Paxos

Anjali Pal and Elijah Greisz

Distributed State

- We want to have a consistent view of state across a distributed system
- Need some way for multiple “agents” to agree on such a state
- Made harder by the presence of failure:
    - **Byzantine faults:** machines can do the “wrong” thing
        - Paxos assumes non-Byzantine
    - Agents can stop or restart arbitrarily
        - Requires that some information is remembered – “stable storage”
    - Messages have no upper bound on delivery time and can be dropped or duplicated
        - Paxos assumes that messages are not corrupted

Consensus Algorithm

Goal: A set of processes agree on a value

Safety:

- Only a value that has been proposed may be chosen
- Only a single value is chosen
- A process never learns that a value has been chosen unless it actually has been.

Liveness:

- Some value is eventually chosen
- If a value has been chosen, any process can eventually learn the value

Three Agent Classes

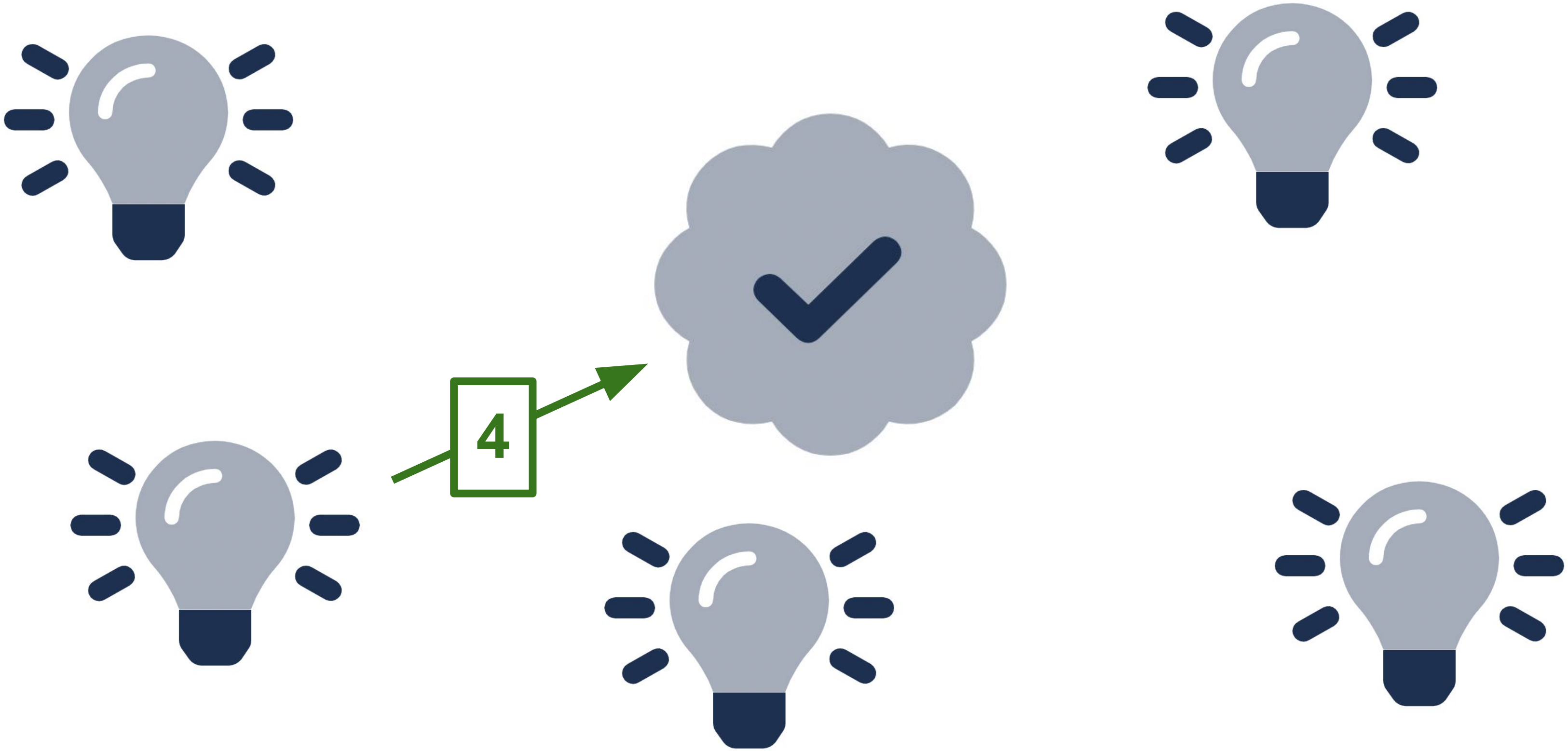
Proposers

Acceptors

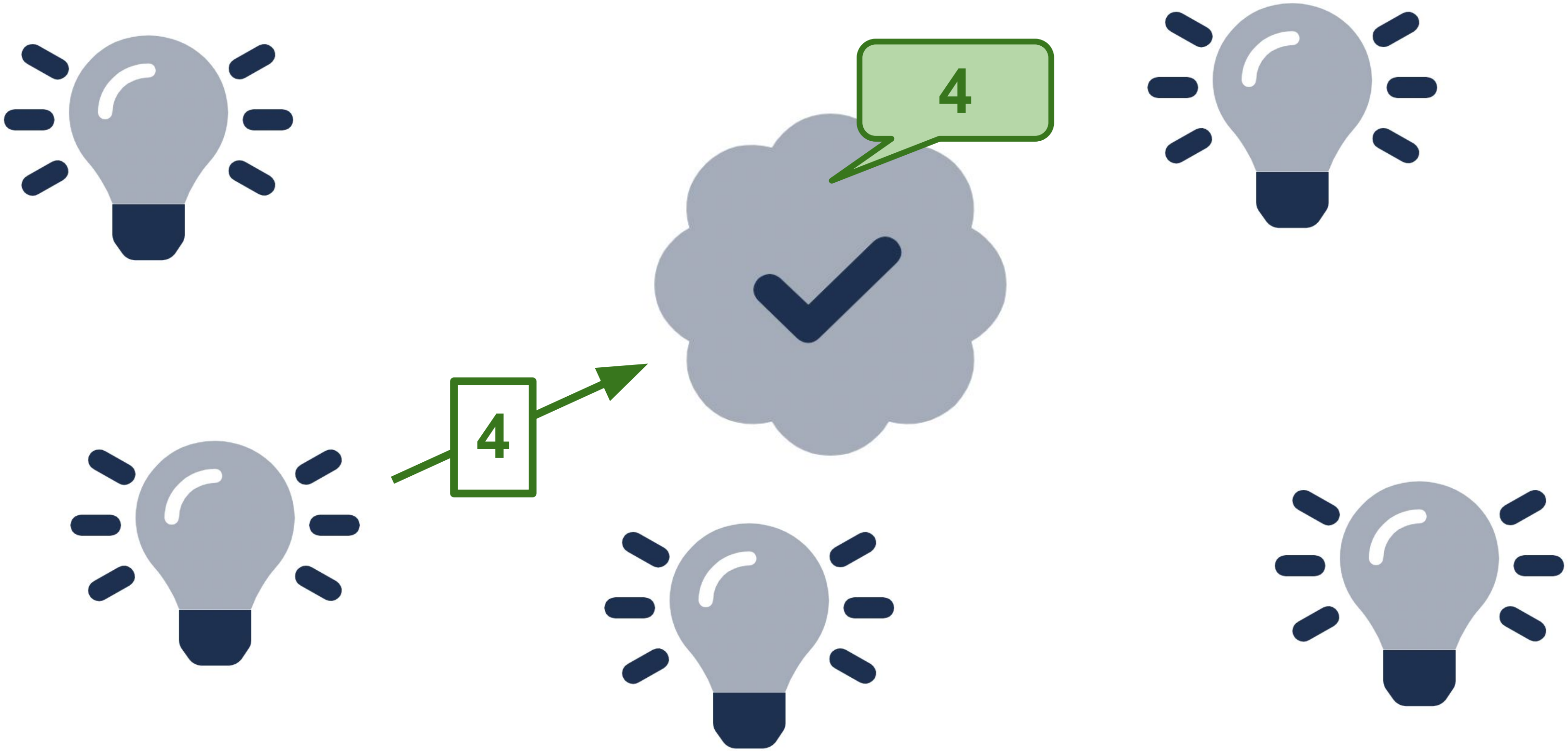
Learners

First Attempt: Single Acceptor

First Attempt: Single Acceptor

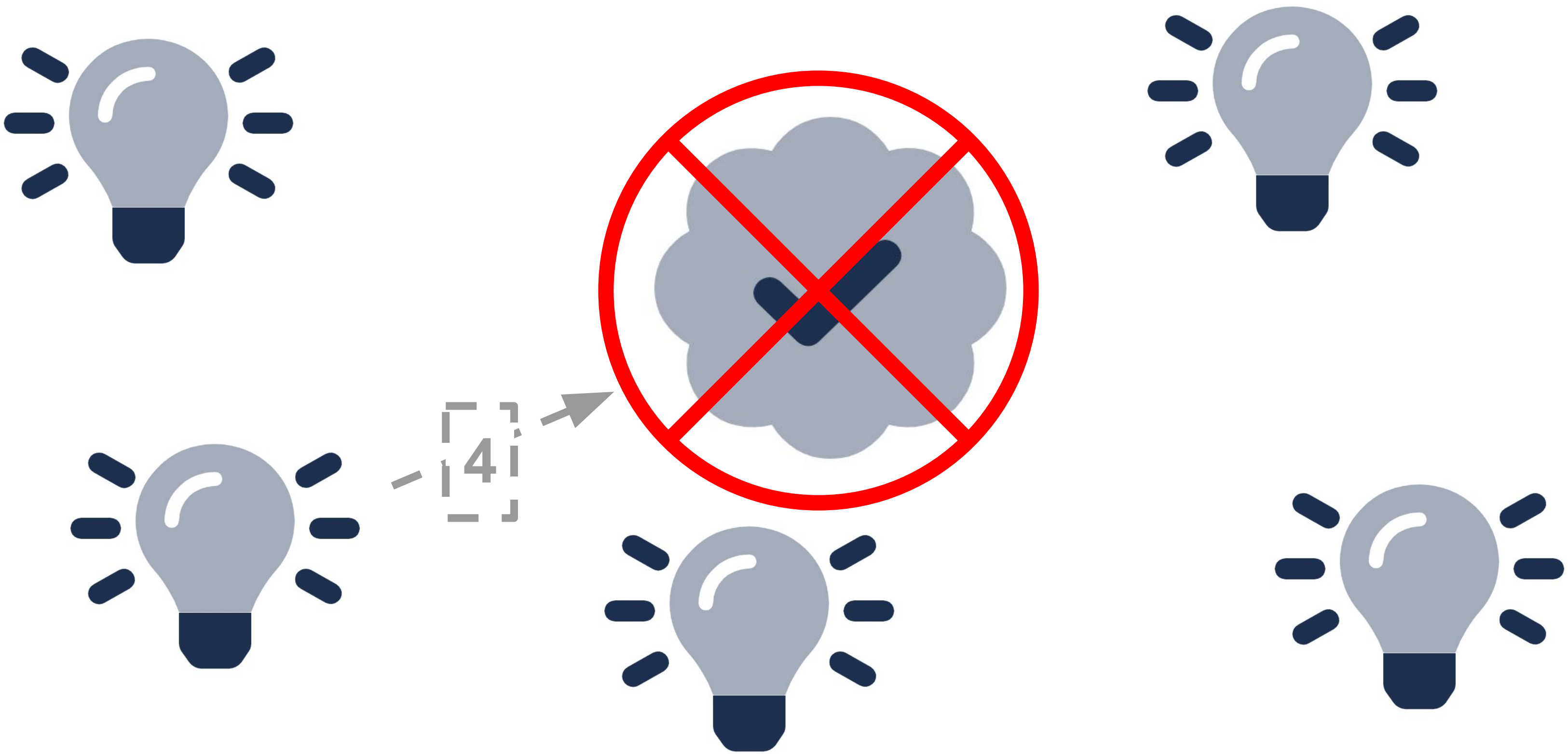


First Attempt: Single Acceptor

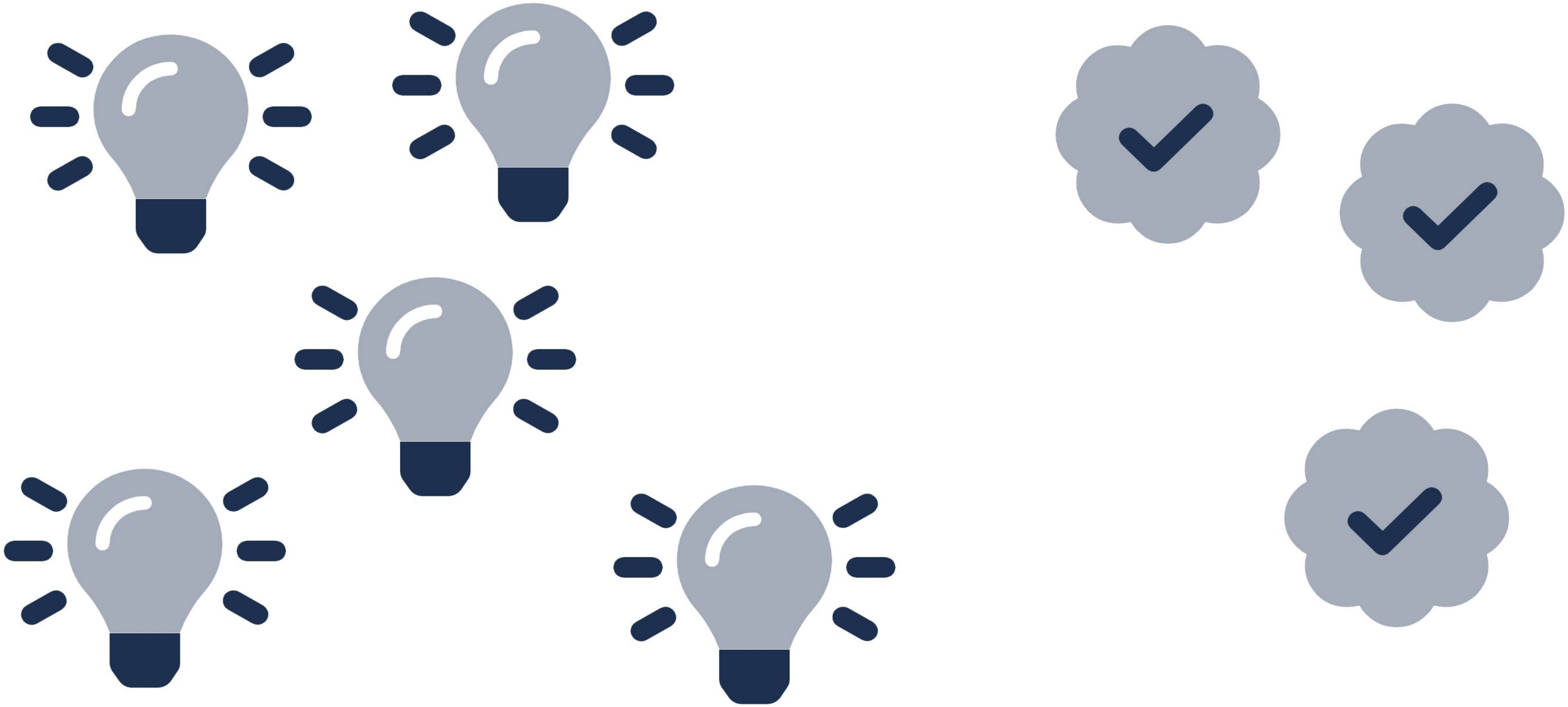


What goes wrong?

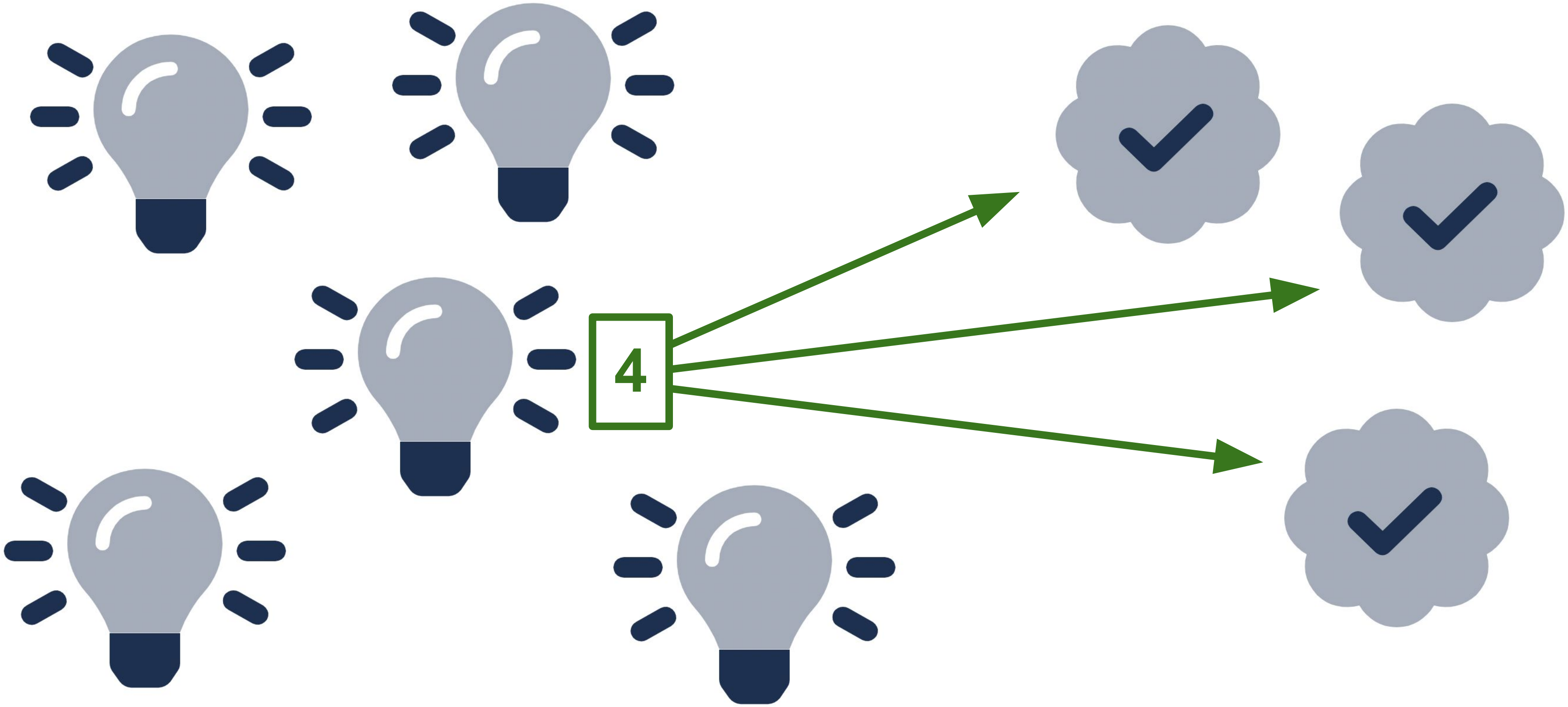
What goes wrong?



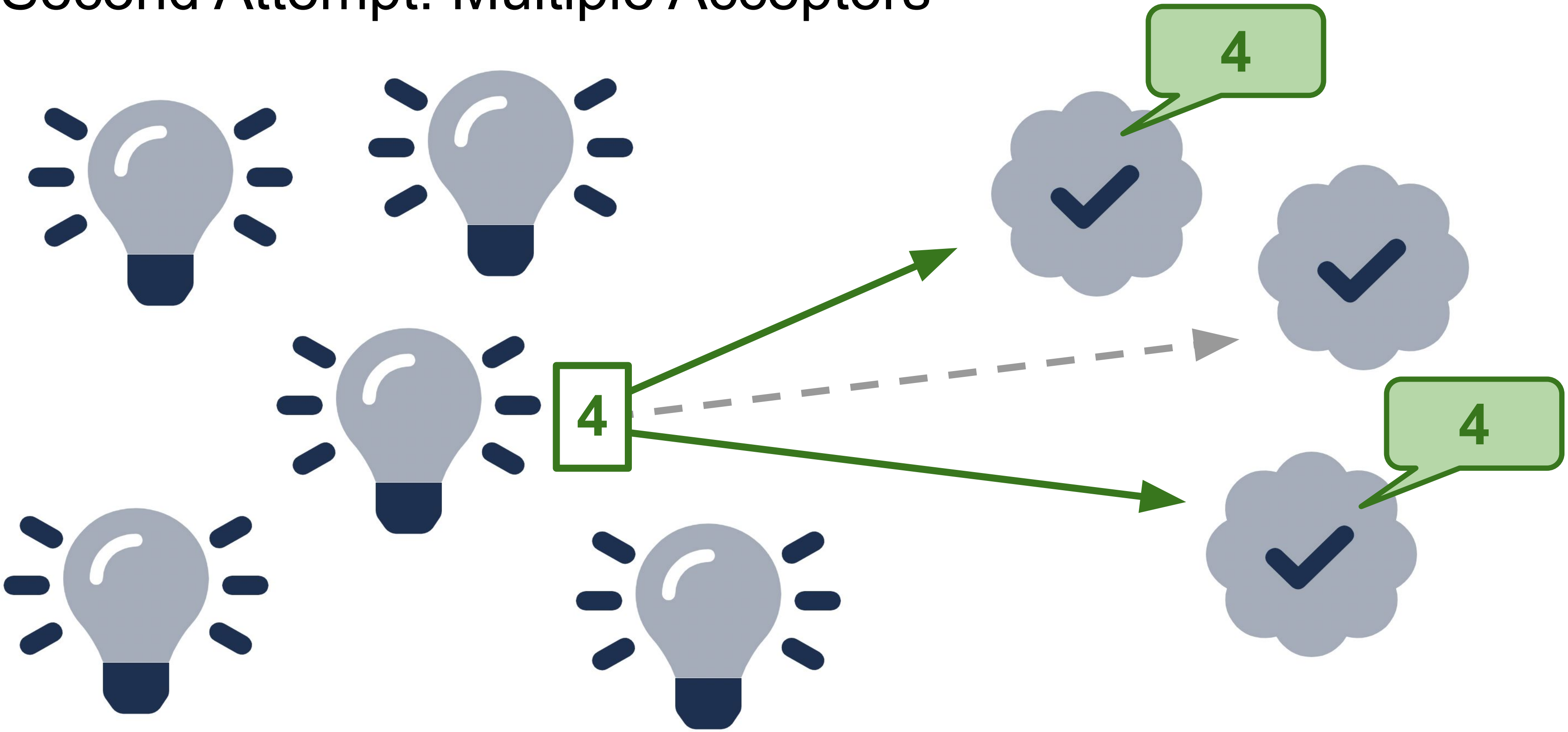
Second Attempt: Multiple Acceptors



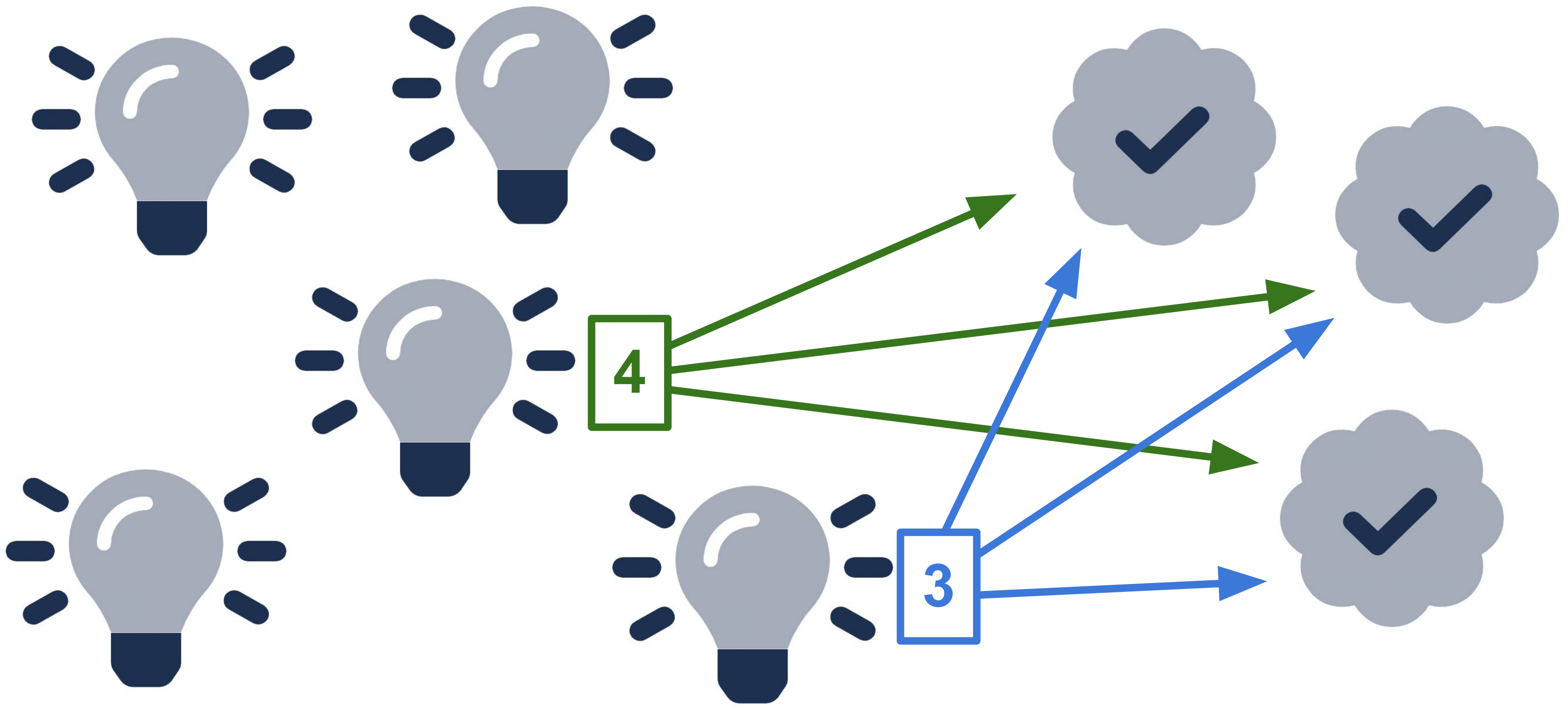
Second Attempt: Multiple Acceptors



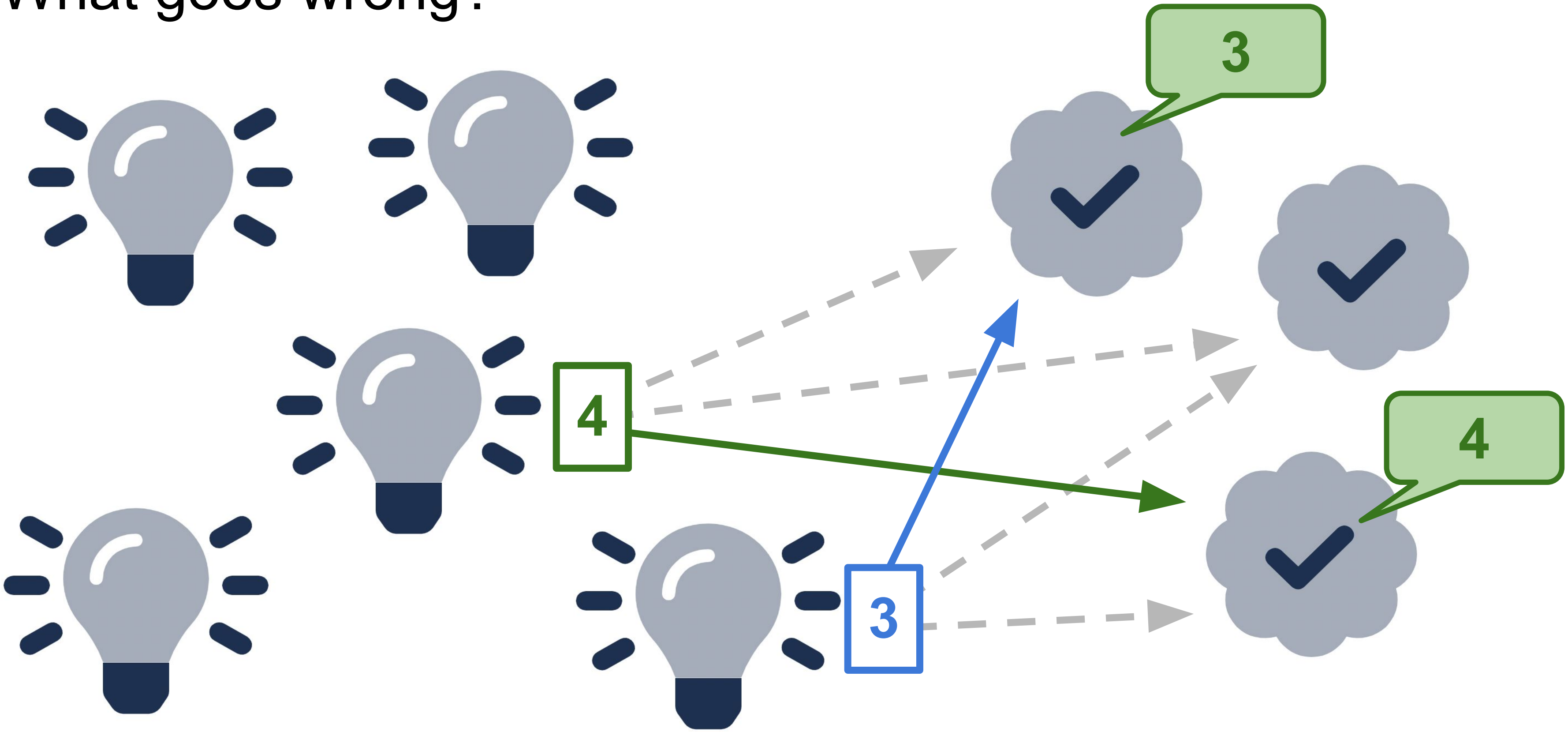
Second Attempt: Multiple Acceptors



What goes wrong?



What goes wrong?



Phase 1

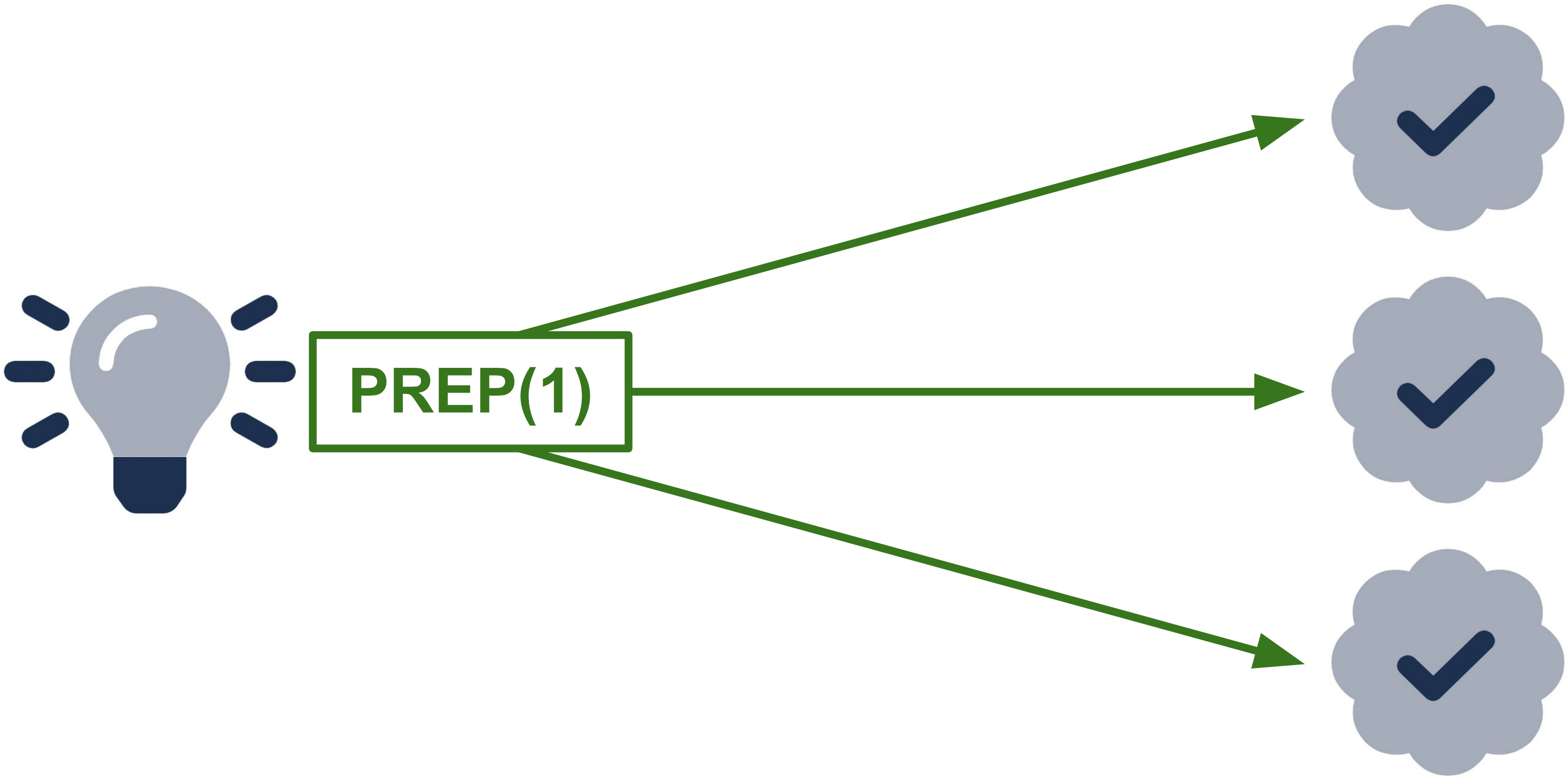
- Proposer sends PREPARE(n) to a majority of acceptors

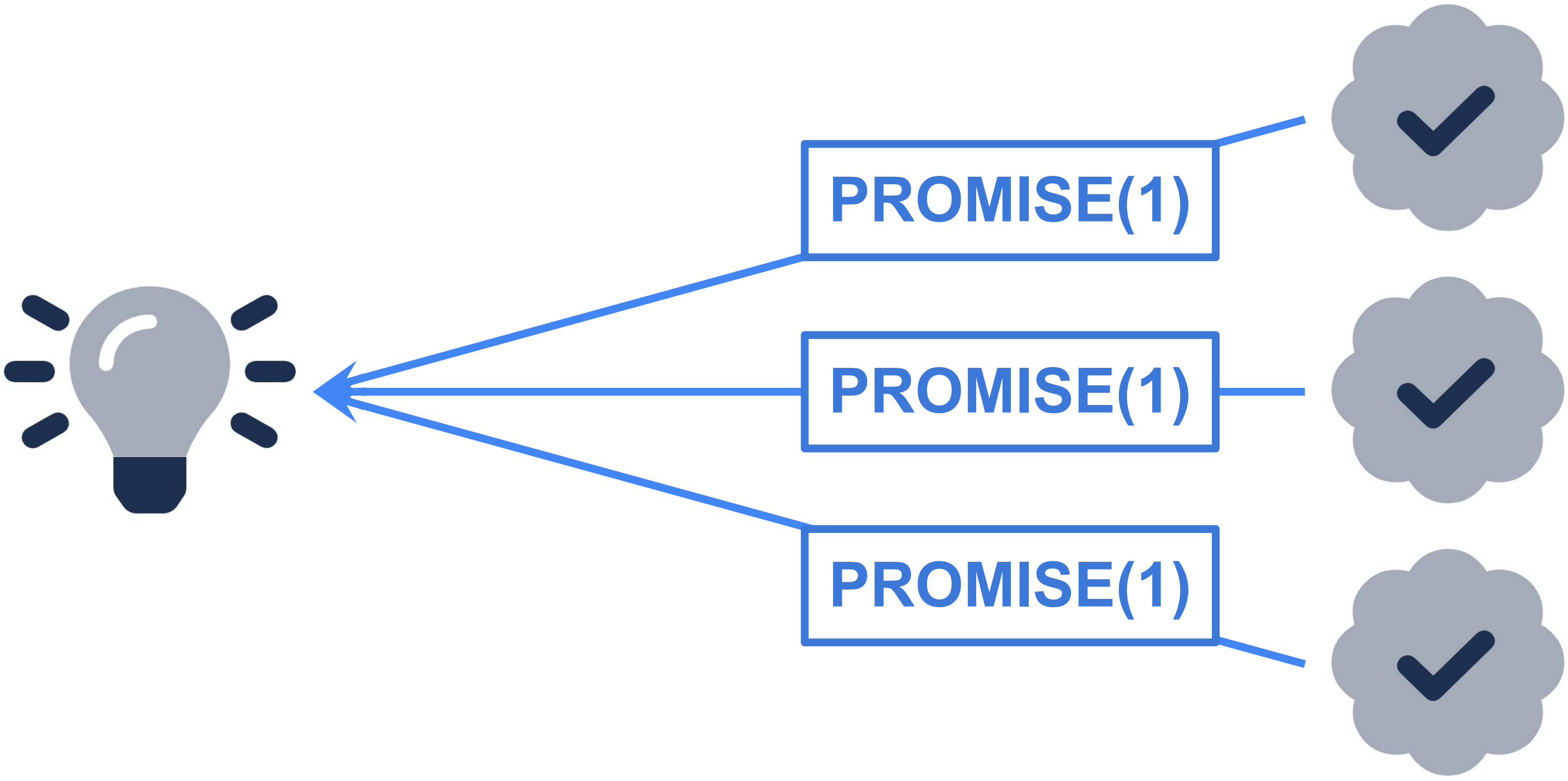
- If acceptor receives PREPARE(n) with *n* greater than any previous PREPARE requests, it responds PROMISE(n, p) indicating it will not accept any proposals <*n* and *p* is the highest-numbered proposal it has previously accepted

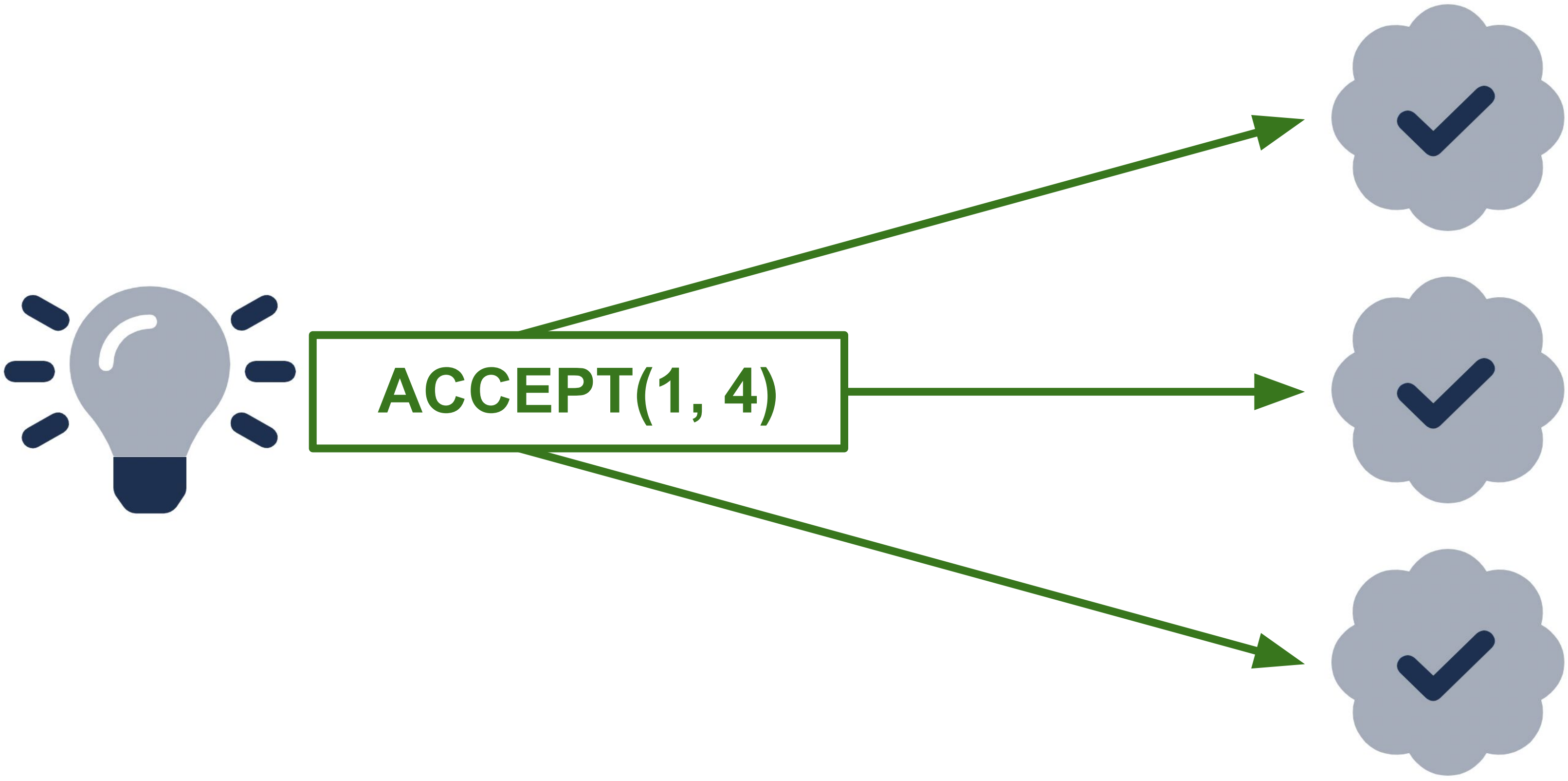
Phase 2

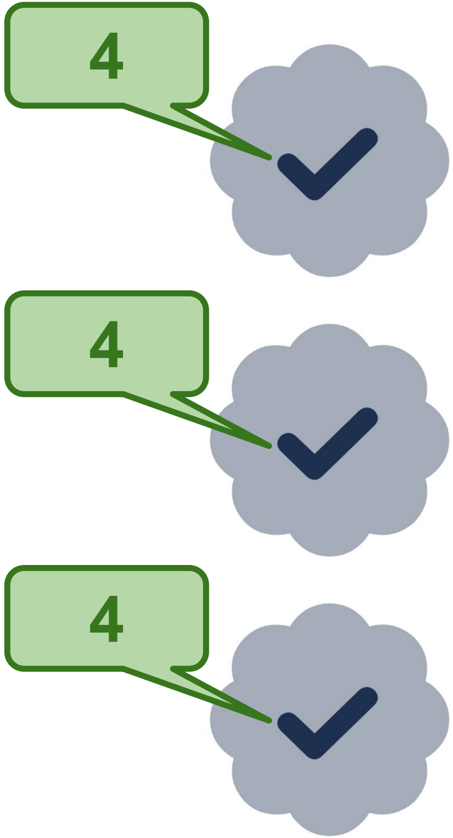
- If the proposer receives a response to its PREPARE(n) requests from a majority of acceptors, then it sends ACCEPT(n, v) to each acceptor, where n is the proposal number and v is the value of the highest-numbered proposal among the responses (or any value if none of the acceptors had previous proposals)

- If an acceptor receives an ACCEPT(n, v) request, it accepts the proposal unless it has already responded to a PREPARE request with a number greater than *n*
    - *chosen* only if a majority of acceptors accept


PREP(1)


PROMISE(1)
PROMISE(1)
PROMISE(1)


ACCEPT(1, 4)


4
4
4

What happens if a proposer fails?

1. Before PREPARE
2. After PREPARE but before ACCEPT
3. After ACCEPT

What happens if an acceptor fails?

1. Before PREPARE
2. After PREPARE but before ACCEPT
3. After ACCEPT

Learners

- Each acceptor could send a message to each learner when it accepts a proposal
    - Requires |A| * |L| responses
- Each acceptor could send a message to a single learner when it accepts a proposal, who then informs the other learners when a value is chosen
    - Requires an extra round for all the learners to discover the chosen value
    - If the distinguished learner fails, the rest of the learners will not learn the message
    - Requires |A| + |L| responses
- Acceptors could send messages to a set of learners
- Learners could ask acceptors what proposal have been accepted
- Learners can have proposers issue a proposal to determine whether a value has already been chosen

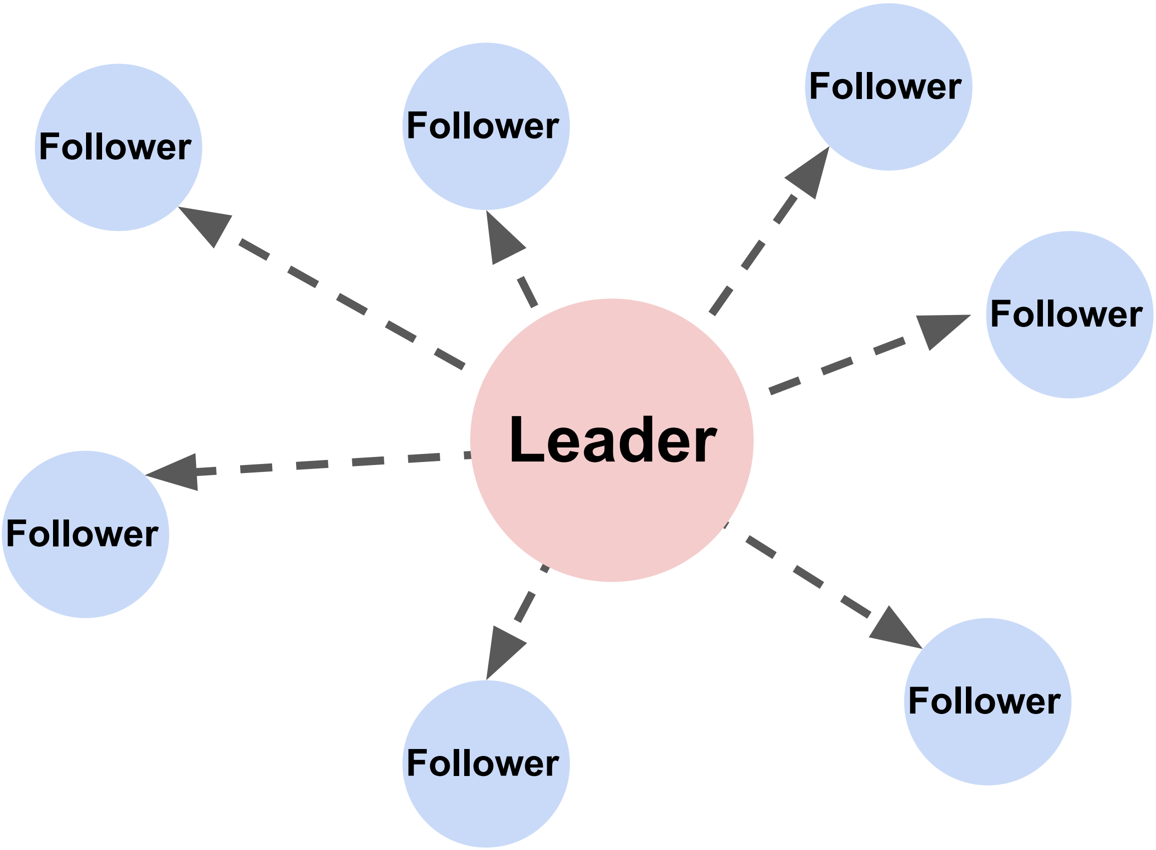
Using Paxos to Implement a State Machine

- The server is a deterministic state machine
- Clients can issue commands
- An implementation with a single central server would fail if that server ever failed, so use a collection of servers that independently implement the state machine
- All the servers will produce the same sequences of states/outputs if they execute the same sequence of commands, so we just need to guarantee that the servers all agree on the command sequence
- Use Paxos repeatedly to choose the next command to execute
    - i.e. 1 Paxos instance per command “slot”
- Each server is a proposer, acceptor, and learner
- Usually elect a single leader for efficiency, but it is not strictly required for safety

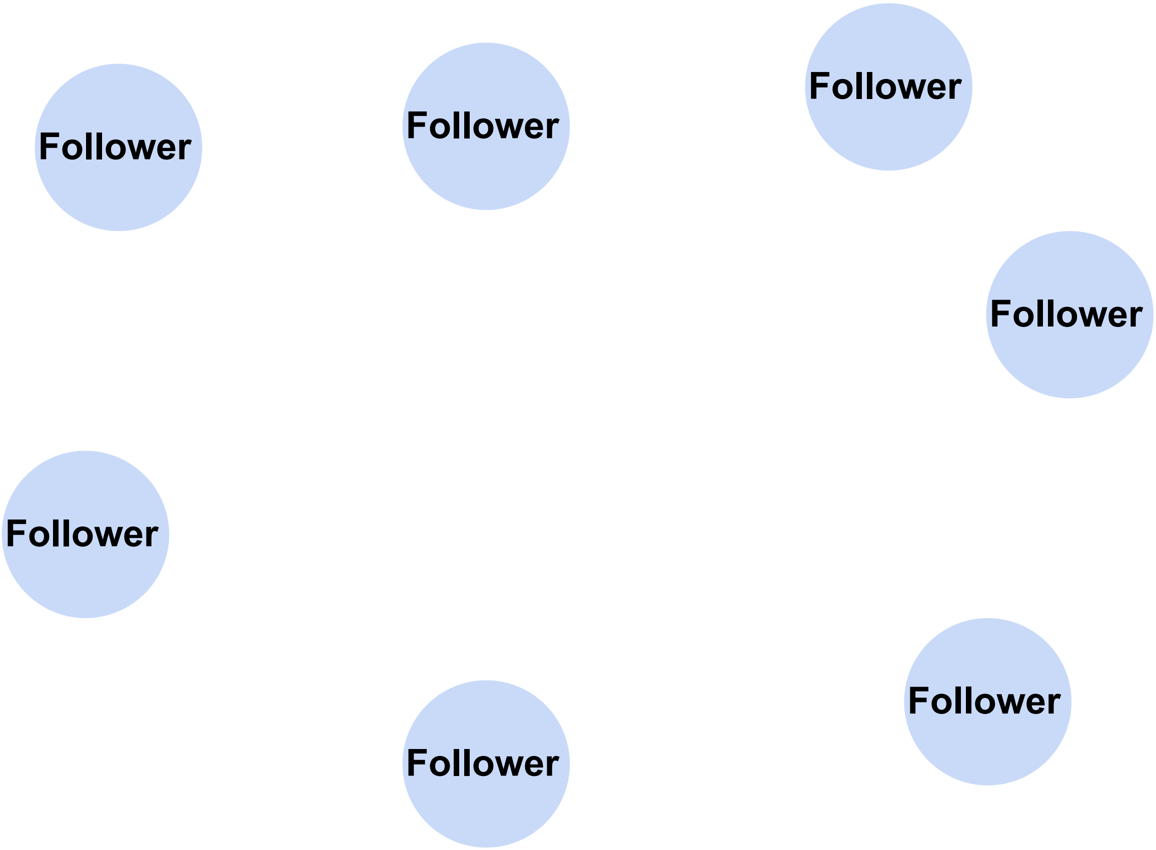
Progress in Paxos

- Everything before this guarantees safety
- But it doesn’t guarantee progress:
    - Two proposers could have an interleaving where they keep proposing higher numbers
- How do we fix this?
    - Can use a *distinguished proposer* – this is the only proposer that tries to make proposals.
    - Need an election process to determine this distinguished proposer
    - In their implementation, same process is the *distinguished proposer* and *distinguished learner.*

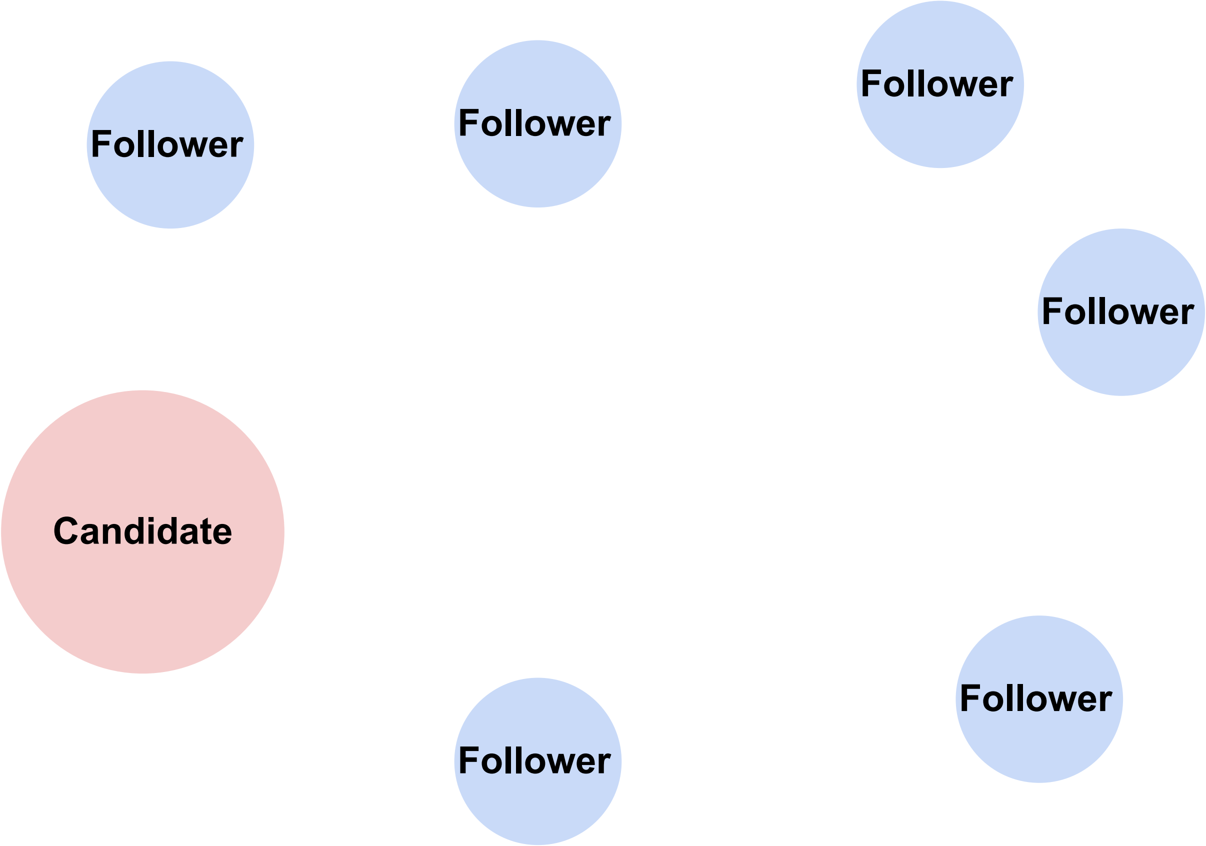
Raft


Follower
Follower
Follower
Follower
Leader
Follower
Follower
Follower

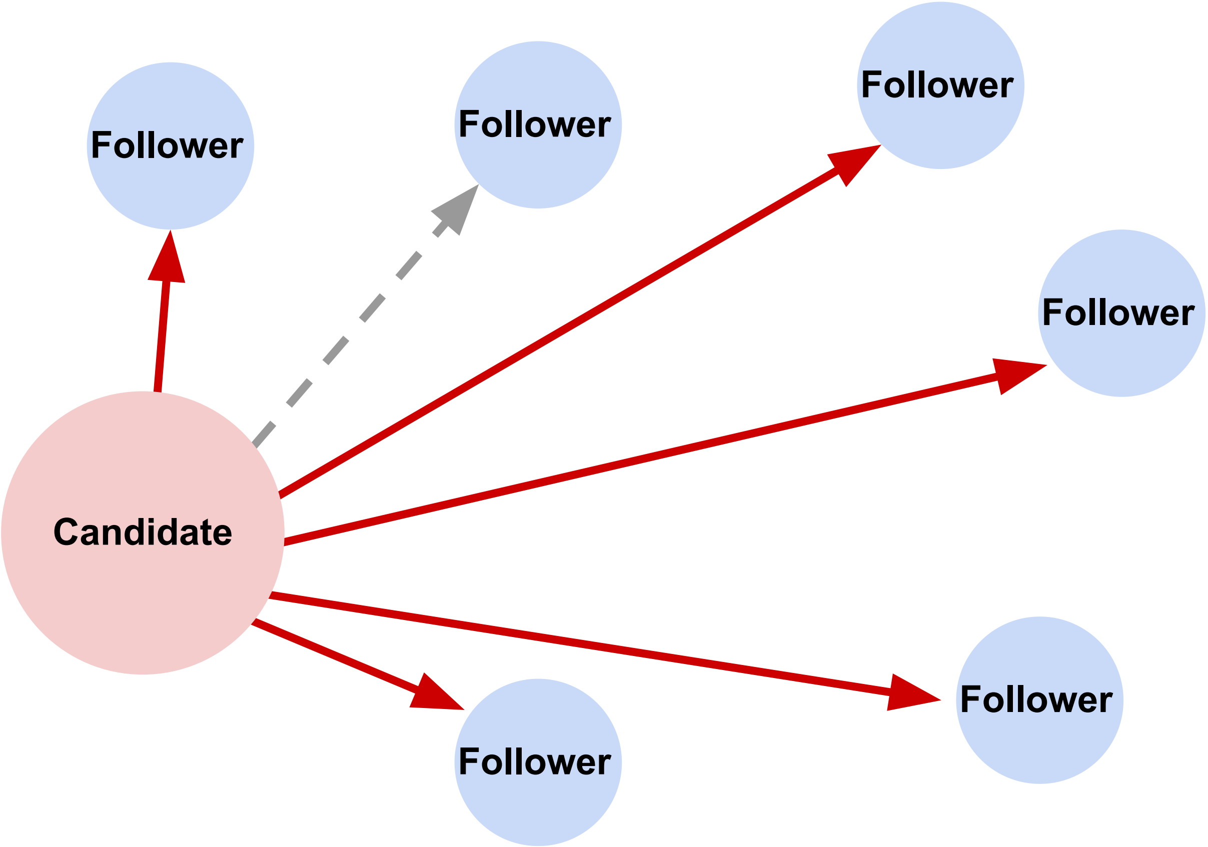
Raft


Follower
Follower
Follower
Follower
Follower
Follower
Follower

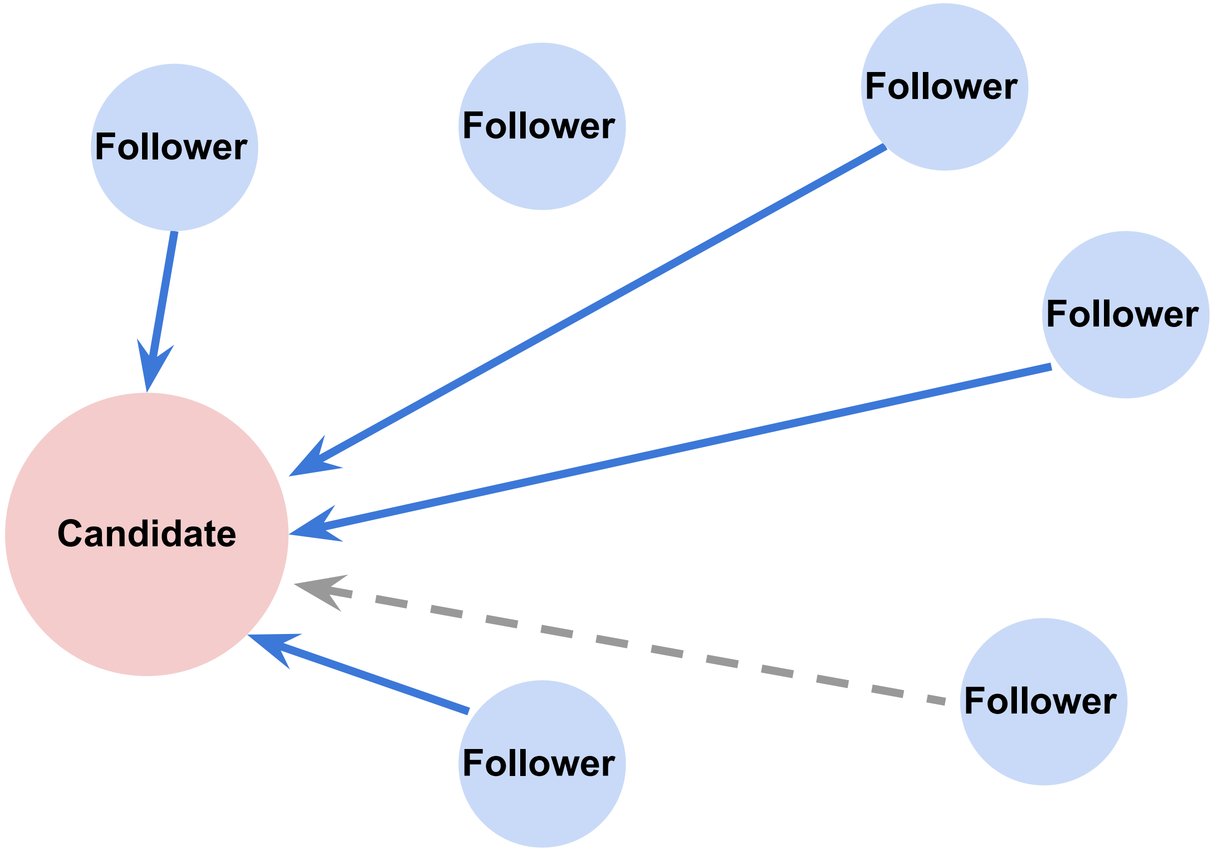
Raft


Follower
Follower
Follower
Follower
Candidate
Follower
Follower

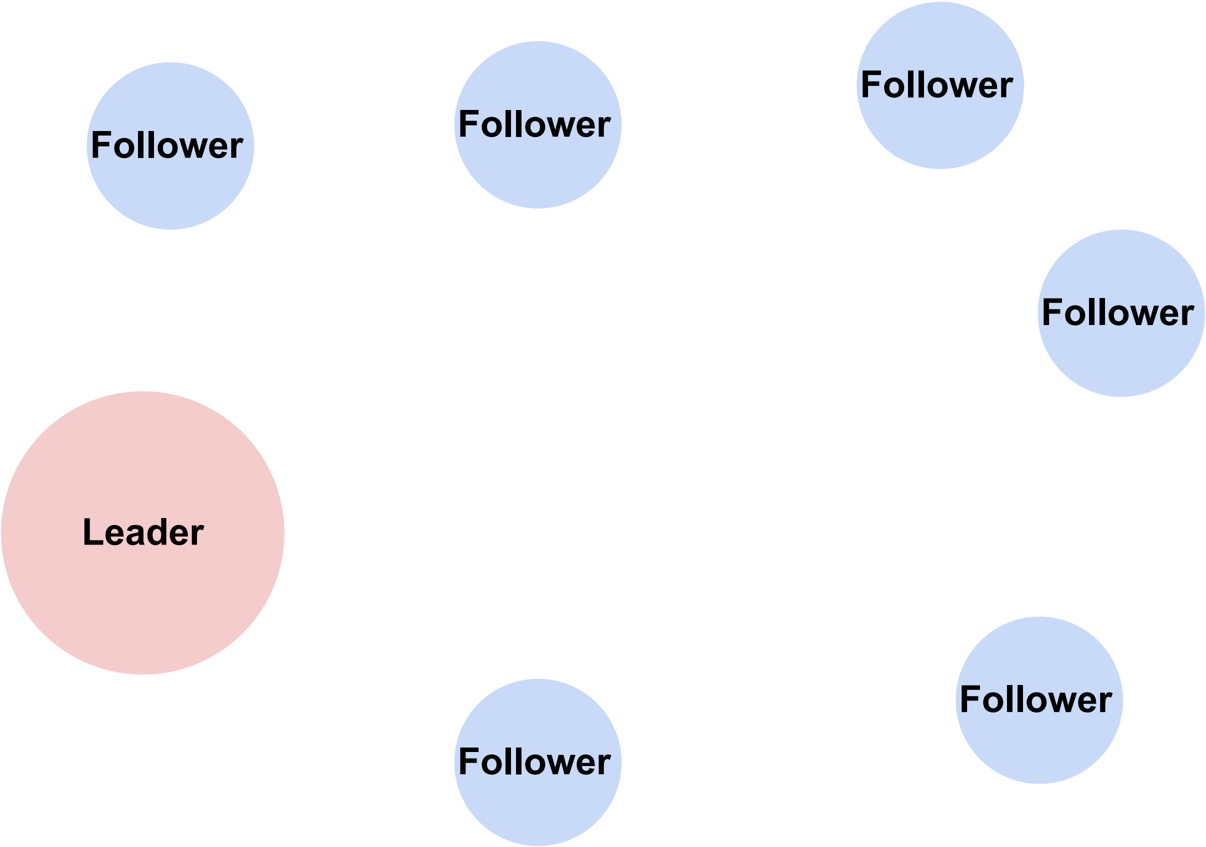
Raft


Follower
Follower
Follower
Follower
Candidate
Follower
Follower

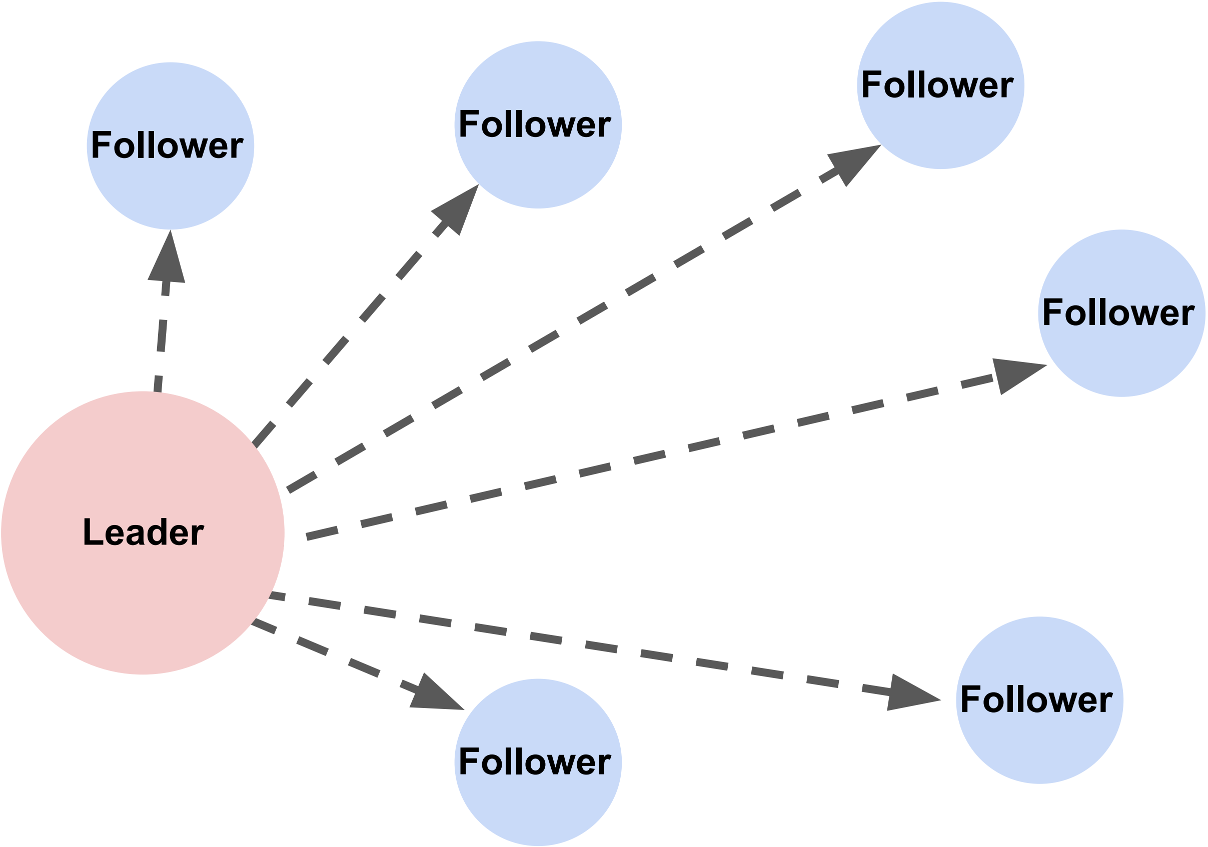
Raft


Follower
Follower
Follower
Follower
Candidate
Follower
Follower

Raft

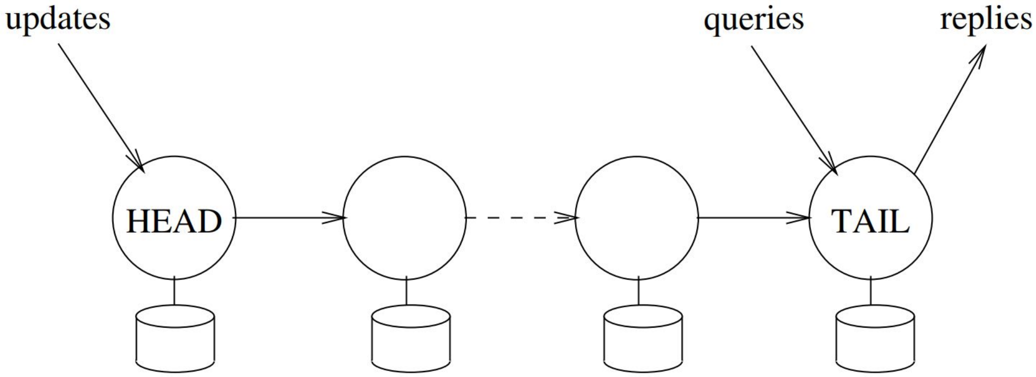

Follower
Follower
Follower
Follower
Leader
Follower
Follower

Raft


Follower
Follower
Follower
Follower
Leader
Follower
Follower

Chain Replication

- Queries go straight to the tail, updates to the head
- *master* detects failures and updates configurations
  - Actually implemented via Paxos
- Assumes fail-stop



What does this do well?

What does this not do well?

Fault-Tolerant Virtual Machines

- If a VM fails, we want to be able to transition to a backup without the client noticing
- How might we keep a backup up to date?

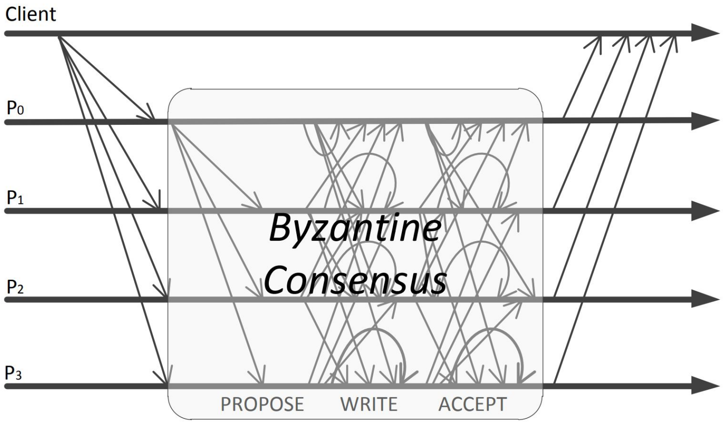
Two possibilities:

- Send all state (CPU, memory, etc.) to backup - too much bandwidth!
- Send only input requests
  - But then must deal with non-deterministic operations
  - ...but this can be dealt with via the hypervisor

- Less than 10% overhead and less than 20 Mb/s bandwidth required
- Limited to uniprocessors

SMART

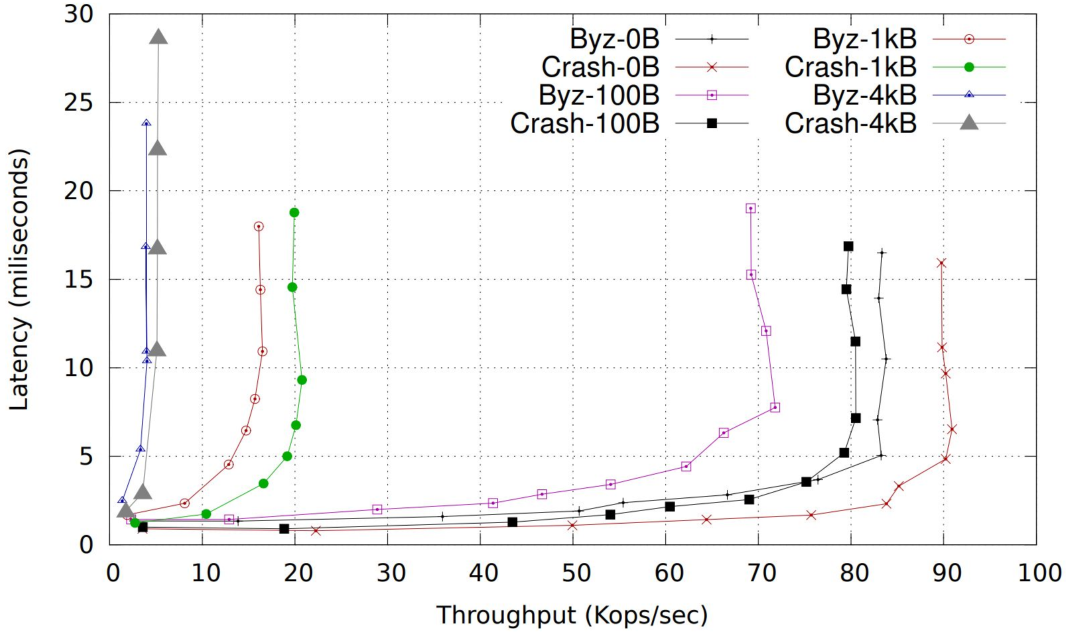
- **Byzantine Fault Tolerant State Machine Replication** (BFT SMR): state system that can tolerate Byzantine faults
- SMART: Java implementation, can prevent non-malicious Byzantine faults
  - Corrupted messages, abnormal processes
- Three steps:



- *PROPOSE* proposes a batch of requests
- *WRITE* and *ACCEPT* use cryptographic hash of batch
- When certain faults occur, begins *synchronization phase* (leader election, state transfer, etc.)

SMART (cont.)

- Can enable “crash fault-tolerant” (CFT) mode
    - No longer protects against Byzantine faults
    - Removes the *WRITE* step from before



- BFT mode increases latency, but less than you might think

2 Phase Commit

- Coordinator and cohorts
- First Phase:
  - Coordinator sends “prepare” message to each cohort
  - Cohorts respond with either “commit-vote” or “abort-vote”
- Second Phase:
  - If all cohorts responded with “commit-vote”, the coordinator sends “commit”. If any cohorts responded with “abort-vote”, the coordinator sends “abort”
  - Cohorts respond with acknowledgement
- Blocking protocol: low availability, no progress if a cohort is down