Parallel and Distributed Computing
CS3006 (BCS-6C/6D)
Lecture 27 (26)

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

09 May, 2023

Previous Lecture

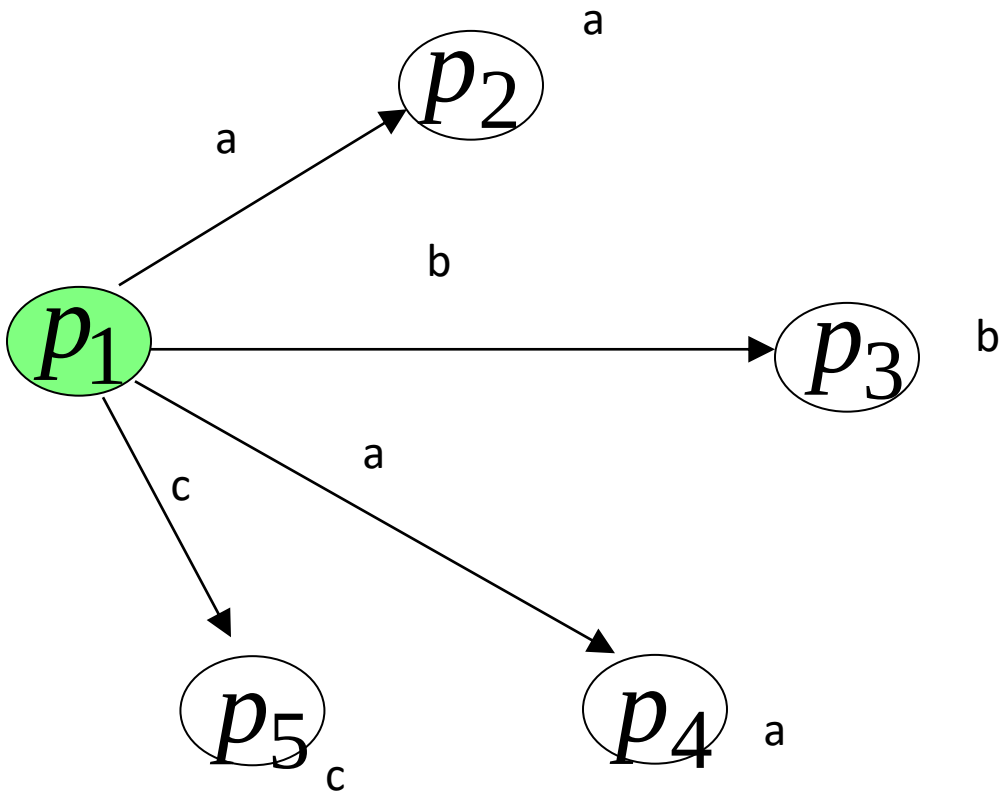
- Logical Clocks
  - Vector Clocks
- Failures in Distributed Systems
  - Link Failures
  - Crash Failures (faulty processors)
  - Byzantine processors
- Consensus Problem:
  - Definitions of termination, agreement and validity
  - Fault-free Algorithm
  - F-resilient Consensus Algorithm

- Lemma: In the algorithm, at the end of the round with no failure, all the processors know the same set of values.

- Proof: For the sake of contradiction, assume the claim is false. Let x be a value which is known only to a subset of (non-faulty) processors. But when a processor knew x for the first time, in the next round it broadcasted it to all. So, the only possibility is that it received it right in this round, otherwise all the others should know x as well. But in this round there are no failures, and so x must be received by all.

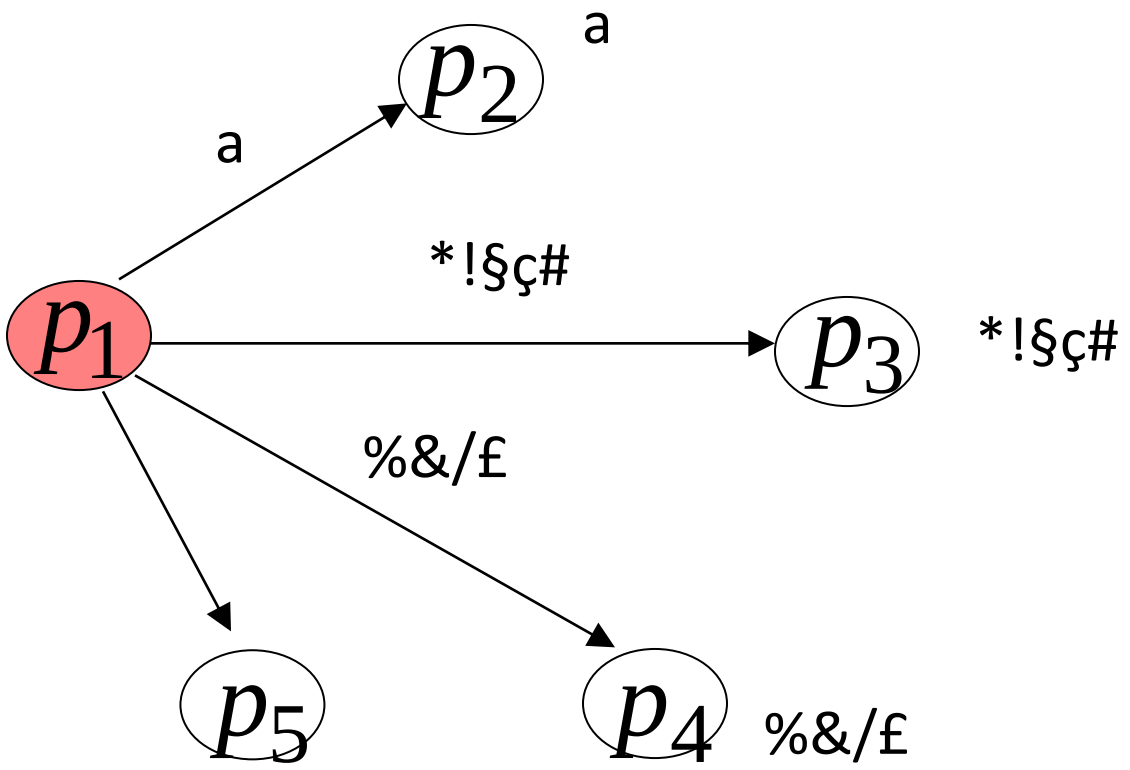
Byzantine Failures

Non-faulty processor

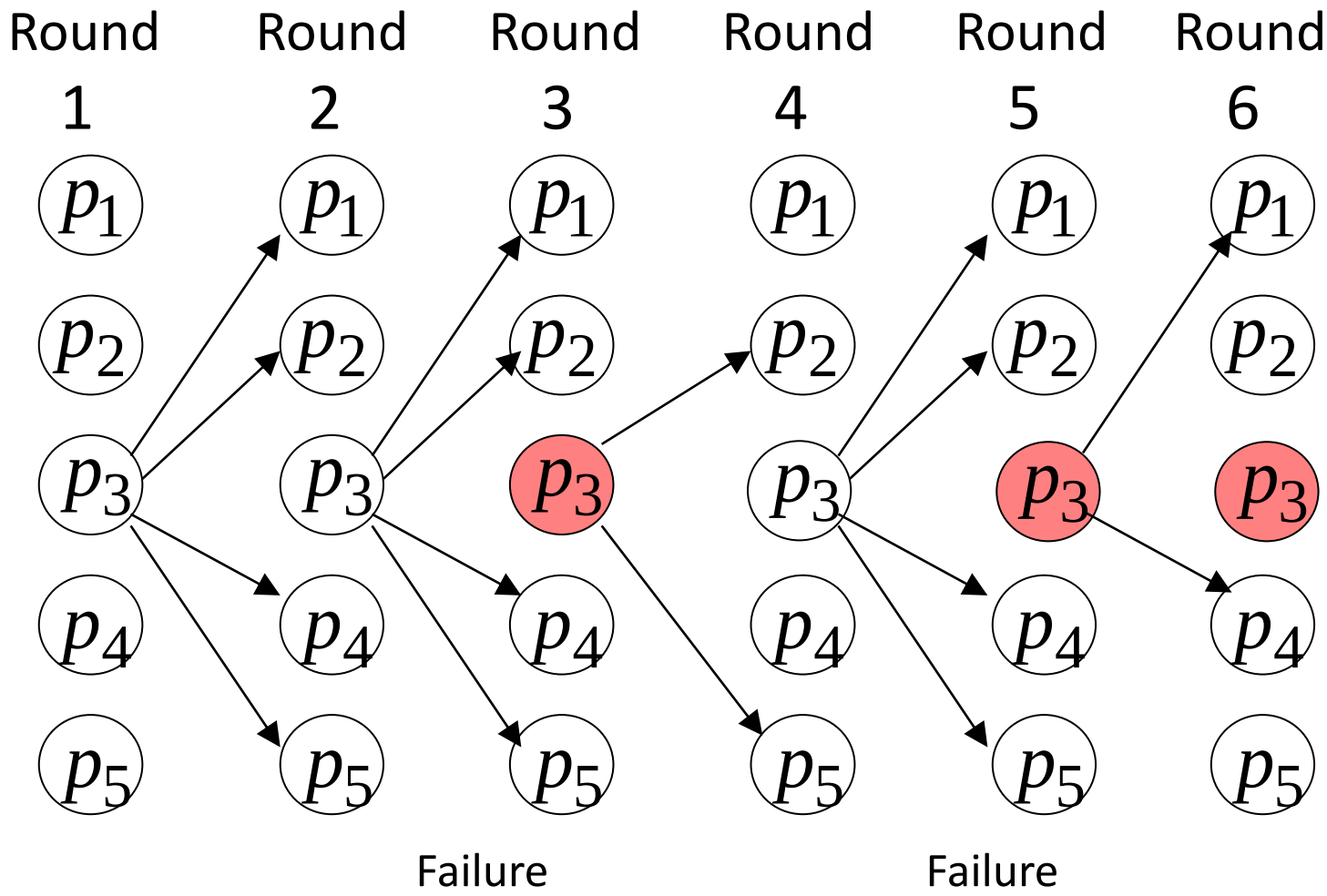


Byzantine Failures

Faulty processor



Processor sends arbitrary messages, plus some messages may not be sent



After failure the processor may continue functioning in the network

Consensus under link failures: the N generals problem

- There are N generals of the same army who have encamped a short distance apart.
- Their objective is to capture a hill, which is possible only if they attack simultaneously.
- If only a few generals attack, they will be defeated.
- The N generals can only communicate by sending messengers, which is not reliable.
- Is it possible for them to attack simultaneously?

Here the situation is much harder

- Generally we need at least $3f+1$ processes in a system to tolerate f Byzantine failures
  - For example, to tolerate 1 failure we need 4 or more processes
- We also need $f+1$ “rounds”
- Let’s see why this happens

Byzantine scenario

- Generals (N of them) surround a city
  - They communicate by courier

- Each has an opinion: “attack” or “wait”
  - In fact, an attack would succeed: the city will fall.
  - Waiting will succeed too: the city will surrender.
  - But if some attack and some wait, disaster ensues

- Some Generals (f of them) are traitors… it doesn’t matter if they attack or wait, but we must prevent them from disrupting the battle
  - Traitor can’t forge messages from other Generals

A timeline perspective

- Suppose that p and q favor attack, r is a traitor and s and t favor waiting… assume that in a tie vote, we attack

A timeline perspective

- After first round collected votes are:
  - {attack, attack, wait, wait, traitor's-vote}

What can the traitor do?

- Add a legitimate vote of “attack”
  - Anyone with 3 votes to attack knows the outcome
- Add a legitimate vote of “wait”
  - Vote now favors “wait”
- Or send different votes to different folks
- Or don’t send a vote, at all, to some

Outcomes?

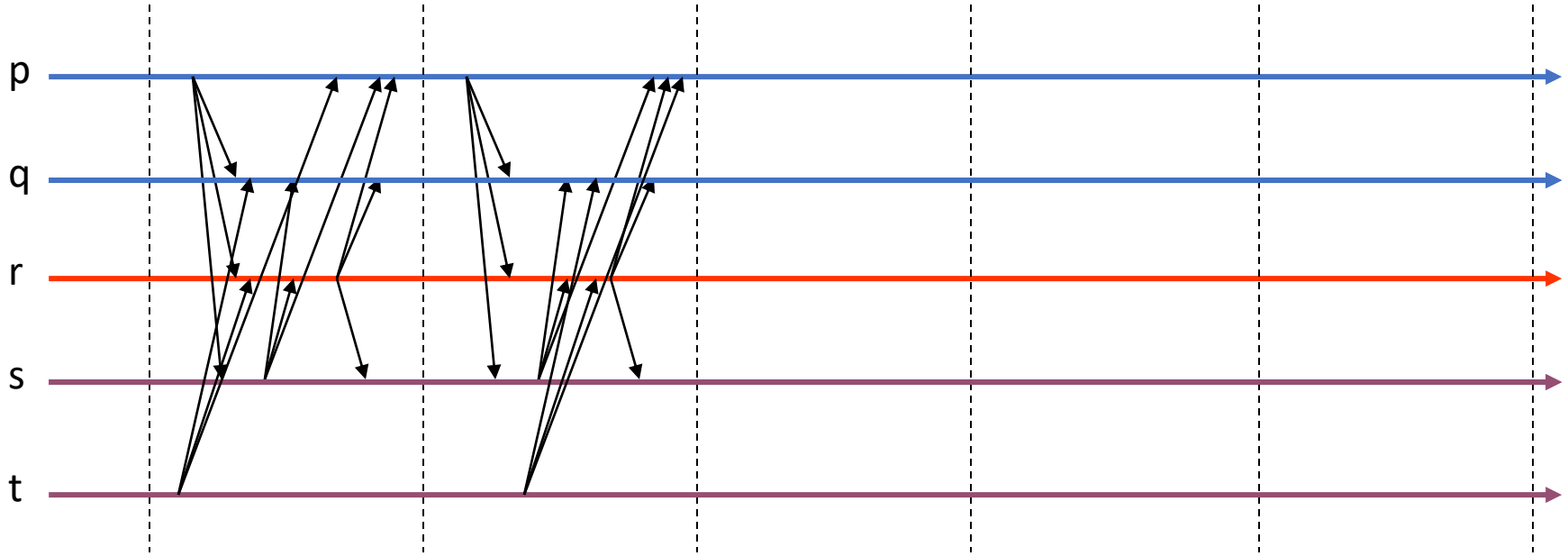
- Traitor simply votes:
  - Either all see {a,a,a,w,w}
  - Or all see {a,a,w,w,w}
- Traitor double-votes
  - Some see {a,a,a,w,w} and some {a,a,w,w,w}
- Traitor withholds some vote(s)
  - Some see {a,a,w,w}, perhaps others see {a,a,a,w,w,} and still others see {a,a,w,w,w}
- Notice that traitor can’t manipulate votes of loyal Generals!

What can we do?

- Clearly we can't decide yet; some loyal Generals might have contradictory data
  - In fact if anyone has 3 votes to attack, they can already "decide".
  - Similarly, anyone with just 4 votes can decide
  - But with 3 votes to "wait" a General isn't sure (one could be a traitor…)

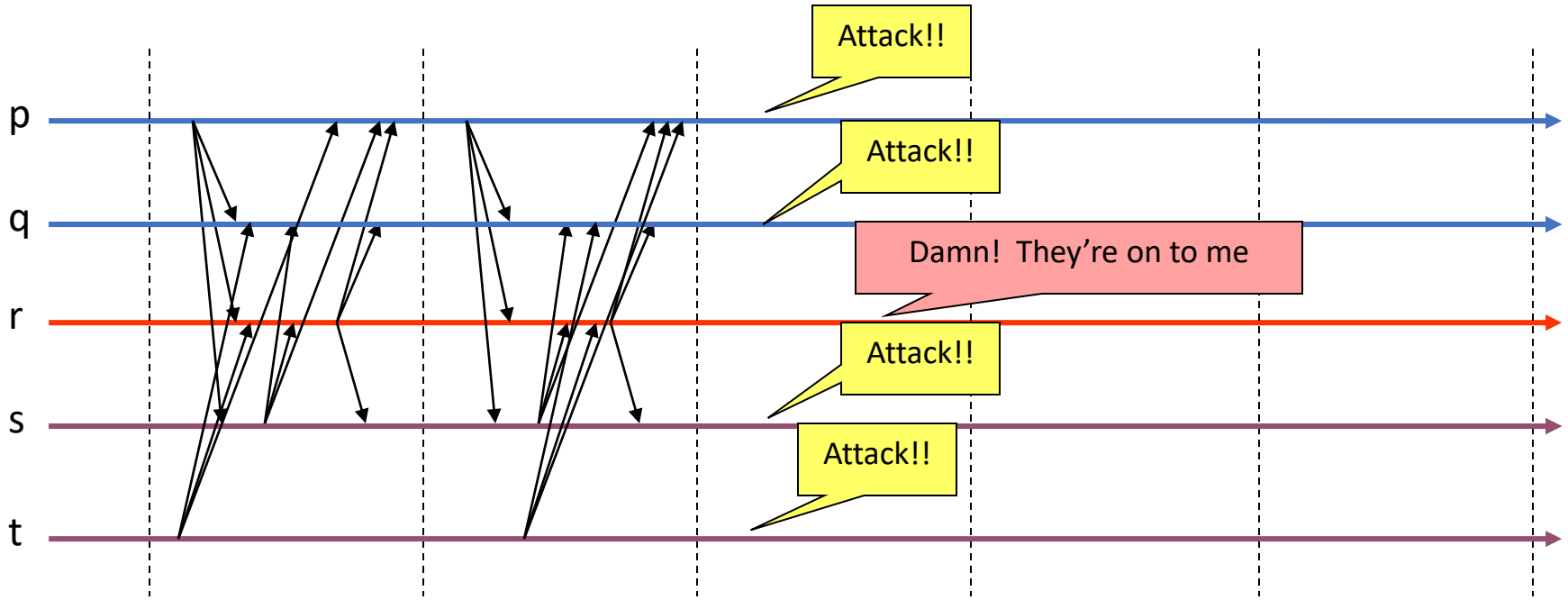
- So: in round 2, each sends out "witness" messages: here's what I saw in round 1

A timeline perspective



- In second round if the traitor didn’t behave identically for all Generals, we can weed out his faulty votes

A timeline perspective



- We attack!

Replicated state machines

- Allows a collection of servers to
  - Maintain identical copies of the same data
  - Continue operating when some servers are down
    - A majority of the servers must remain up
- Many applications
- Typically built around a distributed log
- Each server stores a log containing commands

Consensus

- Goal is to *operate multiple servers* in identical state for *fault-tolerance*
- Example services:
  - *File system*
  - *Database*
  - *Shared memory*
  - *Ledger*
  - *Key-value store*
- Clients issue *requests* to change *state machines*
- Servers maintain *identical logs* of *linearized client requests*
- Servers maintain *identical state machines*

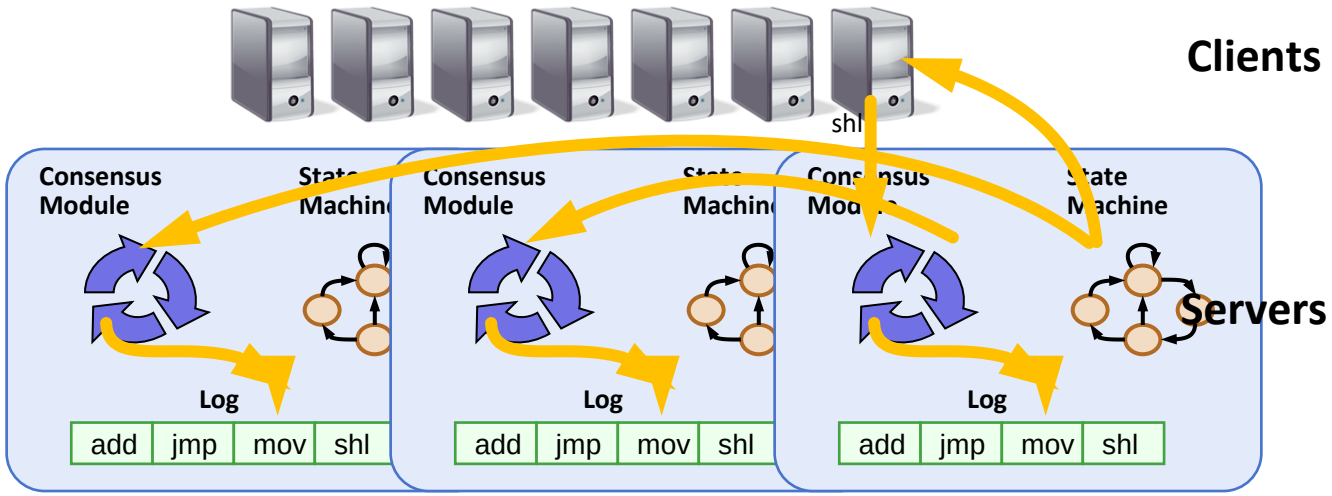
Raft: A Consensus Algorithm for Replicated Logs

Diego Ongaro and John Ousterhout

Stanford University

Link: https://www.cs.princeton.edu/courses/archive/spring21/cos418/docs/L14-raft.pdf

Goal: Replicated Log



- Replicated log => replicated state machine
  - 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
- Failure model: fail-stop (not Byzantine), delayed/lost messages

Approaches to Consensus

Two *general approaches to consensus*:

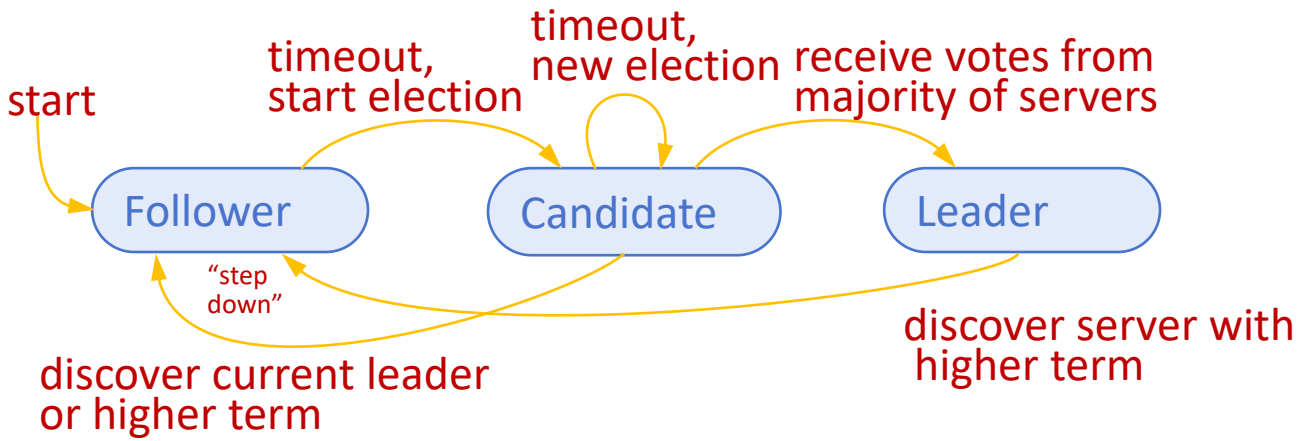
- ***Symmetric, leader-less***:
  - All servers have *equal roles*
  - Clients can contact *any server*
- ***Asymmetric, leader-based***:
  - At any given time, *one server is in charge*, others accept its decisions
  - Clients *communicate* with the *leader*
- ***Raft*** uses a *leader*:
  - *Decomposes* the problem (*normal operation*, *leader changes*)
  - *Simplifies normal operation* (no conflicts)
  - More *efficient* than *leader-less approaches*

Raft Overview

1. **Leader election**:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. **Normal operation (basic log replication)**
3. **Safety and consistency after leader changes**
4. **Neutralizing old leaders**
5. **Client interactions**
   - Implementing linearizable semantics
6. **Configuration changes**:
   - Adding and removing servers

Server States

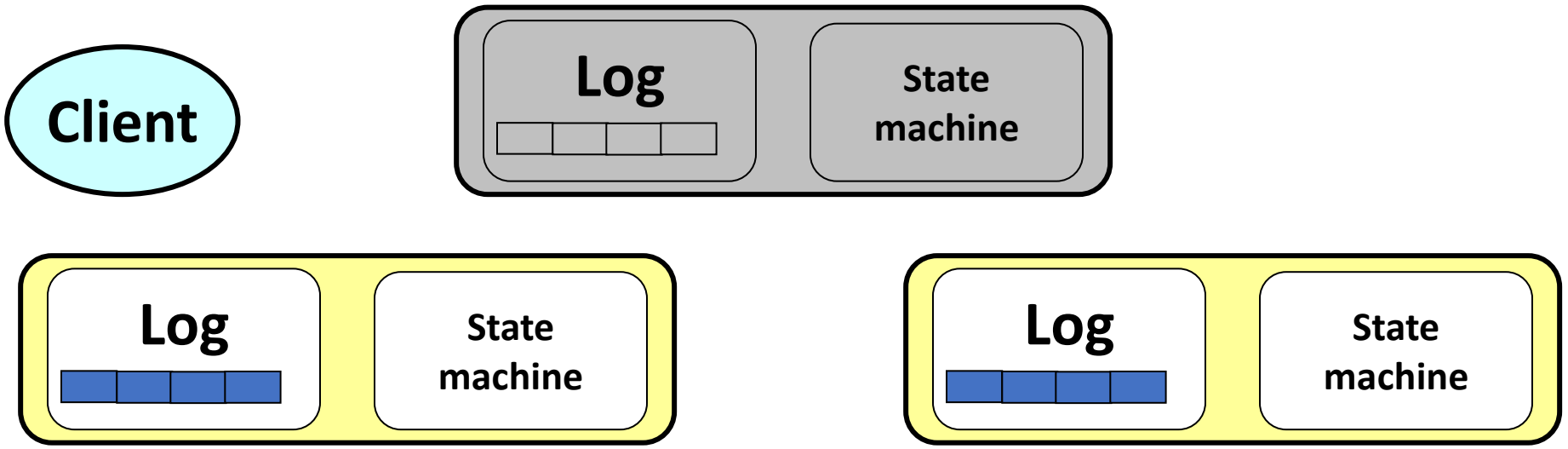
- At any given time, each server is either:
  - **Leader**: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - **Follower**: completely passive (issues no RPCs, responds to incoming RPCs)
  - **Candidate**: used to elect a new leader
- Normal operation: 1 leader, N-1 followers



Leader elections

- Servers start being ***followers***
- Remain followers as long as they receive valid RPCs from a leader or candidate
- When a follower receives no communication over a period of time (the ***election timeout***), it starts an election to pick a ***new leader***

The leader fails



- Followers notice at ***different times*** the lack of heartbeats
- Decide to elect a new leader

Starting an election

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

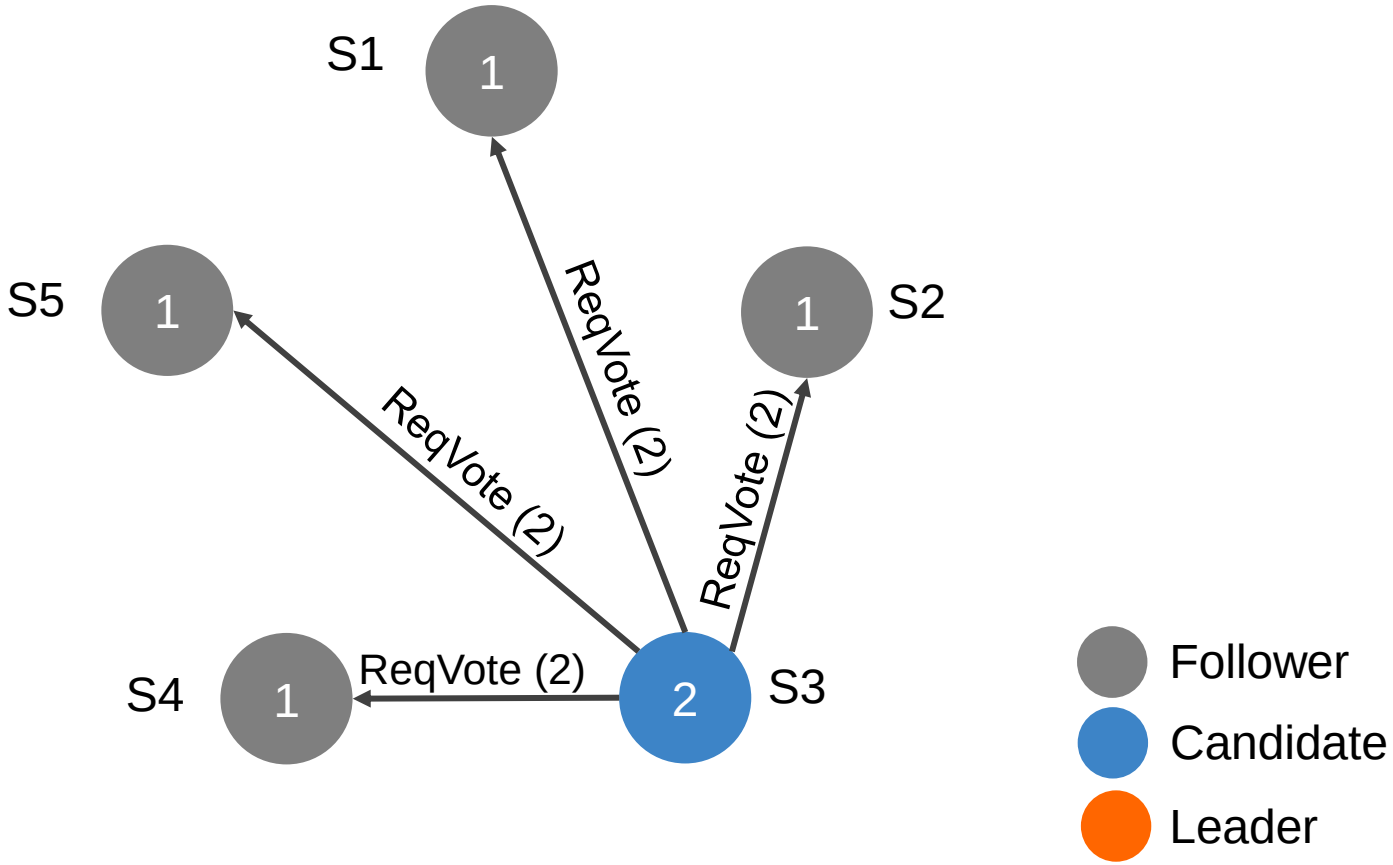
Acting as a candidate

- 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

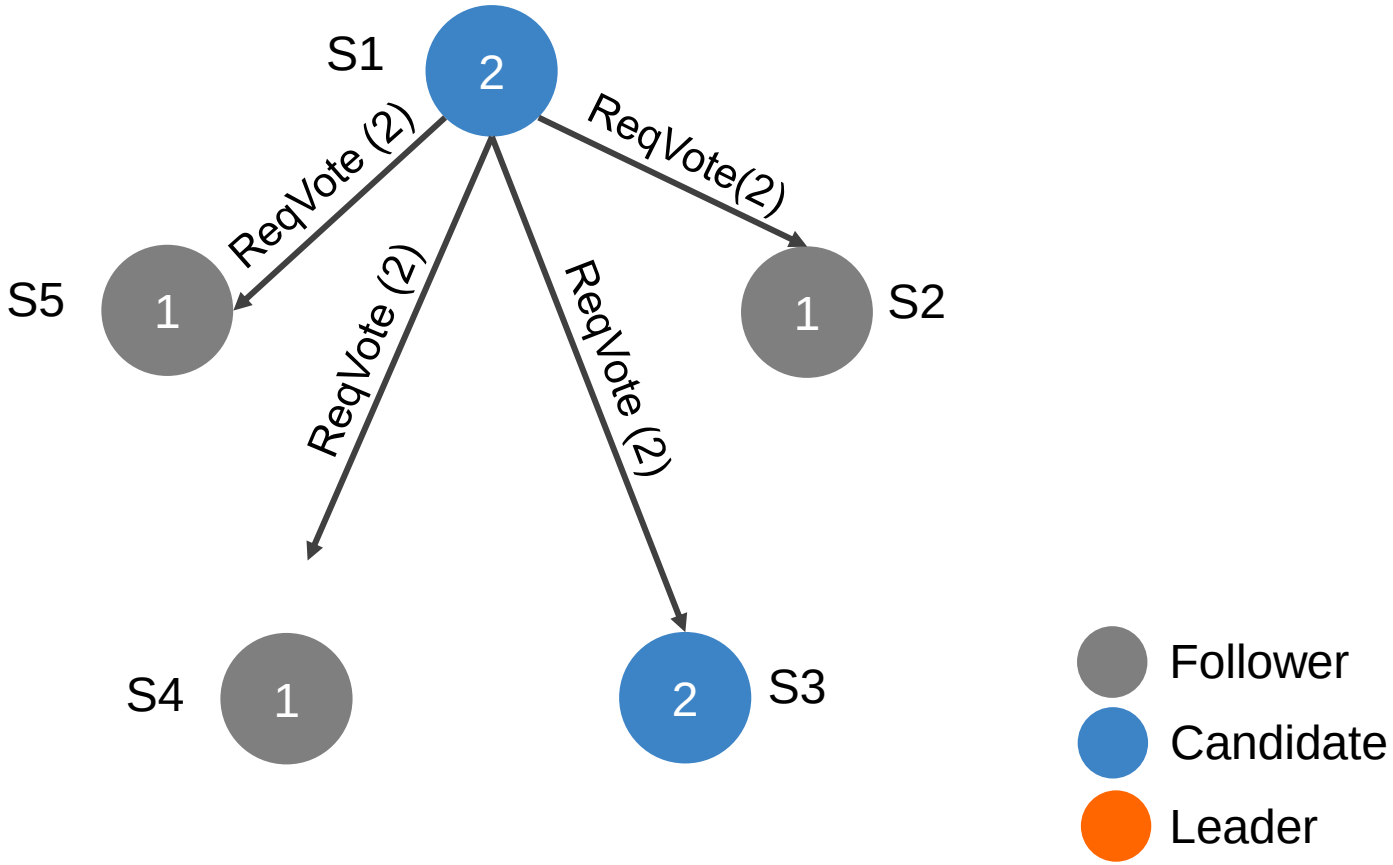
Winning an election

- Must receive votes from a majority of the servers in the cluster for the same term
  - Each server will vote for at most one candidate in a given term
    - The first one that contacted it
- Majority rule ensures that at most one candidate can win the election
- Winner becomes ***leader*** and sends heartbeat messages to all of the other servers
  - To assert its new role

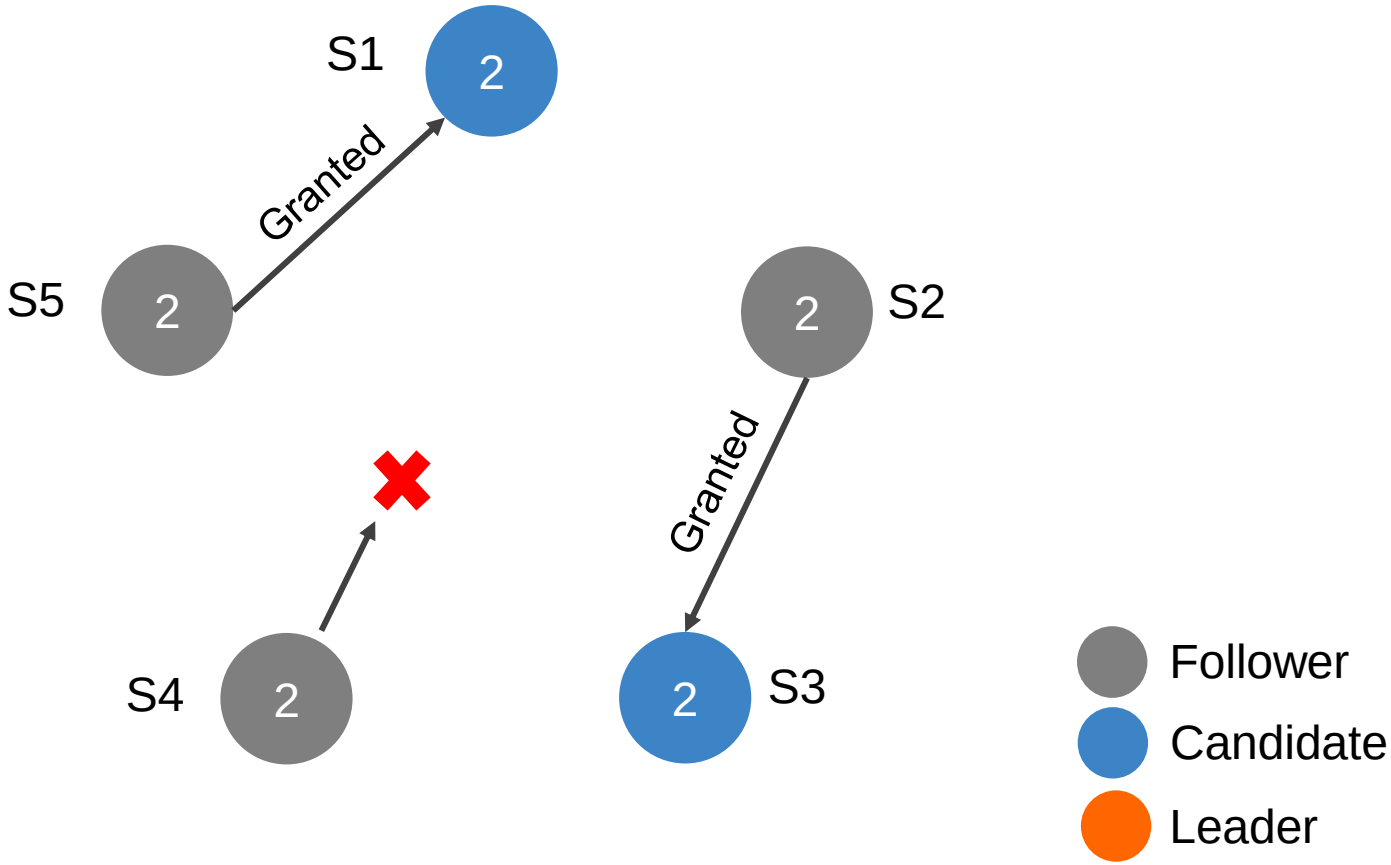

S1
1
S5
1
1
S2
S4
1
1
S3
Follower
Candidate
Leader



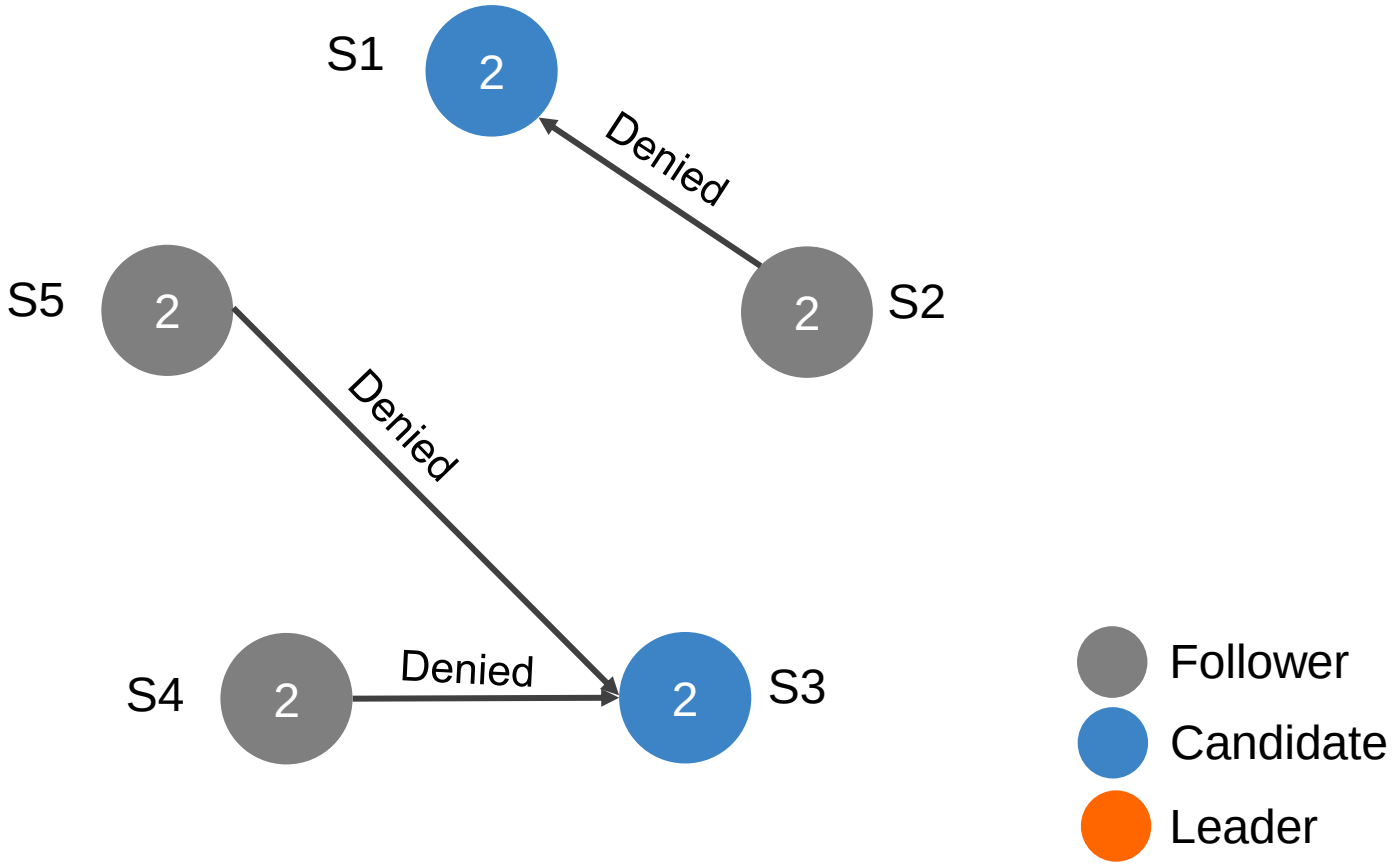
S3 timeouts, switch to candidate state,
increment term, vote itself as a leader and ask everyone else to confirm

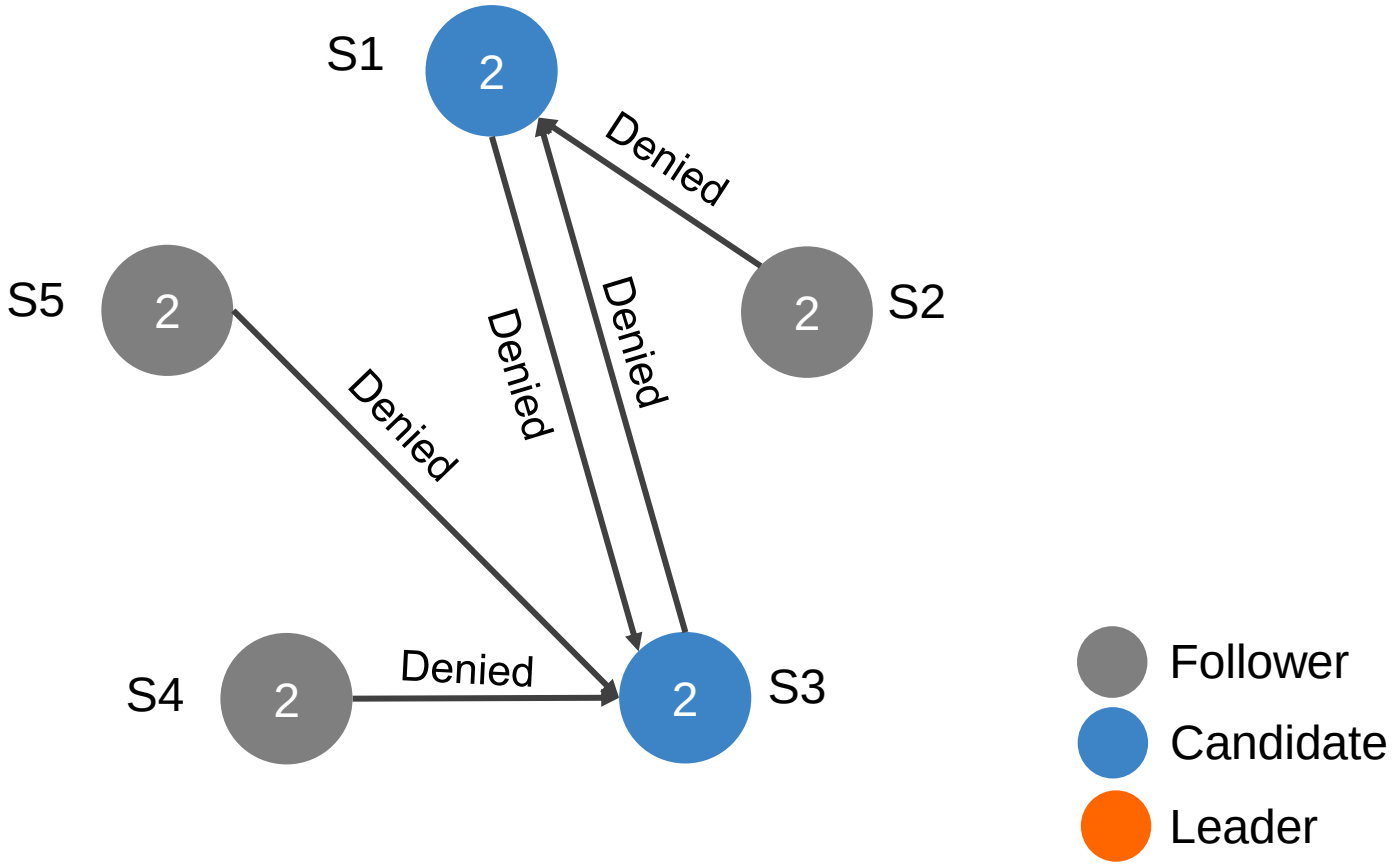


Concurrently S1 timeouts, switch to candidate state,
increment term, vote itself as a leader and ask everyone else to confirm

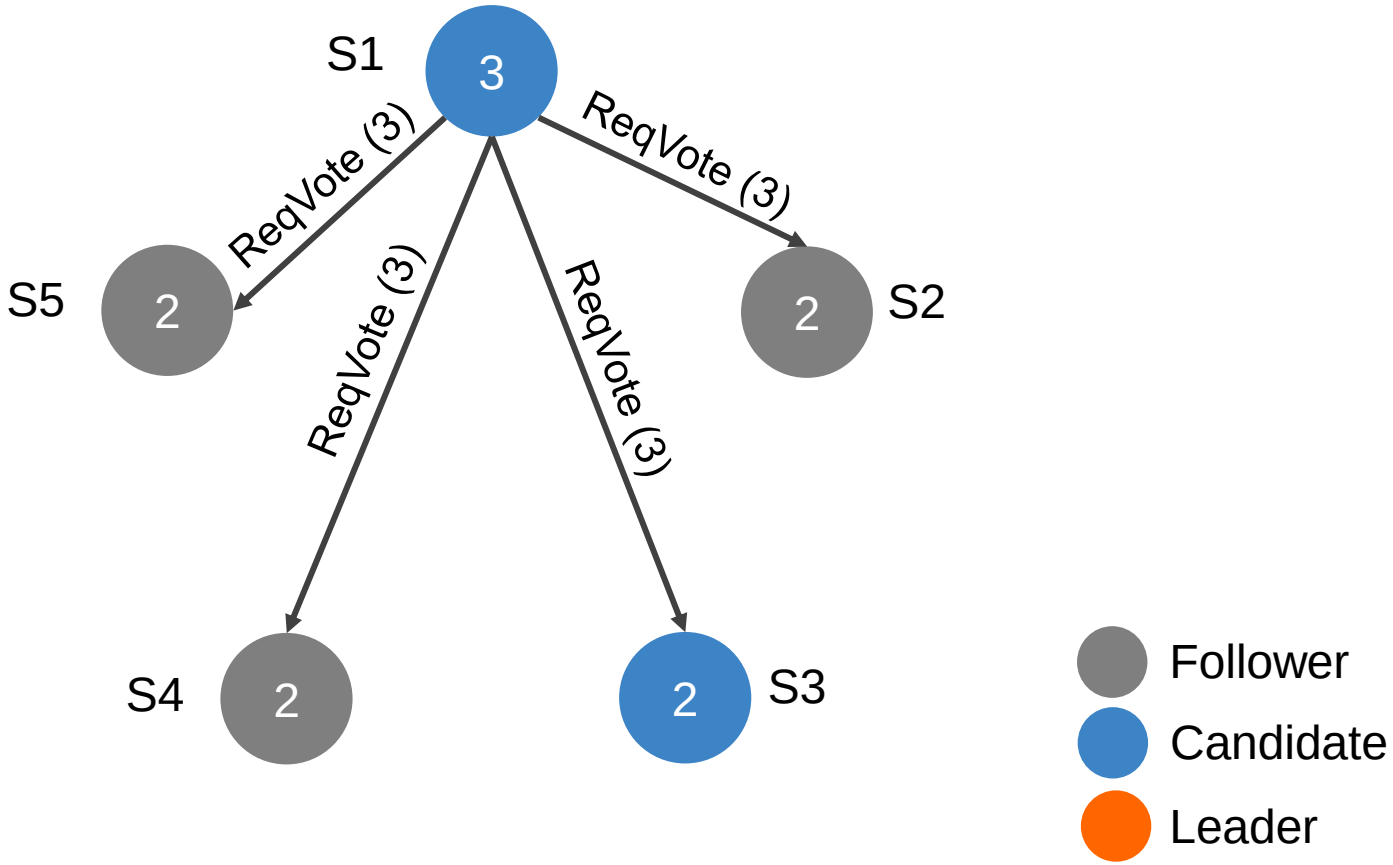


S4, S5 grant vote to S1
S2 grants vote to S3

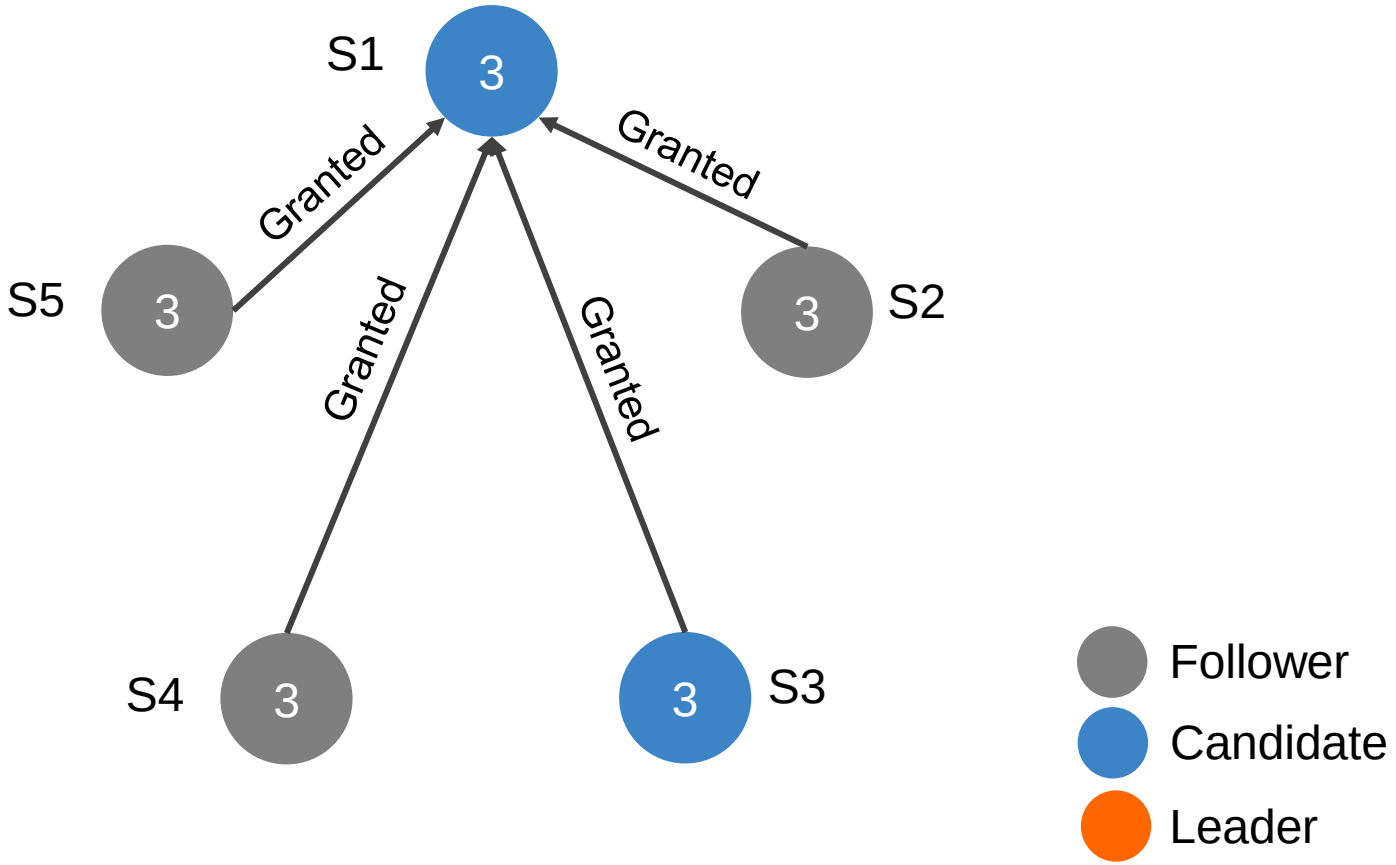

S1
2
Denied
S5
2
2
S2
Denied
S4
2
Denied
2
S3
Follower
Candidate
Leader



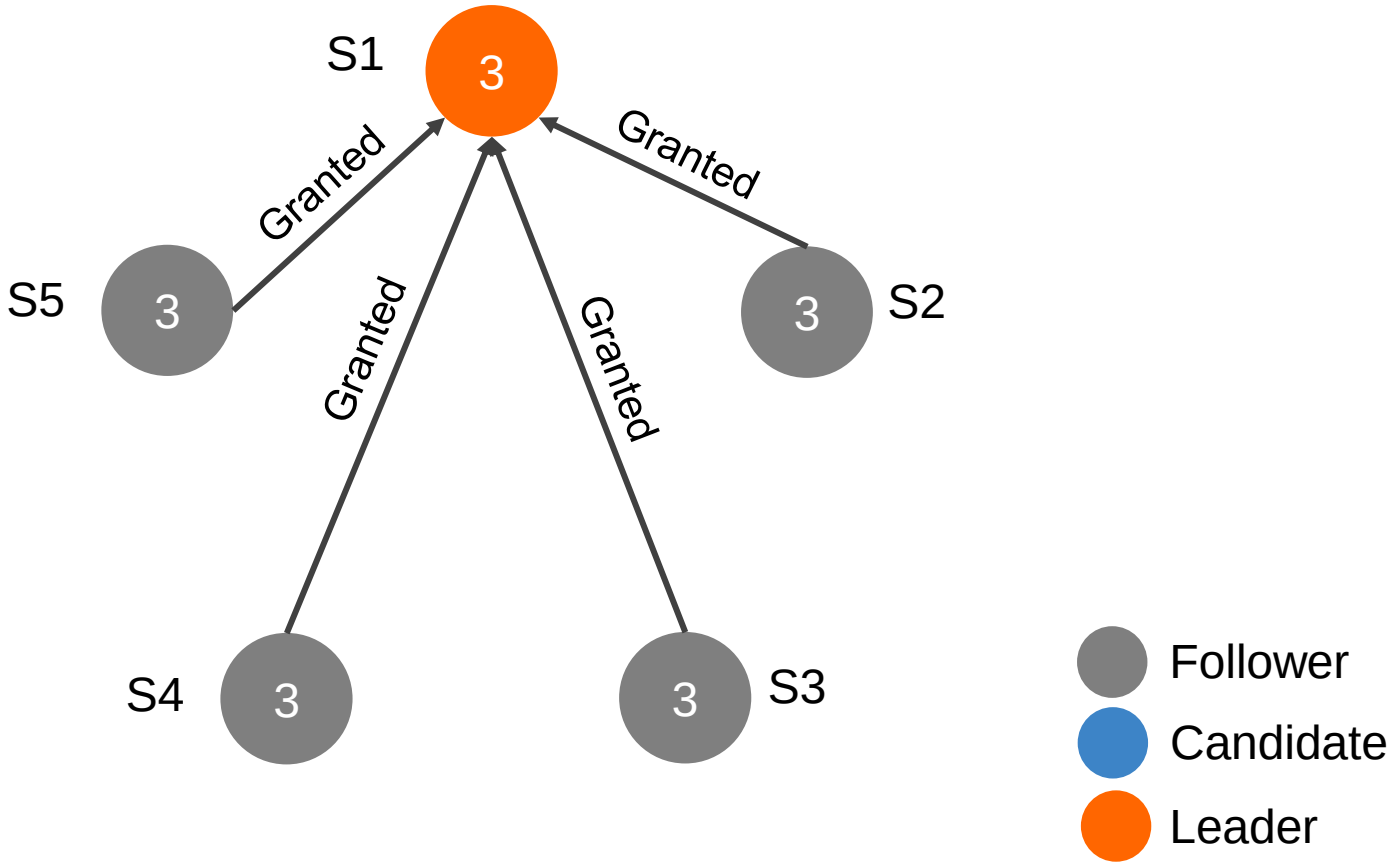
Neither candidate gets majority.
After a random delay between 150-300ms try again.



S1 initiates another election for term 3.



Everyone grants the vote to S1



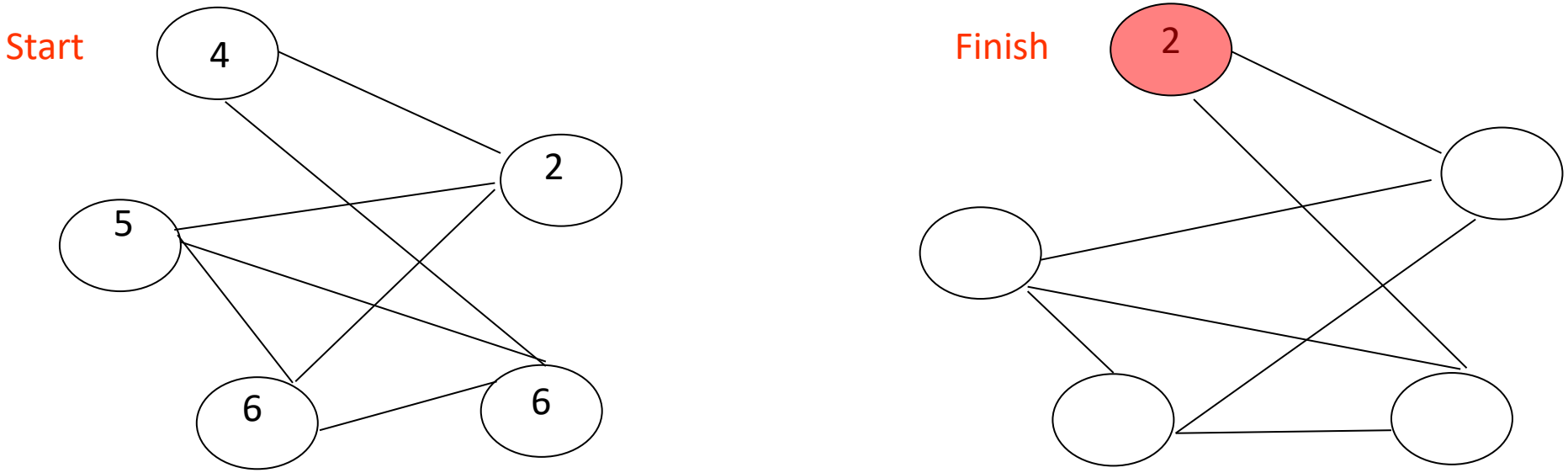
S1 becomes leader for term 3,
and the others become followers.

RAFT - Visualization

- http://thesecretlivesofdata.com/raft/

Quiz 06 (10 minutes)

- With the given start configuration, show a possible finish configuration that enables “AGREEMENT” (assume the top process is faulty)



- Show with an example how vector clocks work, with your example having 3 processes, at least 2 local events and at least 3 messages:

References

1. Slides of Dr. Haroon Mahmood

Helpful Links (Raft):

1. https://raft.github.io/
2. https://people.cs.rutgers.edu/~pxk/417/notes/raft.html
3. https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14.pdf