THE UNIVERSITY of EDINBURGH

Distributed Systems
Fall 2024

Yuvraj Patel

Today's Agenda

Consensus

- Basics
- Paxos and Raft

Leader Election

Next Class is on Tuesday(22/10) and not Monday(21/10)

Why Consensus?

Multiple use-cases

- Replication – make sure the replicated data is same on all the nodes
- Failure Detection – a machine/leader has failed/stopped responding
- Leader Election – elect a leader to initiate a snapshot, etc.
- and many more…

All the above scenarios involve

- Multiple parties
- Presence of faults
- Coordinate amongst themselves
- Need to agree to something or arrive at a decision

Consensus Problem – Single value formulation

Consensus Protocol

Consider a distributed system with n nodes

- Each node i has an input x_i
- Faults may happen at arbitrary times

Output

- All nodes agree on a single value; Value cannot change later

Guarantee the following

- Termination: Every non-faulty node eventually decides
- Agreement: All non-faulty nodes decide on the same value
- Validity: The decided value must be the input of at least one node

Consensus Protocol (contd…)

Not democratic; Value proposed by a small minority can be decided

Consensus possible depends on multiple parameters

Most important parameters

- System Model – Synchronous or Asynchronous
- Fault types – Crash or Byzantine

Synchronous vs. Asynchronous Systems

Synchronous systems

- Process execution speeds and message delivery times are bounded
- Can detect omission and timing failures

Asynchronous systems

- No assumptions about process execution speed or message delivery times
- Cannot reliably detect crash failures

Consensus

- Challenging in Asynchronous systems
- Solvable in Synchronous systems
- Algorithm for Asynchronous systems will work for Synchronous systems

Impossibility in Asynchronous Systems

Fischer, Nancy & Paterson show it is impossible to achieve consensus in asynchronous system with a single faulty process

They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults

- With no crash too, algorithm may not terminate
- Proof constructs infinite non-terminating runs

One of the most fundamental results in distributed systems.

- Interested students can check the FLP paper -- https://dl.acm.org/doi/pdf/10.1145/3149.214121

How To Solve Consensus Then...

Paxos algorithm – Invented by Leslie Lamport

Most popular consensus solving algorithm

- Does not solve consensus problem (FLP still applies)

Used in many real-world systems – Yahoo, Google, etc.

Provides safety and eventual liveness

- Safety – Consensus is not violated
- Liveness – Good chance consensus reached sometime in future; No guarantee it will terminate

Assume partially synchronous systems to avoid impossibility aspects

Paxos Algorithm

Role's node assume

- Proposers: Those who propose values
- Acceptors: Those who accept a proposed value
- Learners: Those who learn the proposed value after a consensus is reached
- One node can play two roles simultaneously

Other assumptions

- Nodes communicate with each other via messages
- Nodes operate independently and at different speed
- Nodes can crash or restart while operating
- Message receipt is asynchronous and can take longer time to be delivered, can be duplicated, and lost in the network. Messages are never corrupted

For majority, need 2m + 1 nodes to handle m failures

Paxos Algorithm – Safety & Liveness

Safety

- Only a single value is chosen
- Only chosen values are learned by nodes
- Only a proposed value can be chosen

Liveness

- Some proposed value eventually chosen if fewer than half of processes fail
- If value is chosen, a process eventually learns it

Paxos is safe but often live

Strawman Solutions

Single Acceptor: n proposers, 1 acceptor

- Acceptor accepts first value received
- Problem: Single acceptor single point of failure (no liveness)

Multi Acceptor: n proposers, n acceptors

- Acceptor accept first value it receives
  - Problem: Split Vote
- Acceptor accepts every value it receives
  - Problem: Conflicting Choices

Remarks: Once a value has been chosen, future proposals must propose/choose that same value

Proposal Numbers & Rounds

Each proposal has a unique number

- Higher numbers take priority over lower numbers (Older proposals rejected)
- Proposers always propose having a proposal number higher than it has seen/used

Simple Approach: Proposal number = Round Number + Node-ID

- Round Number – Higher than largest round number seen so far
- Need to remember largest round number so far
- Cannot reuse round number value after crash or reboots

Phases

Two phases – Prepare & Accept

Prepare Phase

- Find out any chosen values so far
- Block older and uncompleted proposals

Accept Phase

- Inform acceptors to accept a specific value

Analogous to how government passes laws

- Elect leader
- Propose a Bill
- Accept the Bill and turn in to a Law

Algorithm – Prepare Phase

Proposer

- Choose proposal number n, send <prepare, n> to acceptors

Acceptor

- Only receiving a prepare message
  - If $n > n_h$, where n_h is the highest proposal seen so far by the acceptor

    $n_h = n$. (Promise to not accept older proposals)

    If no prior proposal accepted,

    reply <promise, n, NULL>

    Else

    reply <promise, n, (n_a, v_a)>
  - Else

    Reply <prepare-failed>

Algorithm – Accept Phase

Proposer

- If receive promise from majority of the acceptors,
  Determine any earlier chosen value v_a for n_a and choose latest value or any value v selected by the proposer
  send <accept, n, v> to acceptors

Acceptors

- If $n >= n_h$
  $n_a = n_h = n$
  $v_a = v$
- reply <accept, n_h>

Proposer

- When responses received from the majority
  If any $n_h > n$
    Start from prepare phase again
  Else
    Value is chosen

Example – Everything works fine

Example – Acceptor failure

Accept Phase Failure

Prepare Phase Failure

Example – Proposed failure

Prepare Phase Failure

Example – Proposed failure

Accept Phase Failure

Failure Handling Summary

One proposer

- One or more acceptors fail
  - Still works as long as majority nodes are up
- Proposer fails in prepare phase
  - No-op; another proposed can make progress
- Proposer fails in accept phase
  - Another proposer overwrites or finishes the job of failed proposer

Two or more simultaneous proposers

- More complex
- Can lead to livelock (fix with leader election)

Multi Paxos

Basic Paxos comprises two rounds

For real-world systems like databases, every single operation needs to go through Basic Paxos rounds, which is costly

Multi Paxos – Creating a log of agreements

- Assume Proposer is stable
- Use Phase 1 for the Proposer election
- Use Phase 2 multiple times and work on multiple values being accepted

Raft – Consensus Protocol

Designed to be easy to understand

Equivalent to Paxos in fault-tolerance and performance

Decomposed into relatively independent sub-problems

Raft vs Paxos

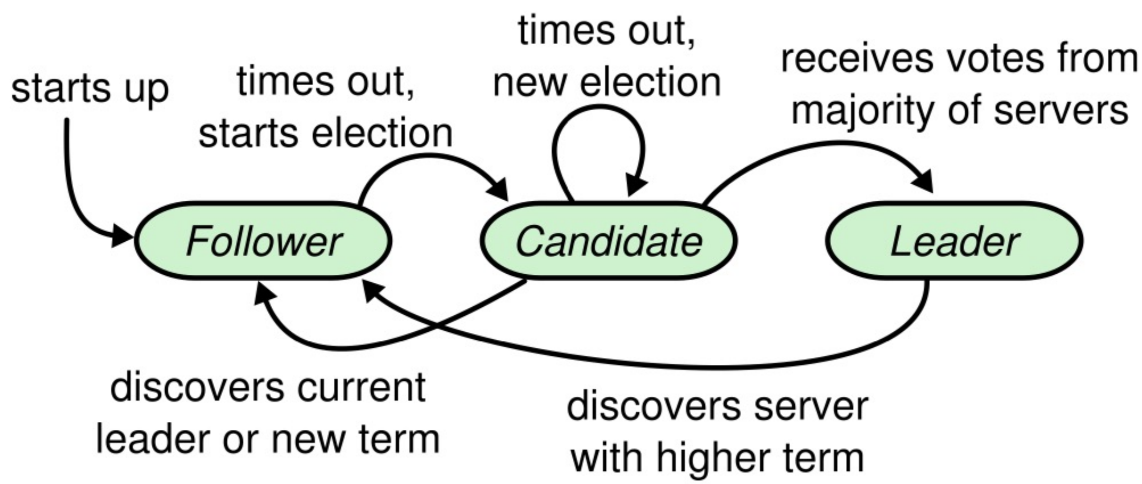
- Paxos – agrees separately on each client operation
- Raft – agrees on each new leader (and on tail of the log); agreement not required for most client operations

Raft is Paxos optimized for log appends

Roles in Raft

A node can be either

- Follower – Passive nodes; They issue no requests on their own; Respond to requests from leaders and candidates
- Candidate – Used to elect a new leader; Transitions from a Follower and transitions to a leader or follower
- Leader – Handles all client requests



High-Level Understanding

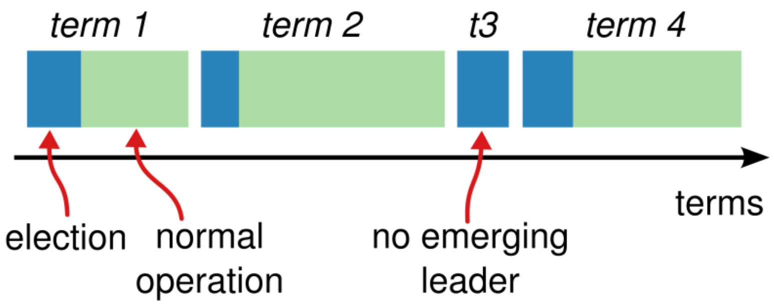
Leader Election

Raft divides time into terms of arbitrary length; terms are numbered consecutive integers

Each term begins with an election, where one or more candidates attempts to become a leader

Two possible outcomes of an election

- Candidates wins with majority; Elected leader for the term
- Split Votes



Leader Election – Normal Scenario

Leader Election – Split Votes

Leader Election

Term acts as a logical clock and helps detect obsolete information such as stale leaders

Each node stores a current term number, increases monotonically

Current terms exchanged while normal communication

- One node's current term smaller than others, it updates it term to larger value
- If leader/candidate discovers its term is out of date; revert to follower role

If node receives a request with a stale term number, reject the request

Log Replication

Log entries over time

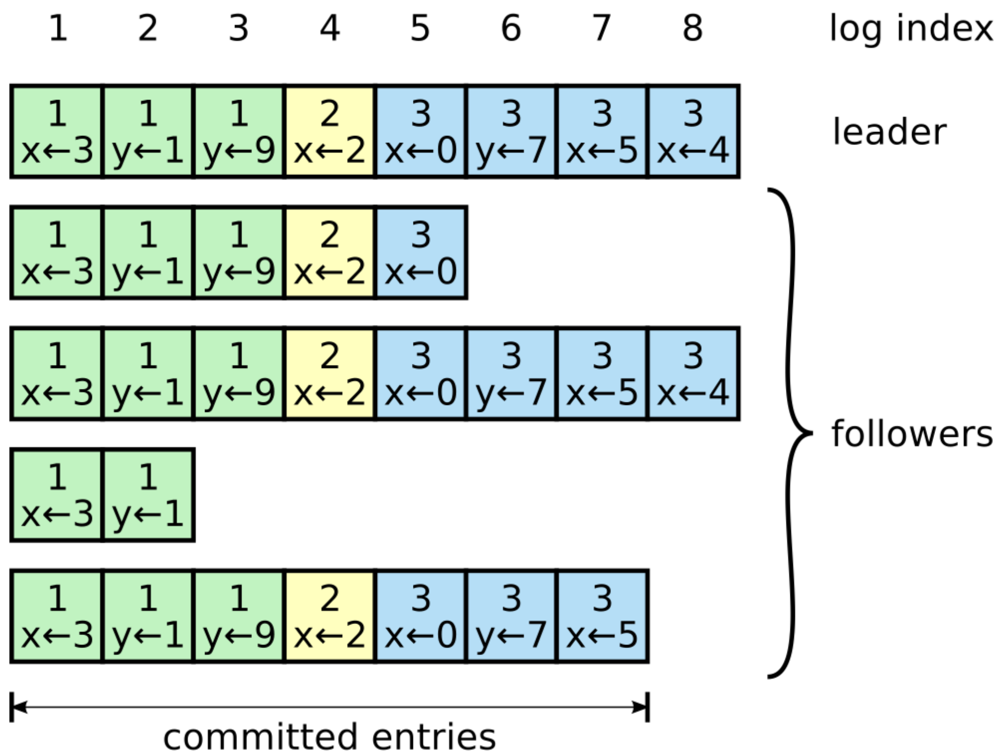
A leader's log is the ultimate truth

While election, ensure that the leader has all committed entries

Leader keeps track of each follower's log

Leader ensures all followers are up to date

- Either remove uncommitted log entries or append to log entries



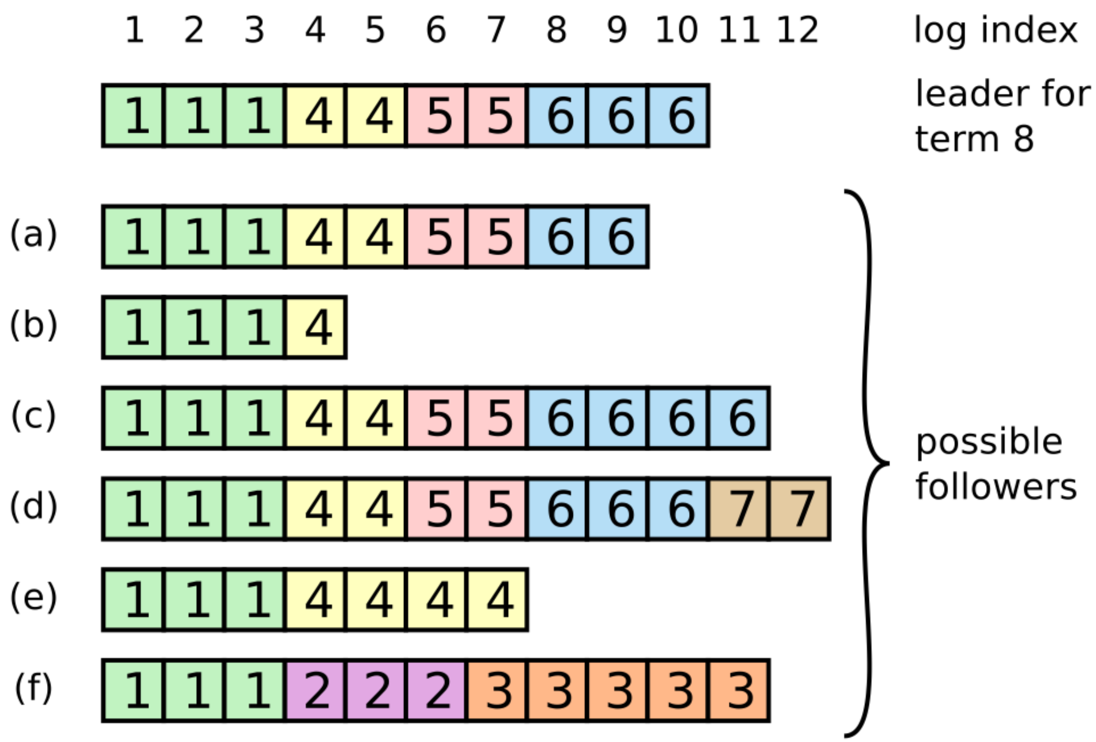
Log entries over time (...contd)

Different possibilities

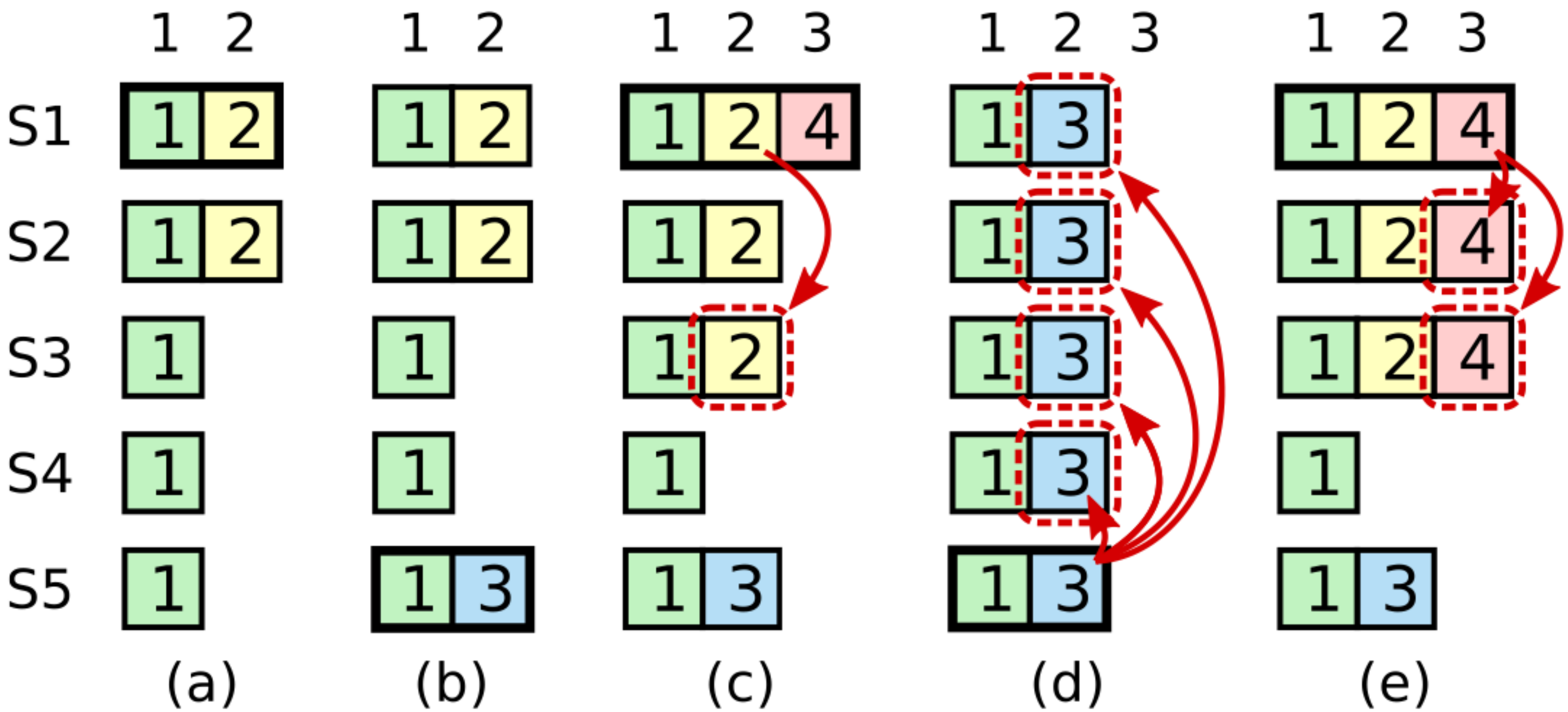
Missing entries → (a-b)

Extra uncommitted entries → (c-d)

Missing + Extra uncommitted entries → (e-f)



Committing Entries From Previous Terms



Leader Election Problem

Need to elect leader to perform tasks and broadcast leader details

If leader fails

- Someone will detect leader failed
- Initiate a leader election to elect another leader
- Only one leader elected, and everyone agrees on who is the leader

System Model & Assumptions

System Model

- N nodes in the system; each node having unique id
- Communicate via messages; messages will eventually be delivered
- Failures/crashes may happen at arbitrary time

Assumptions

- Any node can call for an election
- Any node can call for atmost one election at a time
- Multiple processes can call for an election simultaneously; still lead to a single leader
- Result independent of who calls for an election

Bully Algorithm

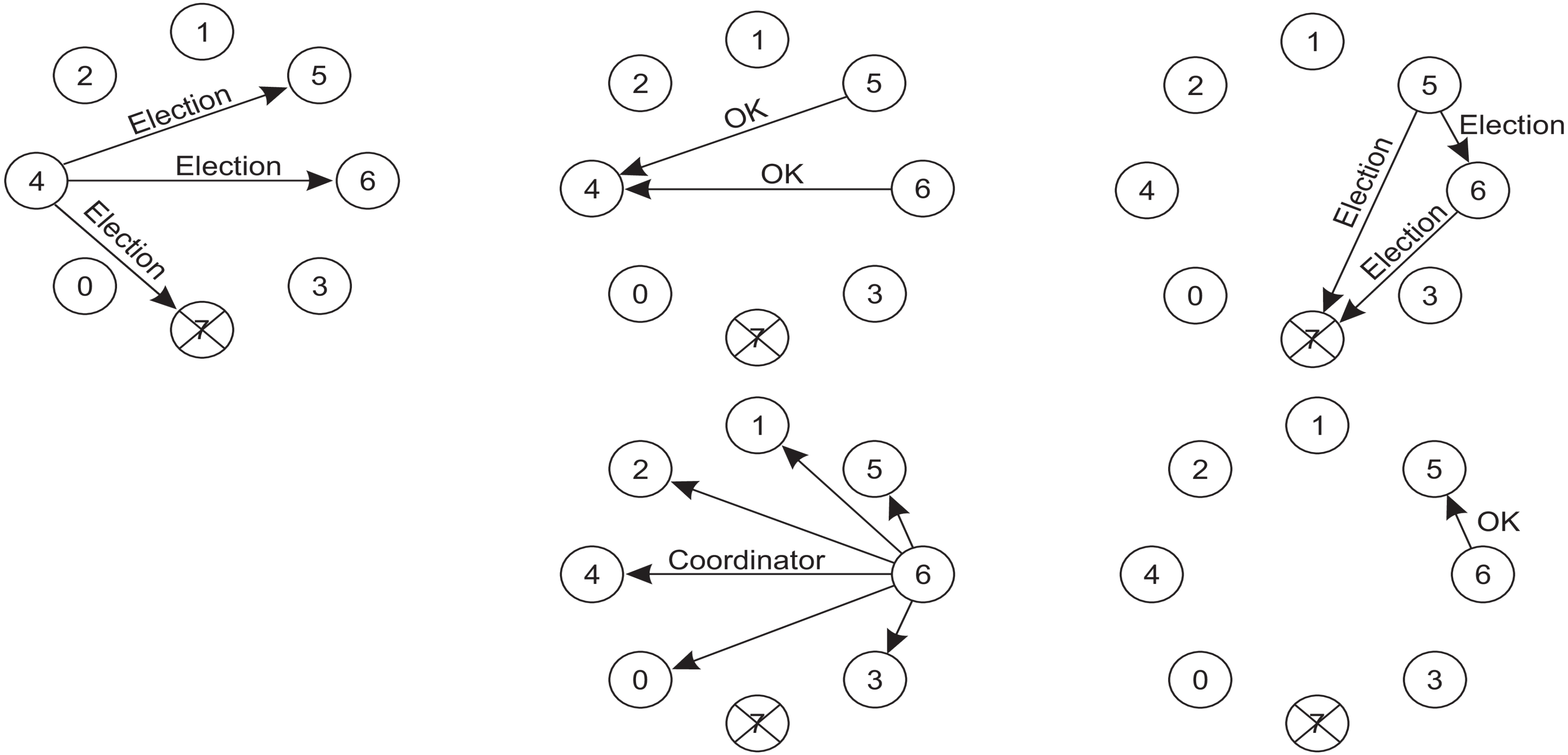
Key Idea: Node with highest ID wins

Consider N nodes $\{N_0, N_1, N_{2\dots} N_n\}$.

Whenever a node N_k notices that the leader is unresponsive, election initiated

- N_k sends an ELECTION message to all the processes with higher IDs: $N_{k+1}, \dots N_n$
- If no one responds, N_k wins
- If one of the higher-up's answers, it takes over and N_k's job is done

Example


1
2
5
Election
Election
4
6
Election
0
3
7
1
2
5
OK
OK
4
6
0
3
7
1
2
5
Election
Election
4
6
Election
0
3
7
1
2
5
Coordinator
4
6
0
3
7
1
2
5
OK
4
6
0
3
7

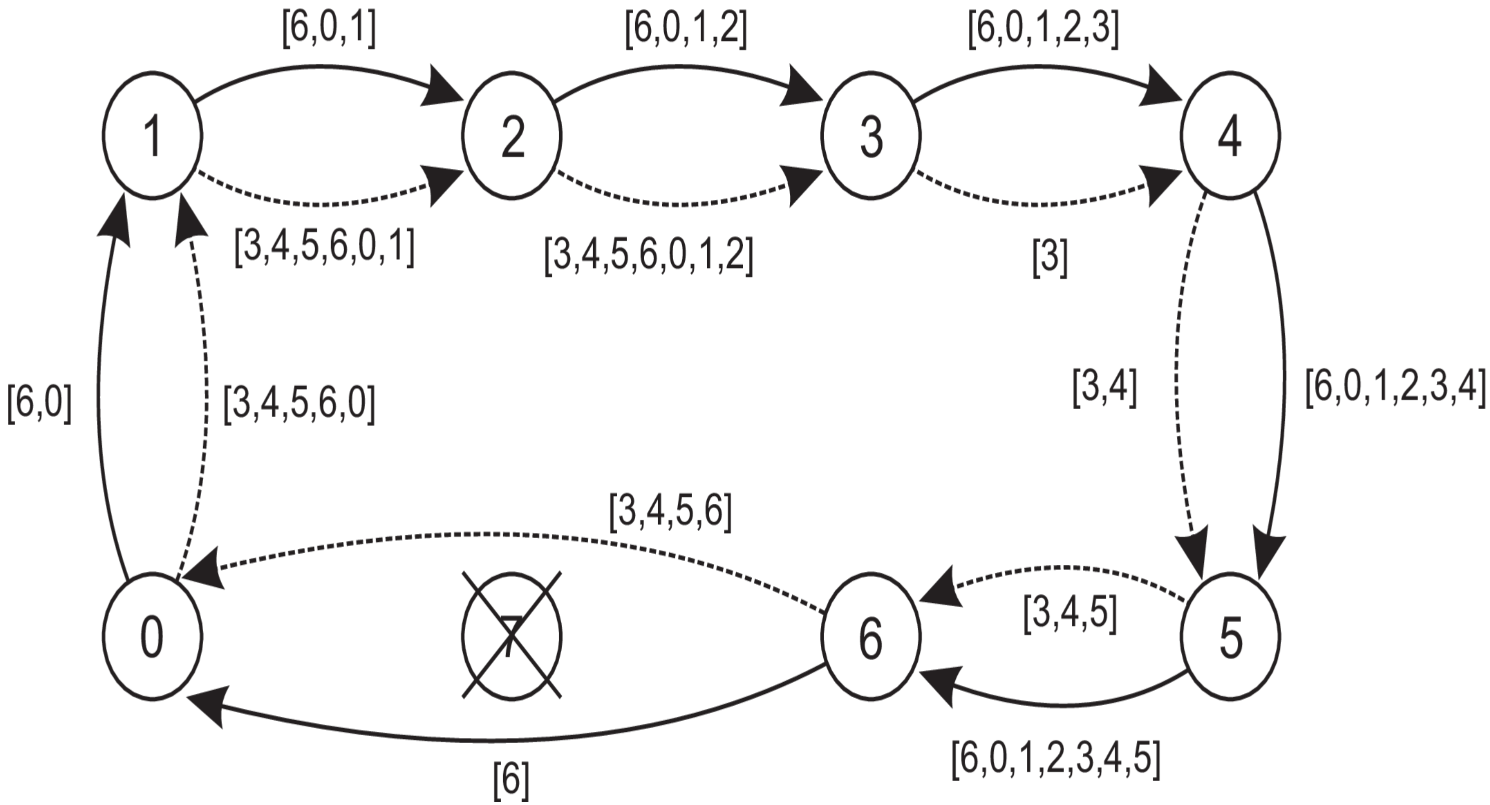
Ring Algorithm

Nodes are organized into a ring. Process with highest id is elected as coordinator

Whenever a node N_k notices that the leader is unresponsive, election initiated

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passe don the next successor
- If a message is passed on, the sender adds itself to the list.
- When the message 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 the living nodes. The one with the highest id is elected as coordinator

Example



The solid line shows the election messages initiated by N_6
The dashed one is election messages initiated by P_3

Both have the same list so it is safe to have two nodes initiating an election