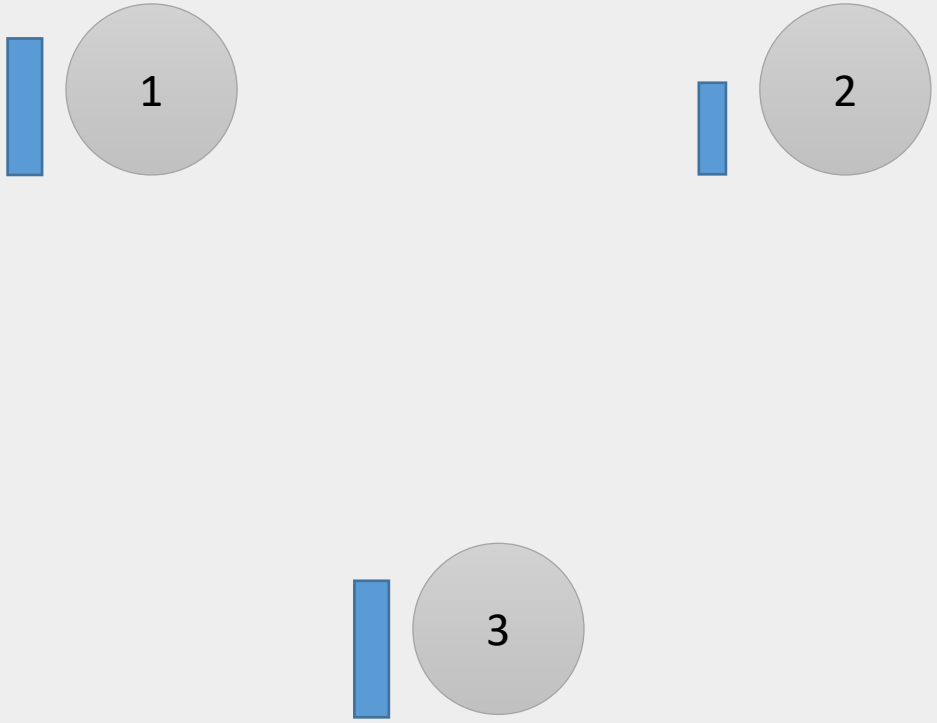
- Leader accepts commands from clients (commands to other nodes will be redirected to the leader)
- Appends command to its own log
- Issue *AppendEntries* RPC to each other node
- When command is applied to all other nodes, leader then commits it to their state machine (includes all prior entries)
- If followers crash or run slowly the leader retries indefinitely until all followers store all log entries

Leader Failure

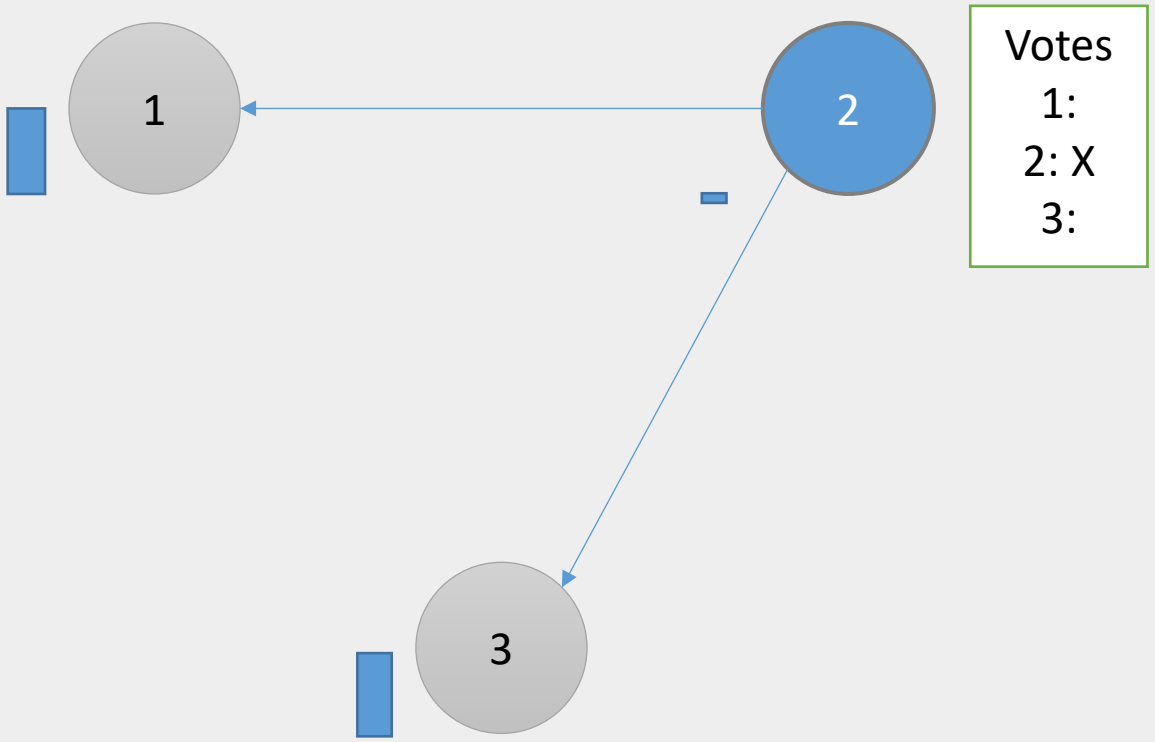
- The leader handles inconsistencies by forcing the followers' logs to duplicate its own (strong leader model)
- Find the latest log where the two logs agree, delete entries in the follower's log after that point, and send follower the leader's entries

- Safety guaranteed by several restrictions
  - Election restriction: voting process to prevent candidate from winning unless its log contains all committed entries
  - Committing from previous term: if leader crashes before commit, future leaders will attempt to complete the commit

Raft leader election



Raft leader election

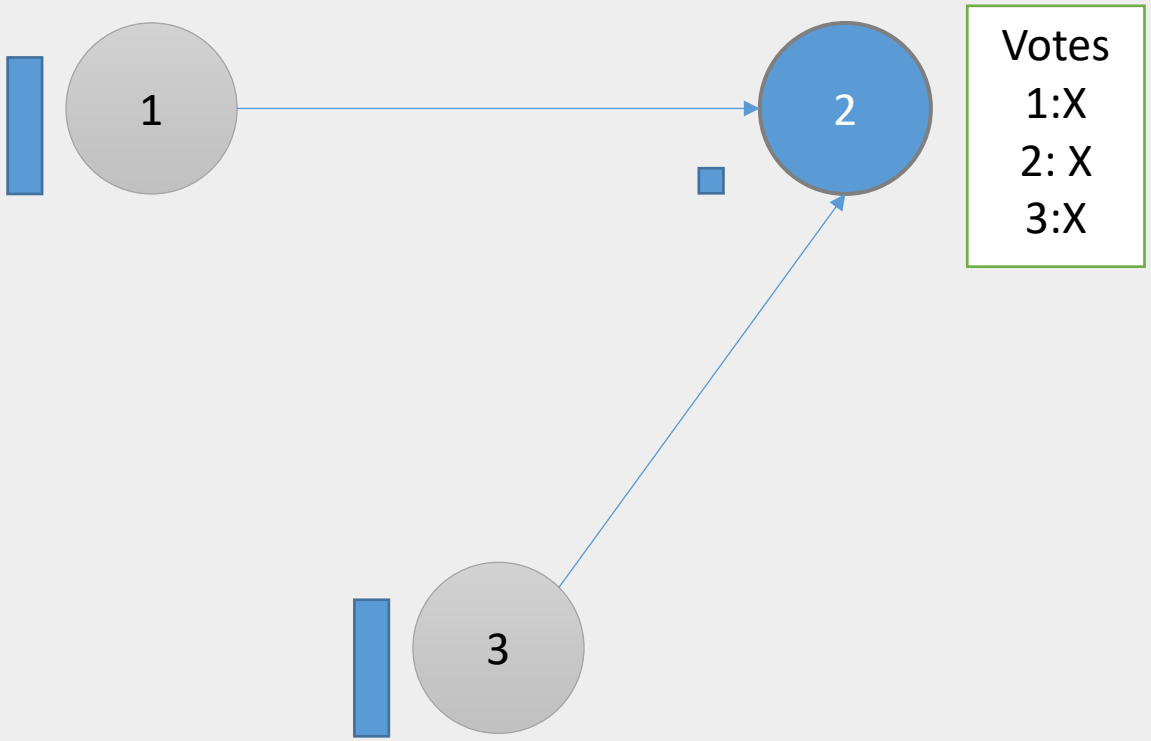


Follower

Candidate

Leader

Raft leader election

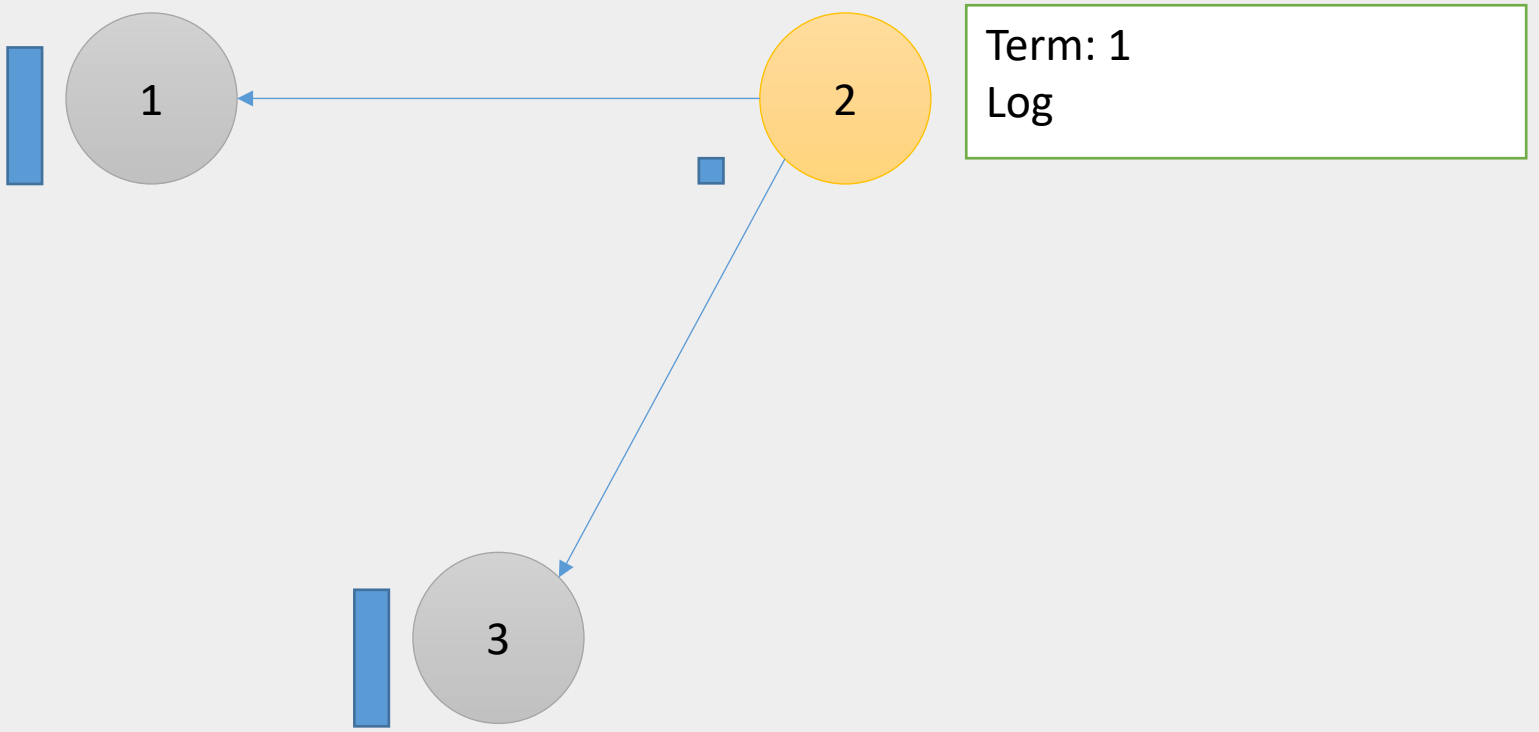


Follower

Candidate

Leader

Raft leader election



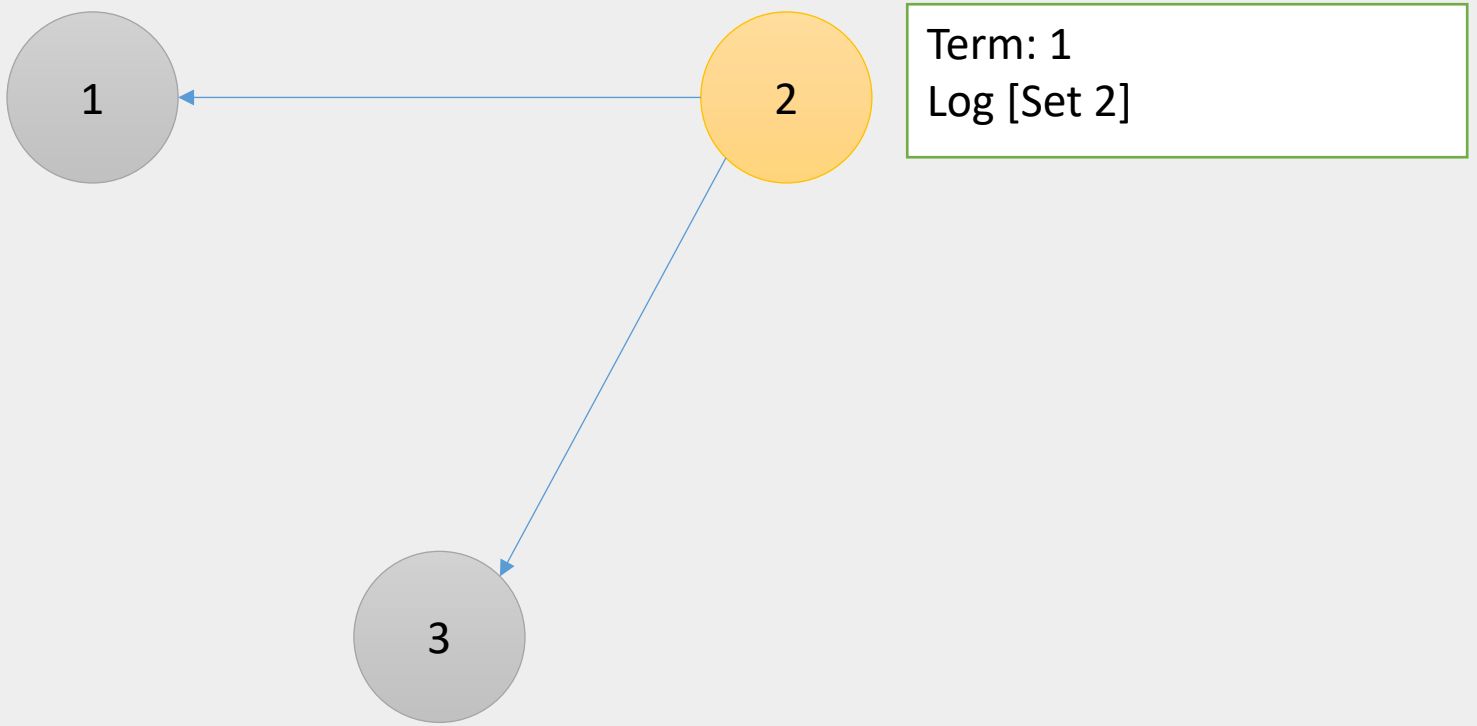
Follower

Candidate

Leader

How do we distribute updates

Raft setting a value

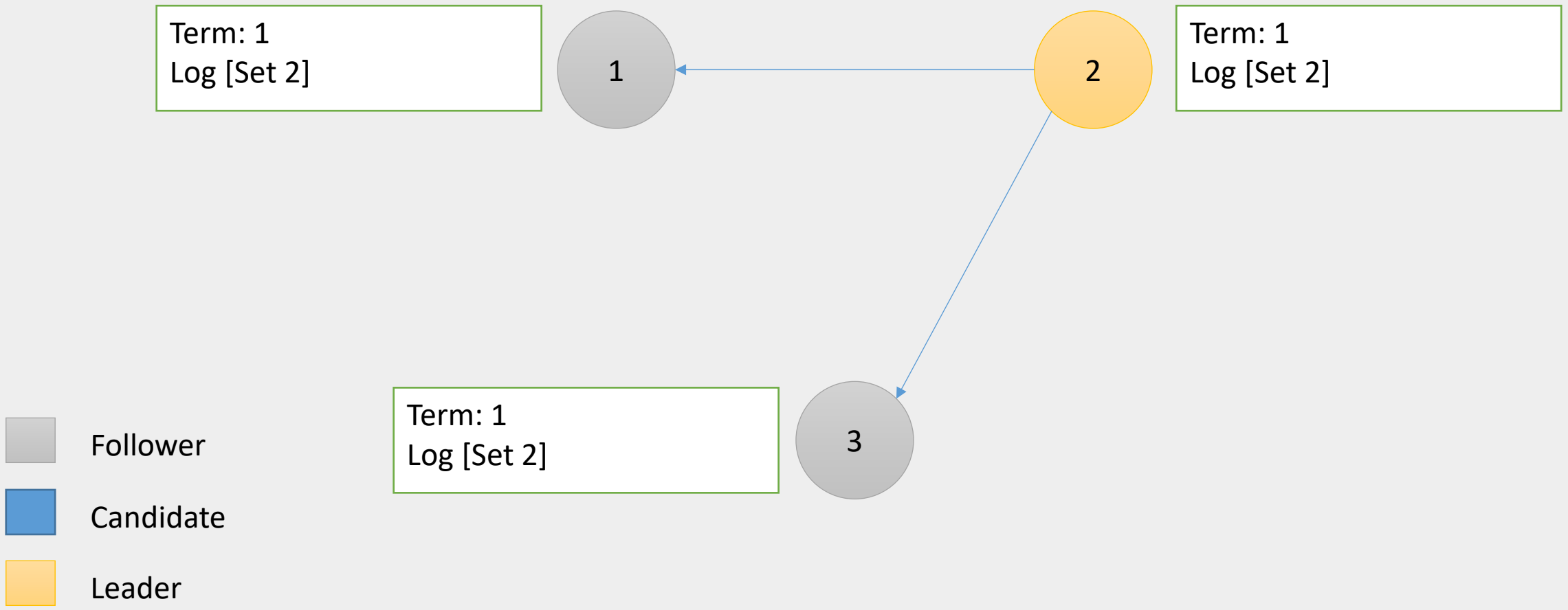


Follower

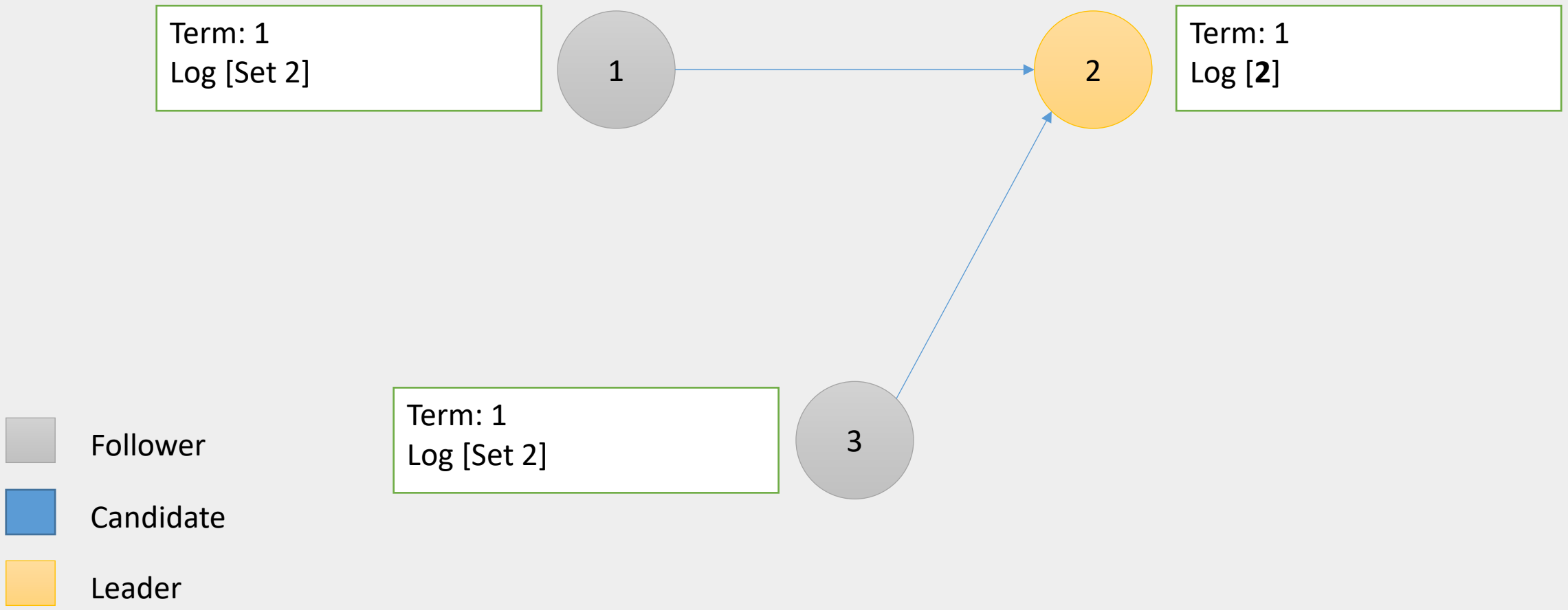
Candidate

Leader

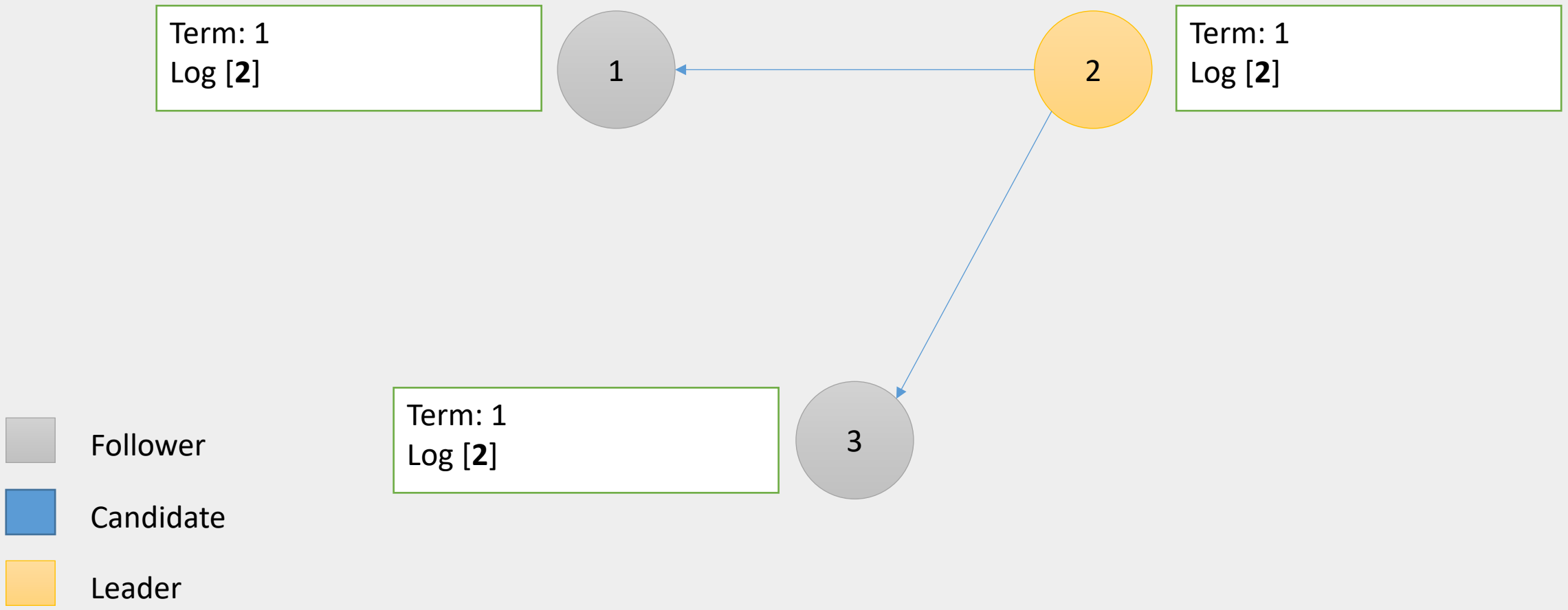
Raft leader election



Raft leader election

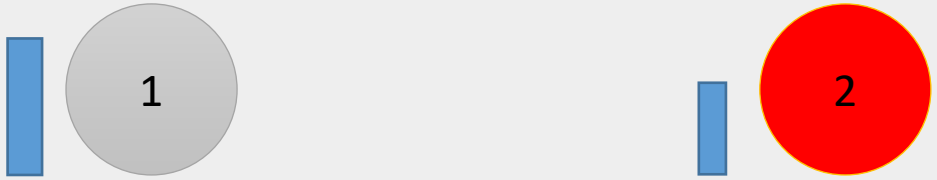


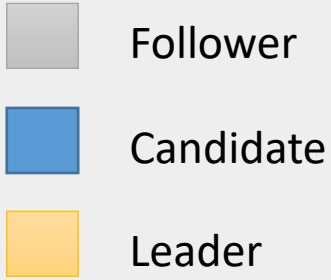
Raft leader election



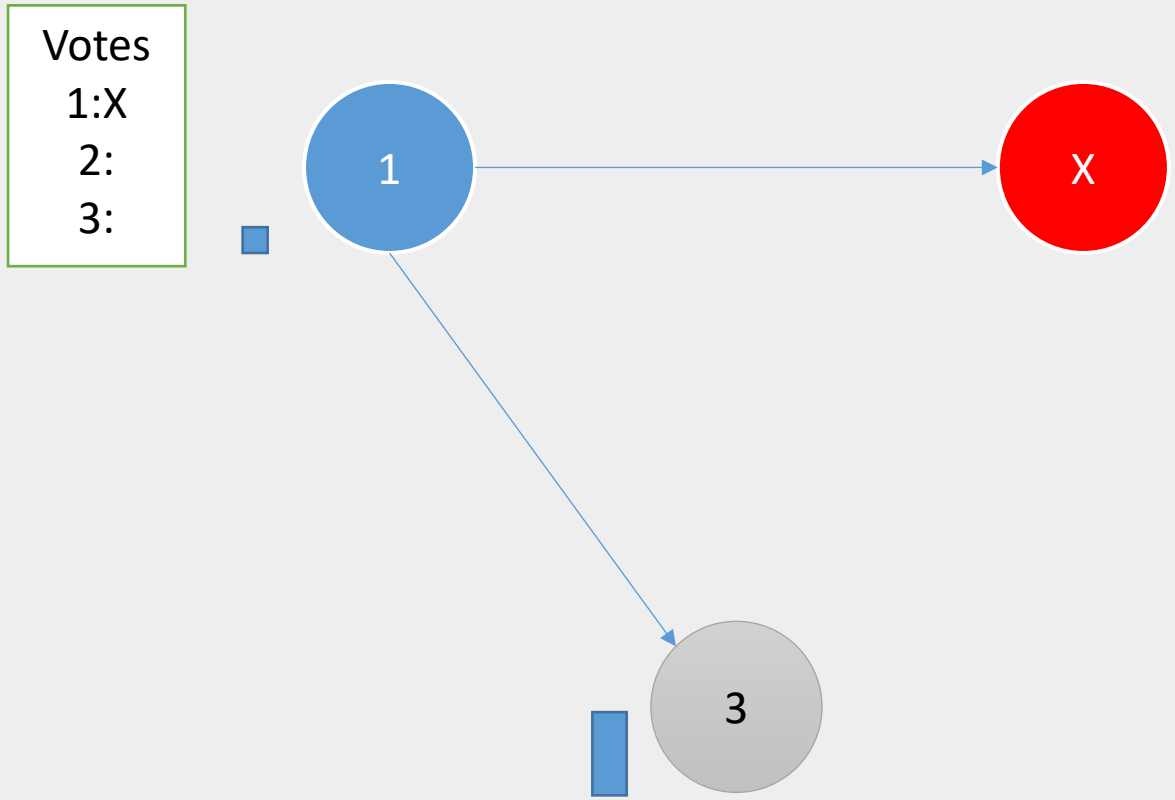
Failure

Raft leader failure





Raft leader failure

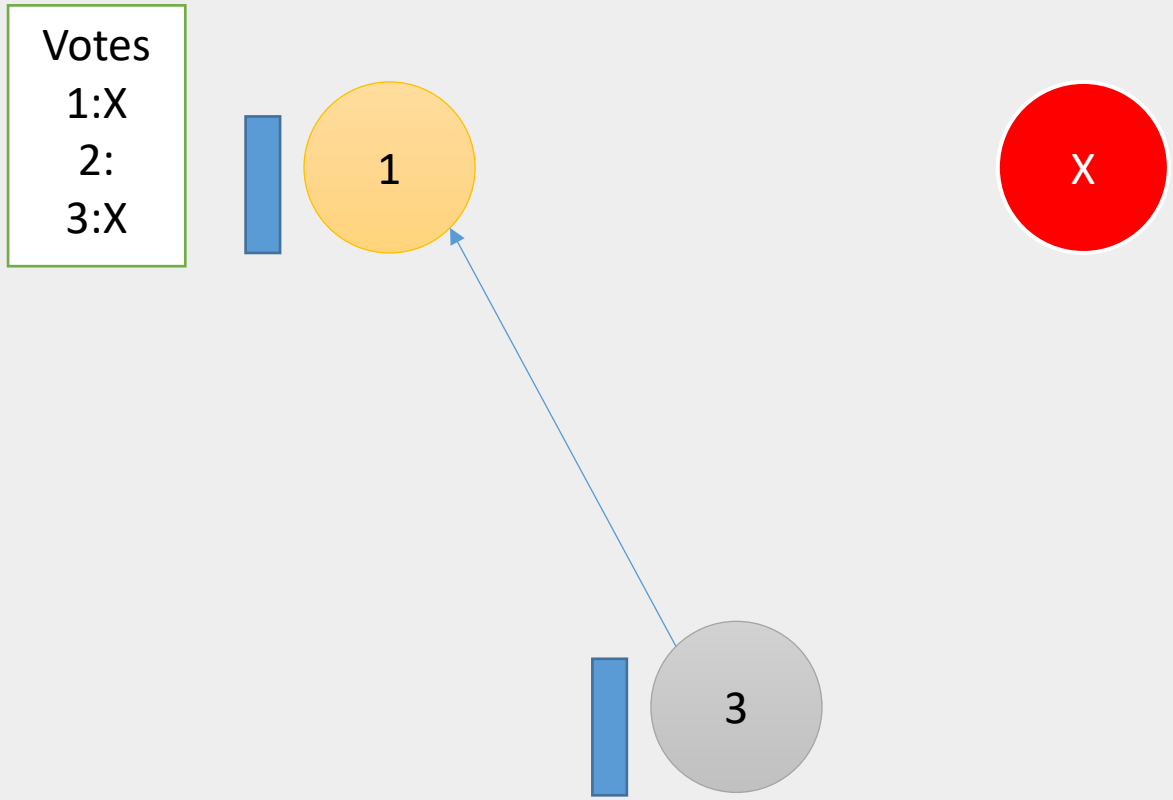


Follower

Candidate

Leader

Raft leader failure

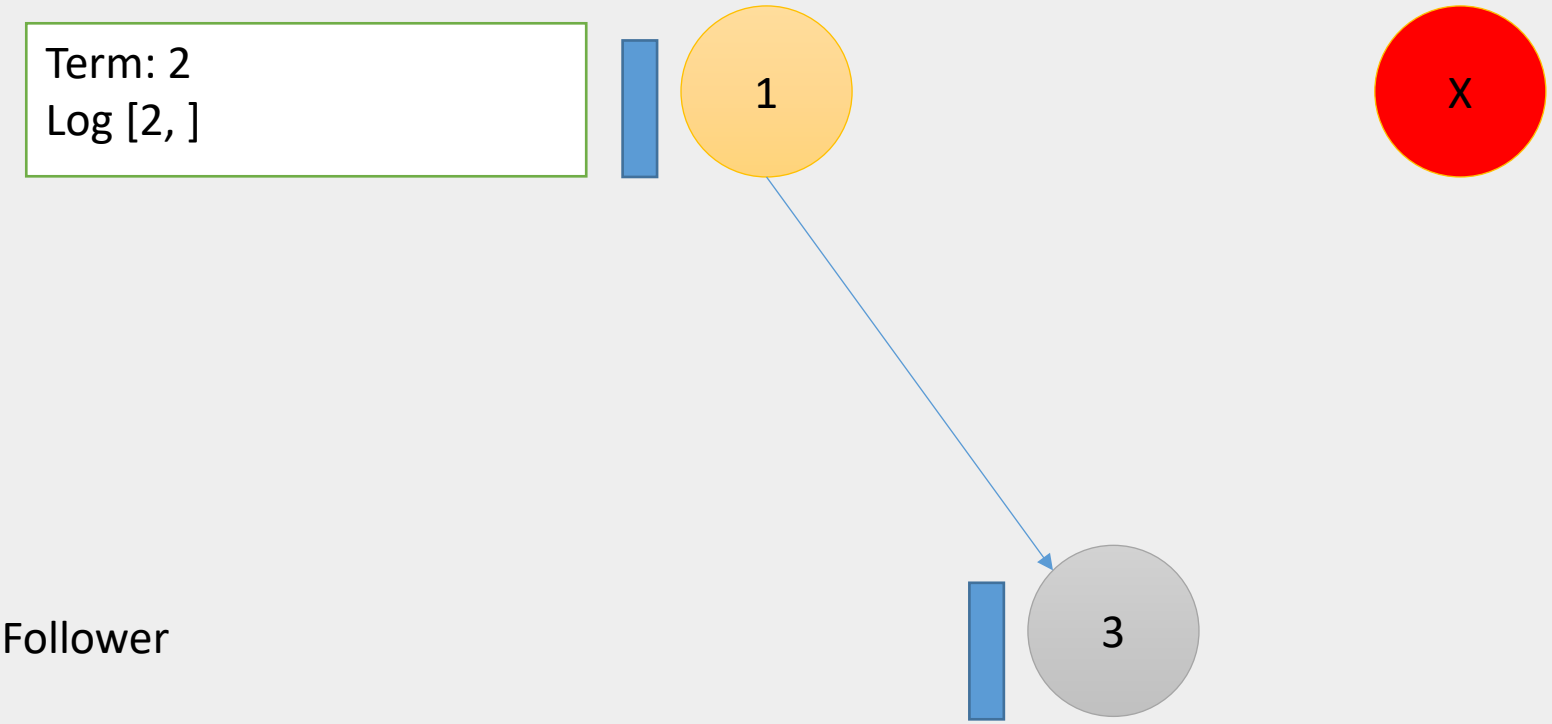


Follower

Candidate

Leader

Raft leader failure



Challenges

- Spilt vote: just wait for the next timeout
  - Hopefully random timeouts will avoid it happening again
- Network partitions: no problems as we need a quorum to update state

Is Raft Byzantine fault tolerant?

- No
- There are extensions that are byzantine fault tolerant

Online examples

- https://raft.github.io/
- http://thesecretlivesofdata.com/raft/

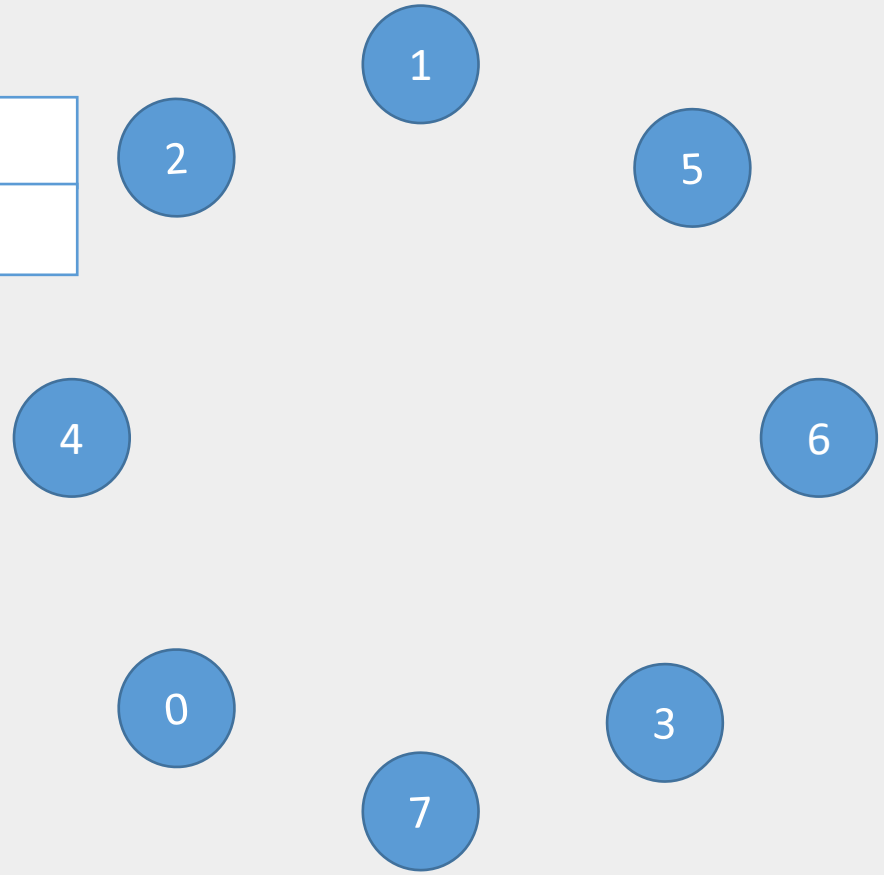
Summary

- Understandable distributed consensus algorithm
  - Few states, simple transition conditions, simple rules
- Addresses replicated state machine problem
- Strong leader model
  - Several pros and cons
- Safety:
  - Leader election/log commitment rules guarantee safety
- Liveness: competing candidates could cause repeated split vote
  - Mitigated by random timeouts (rare in practice)
- Not designed to address Byzantine failures

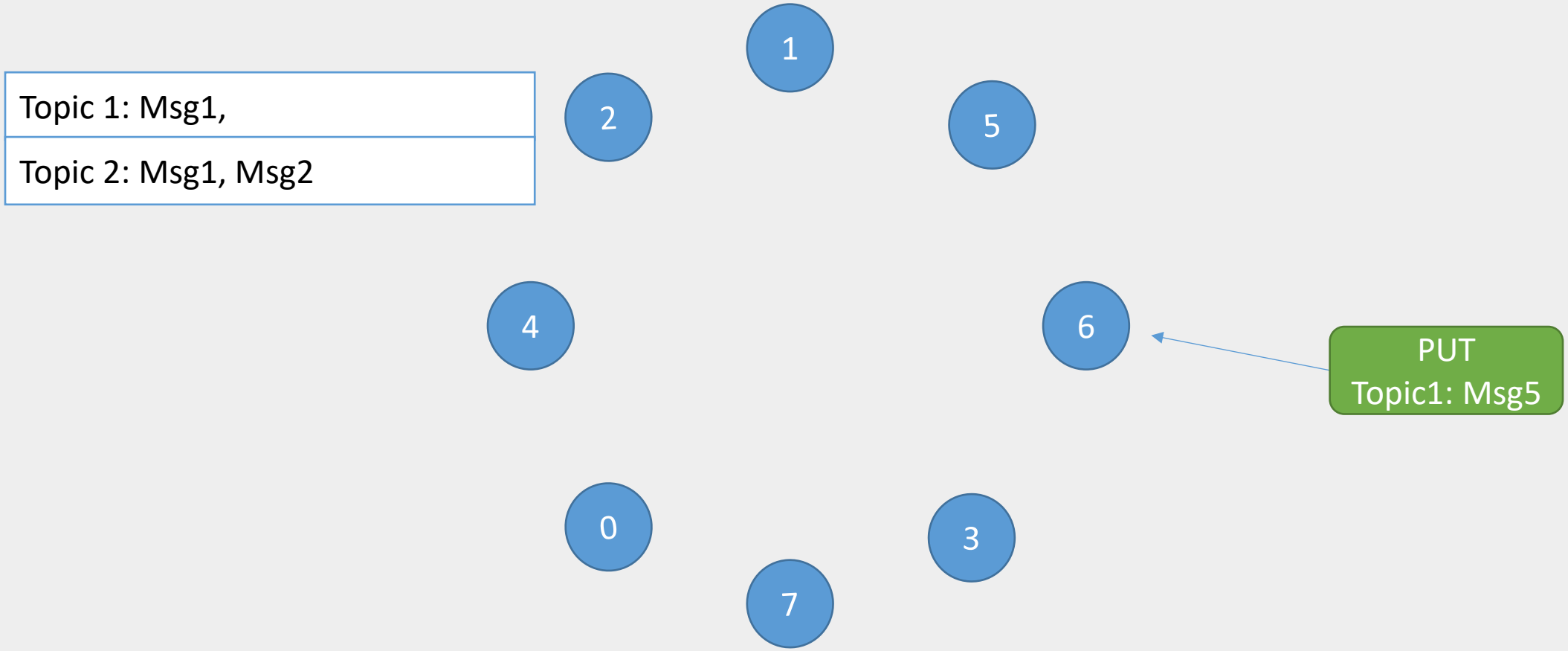
Project
Raft REST Message Queue

Distributed message queue

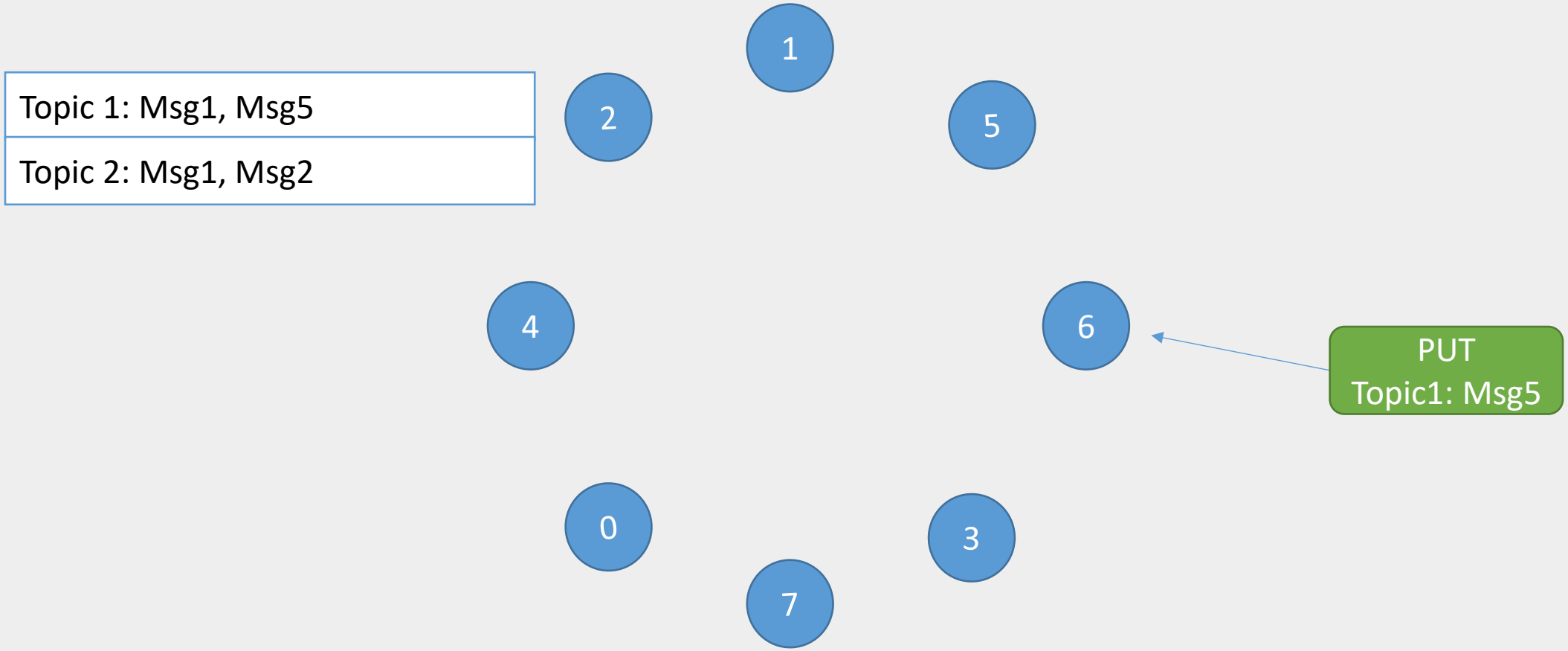
Topic 1: Msg1,
Topic 2: Msg1, Msg2,



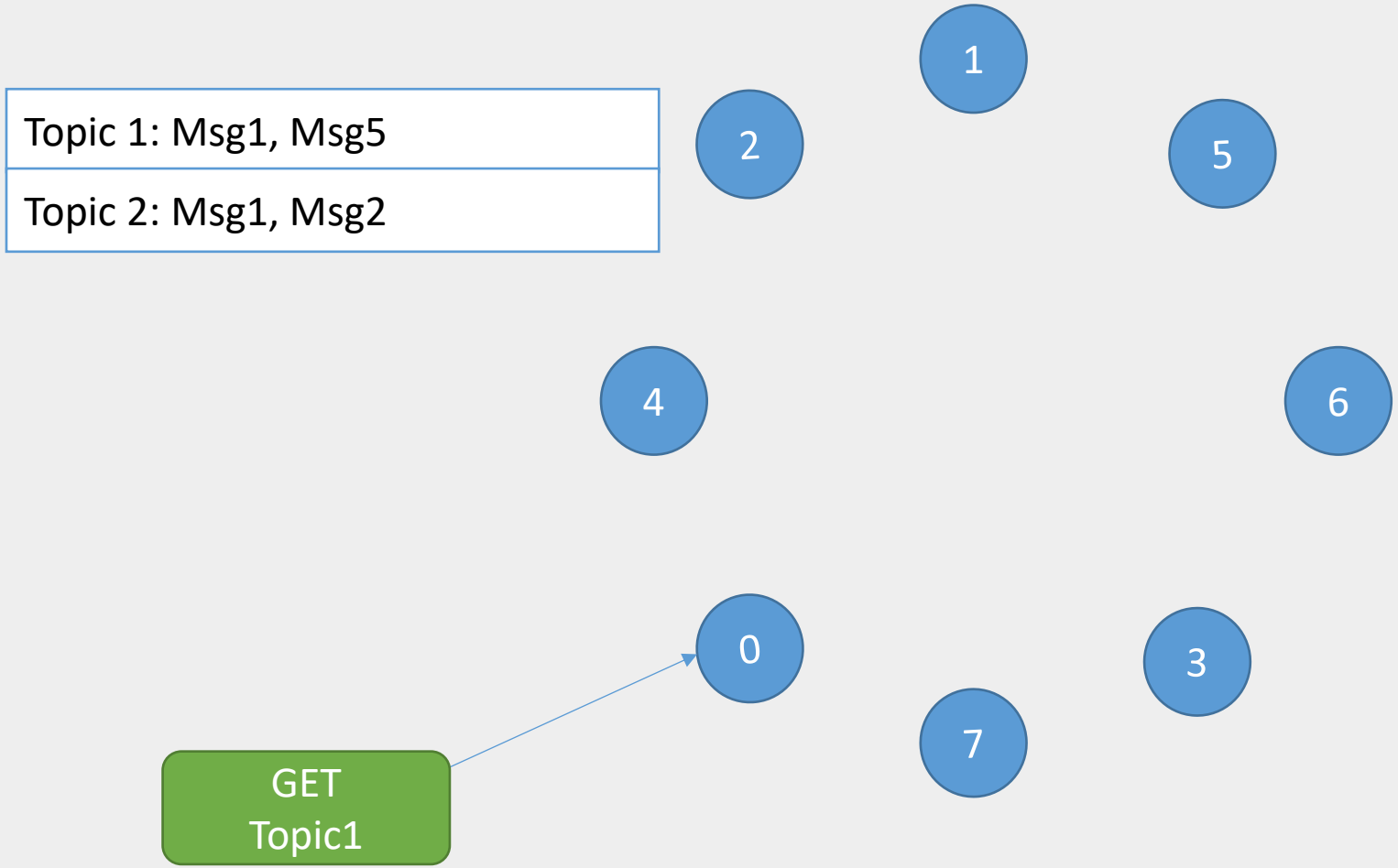
Distributed message queue



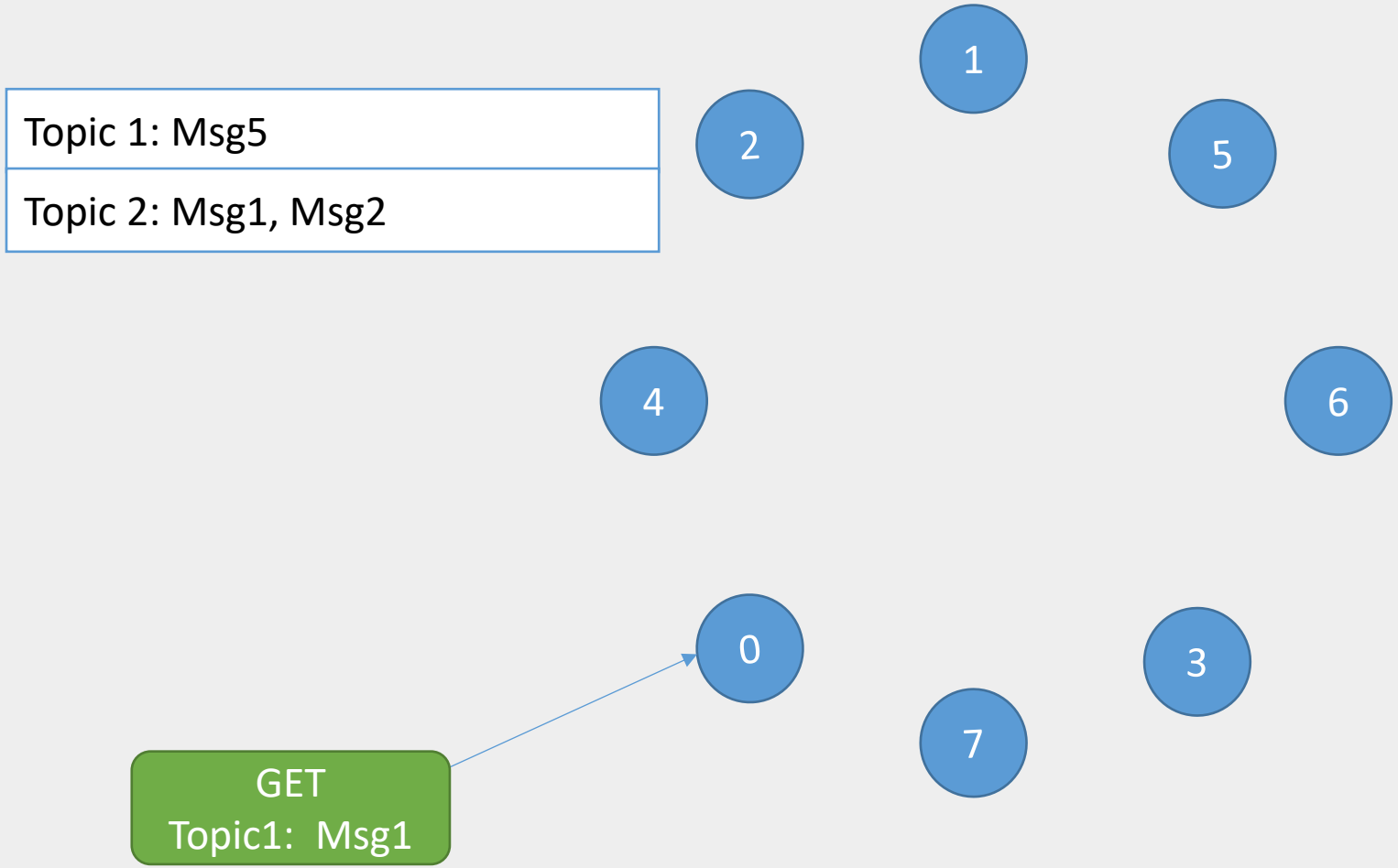
Distributed message queue



Distributed message queue



Distributed message queue



REST Interface

- PUT(topic, message) the message will be appended to the end of the topic's message queue
- GET(topic) will pop the first message from the topic's queue and return the message

Goal

- Implement a distributed, replicated, fault-tolerant message queue
- You will need to ensure that the message queue is distributed across several nodes (or on different ports) and is fault tolerant …

→Raft consensus algorithm

Requirements

- Python3
- REST API
- JSON-based interface for put/get
- Tolerate up to N/2-1 failures: always have a quorum alive
- No use of existing implementations (e.g., consensus algorithms, DHTs) or key-value stores (e.g., Redis) or message brokers

Approach

1. Central message queue
2. Election algorithm
3. Distributed replication (fault tolerance)

.. Lots of testing

- March 9: Report (submission date)

Example

- https://github.com/mpcs-52040/raft_template

Questions?