THE UNIVERSITY *of* EDINBURGH

Distributed Systems Fall 2024

Yuvraj Patel

Disclaimer: Slides prepared using multiple sources (UW-Madison – Remzi, Andrea, Mike; Cambridge – Martin Kleepman; Distributed Systems book by MVS/ AST); University of Edinburgh – Yuvraj Patel

Today's Agenda

Consensus

- Paxos and Raft

Mutual Exclusion

Consensus

Multiple use-cases – Replication, Fault Tolerance, Leader Election, etc.

Consensus Problem – Single value formulation

Consensus Protocol

- A distributed system with n nodes, each node i has an input x_i, faults may happen arbitrarily
- Output - All nodes agree on a single value; Value cannot change later
- Guarantees – Termination (Liveness), Agreement (Safety), Validity

Consensus impossible in Asynchronous Systems with a single faulty process

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.

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

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 proposer 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)

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

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

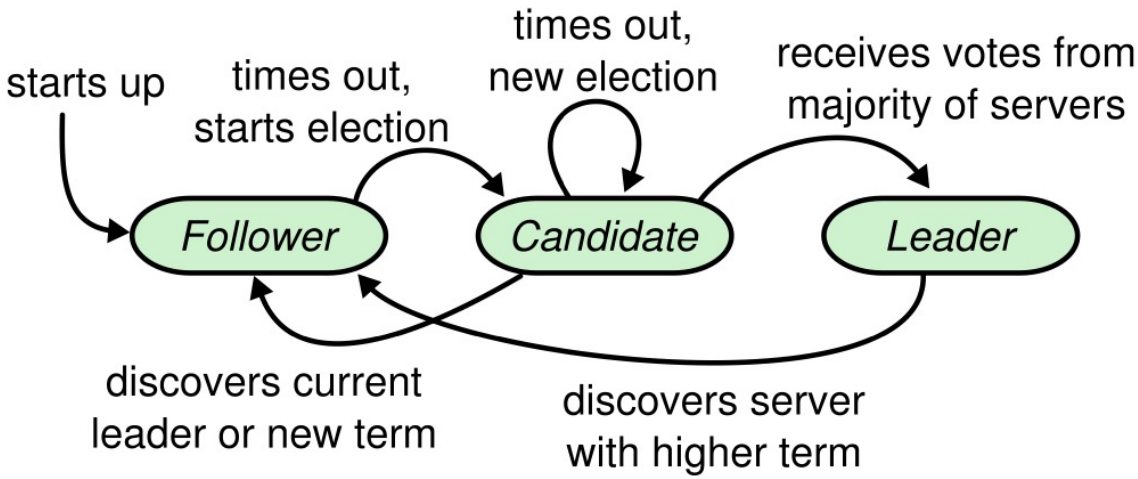
- Paxos – agrees separately on each client operation
- Raft – agrees on each new leader (and on tail of the log); agreement not required for most client operations

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

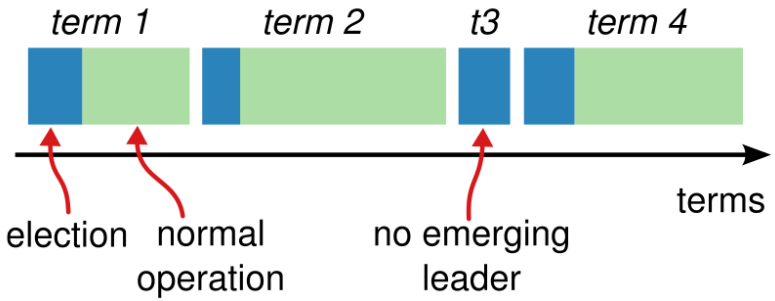
Leader Election

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

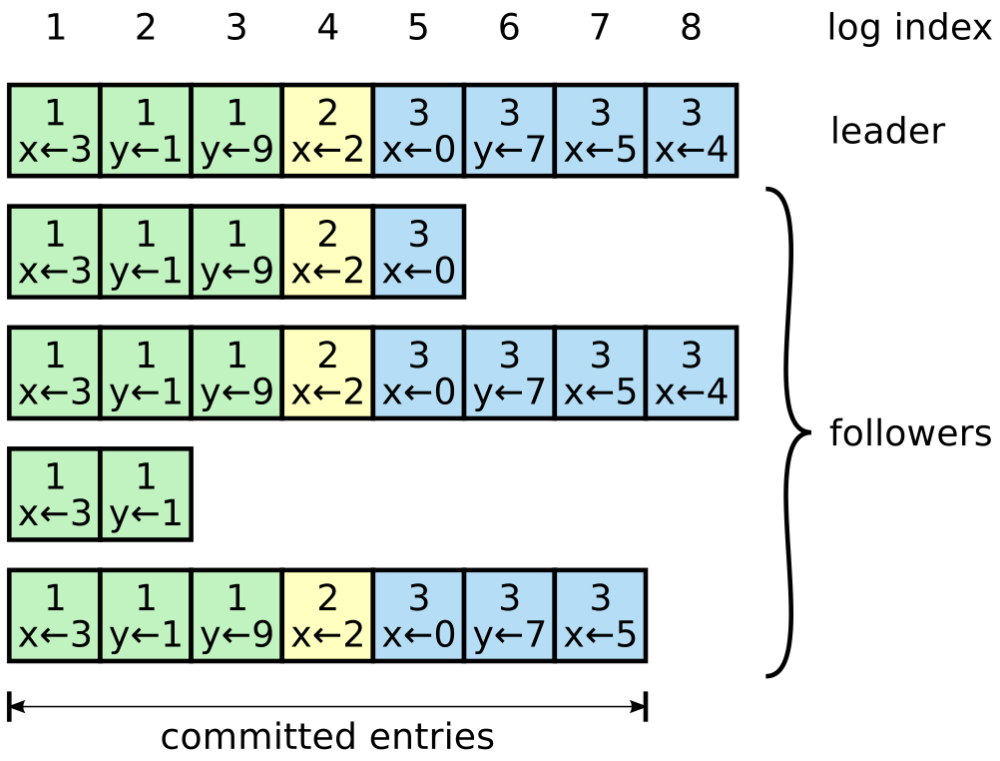
A leader's log is the ultimate truth

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



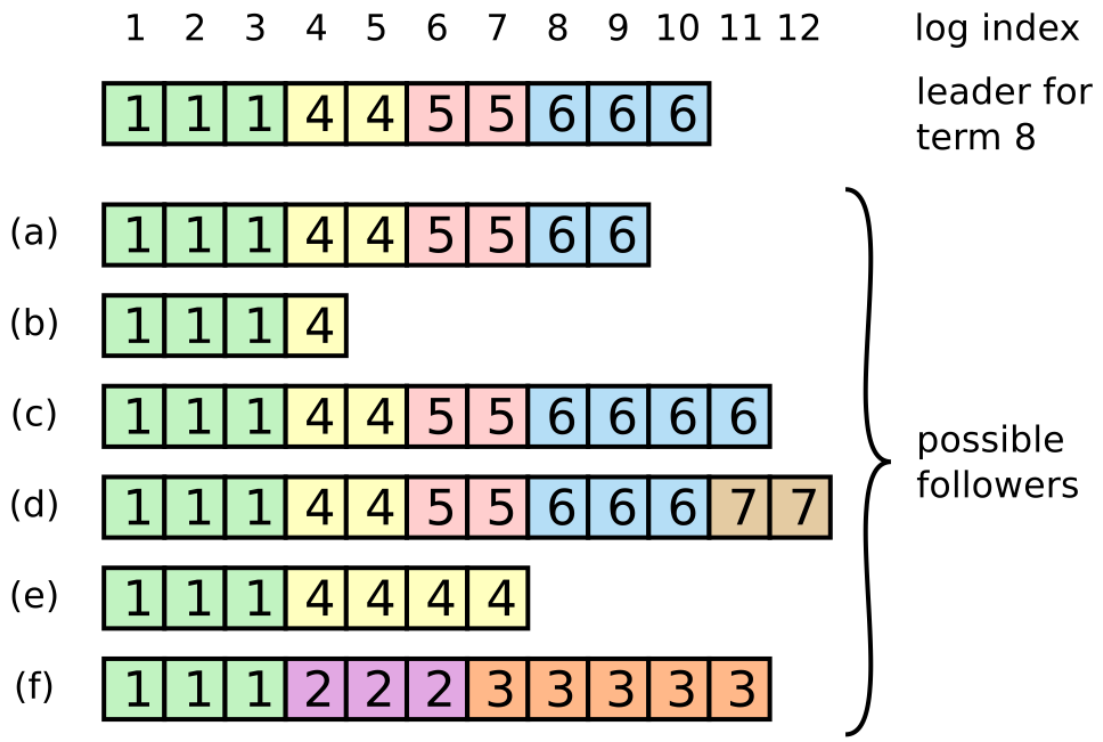
Log entries over time (…contd)

Different possibilities

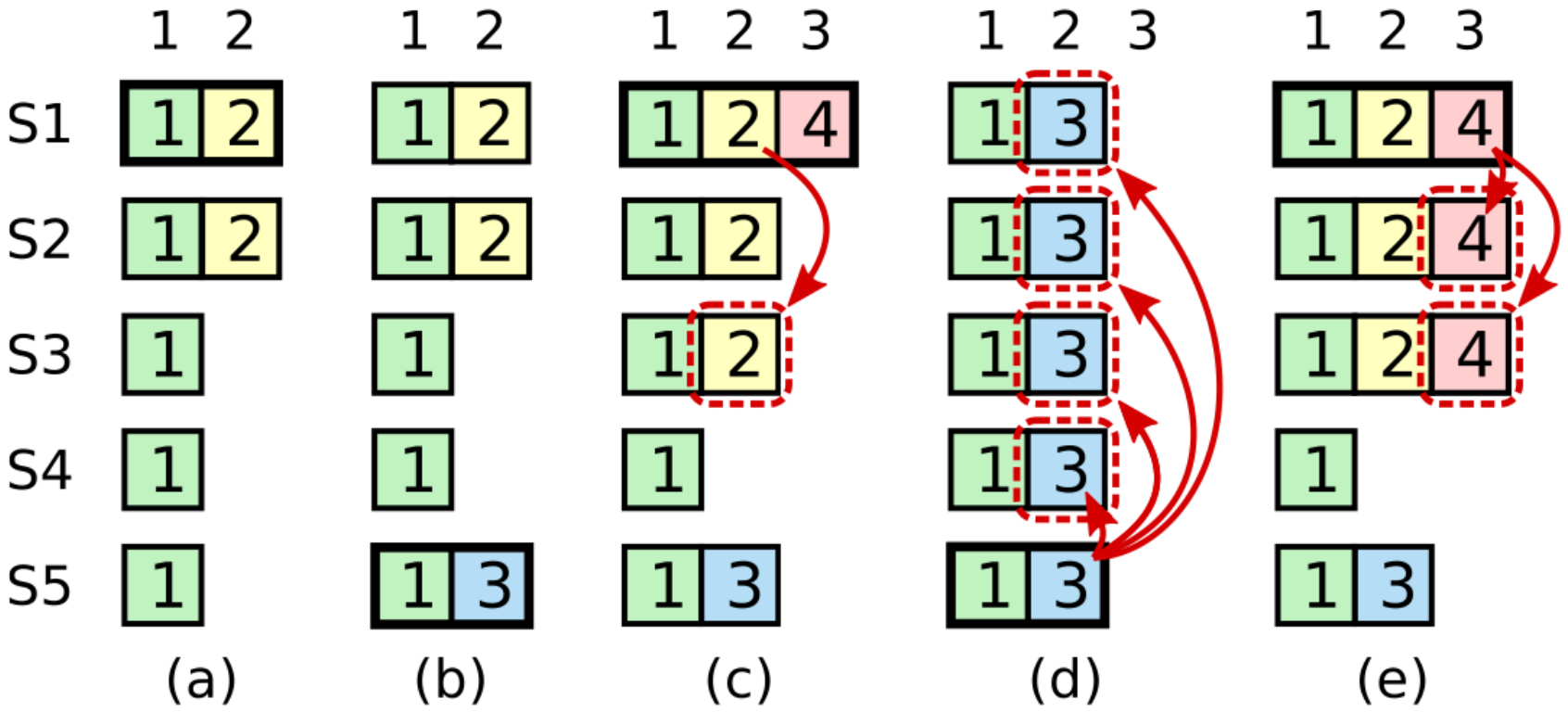
Missing entries → (a-b)

Extra uncommitted entries → (c-d)

Missing + Extra uncommitted entries → (e-f)



Committing Entries From Previous Terms



Why Mutual Exclusion Needed?

Why Mutual Exclusion Needed? (contd…)

Distributed Systems

Cannot share local lock variables

How do we support mutual exclusion in a distributed system?

Let us start simple with a central solution

Distributed Lock – Central Solution

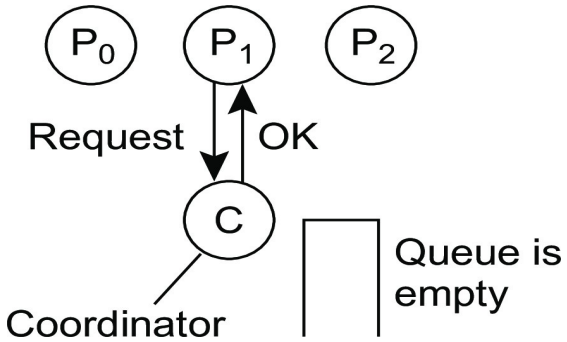
System Model

- Each pair of nodes is connected by reliable channels
- Messages are eventually delivered to recipient, and in FIFO order
- Nodes do not fail

Central Solution

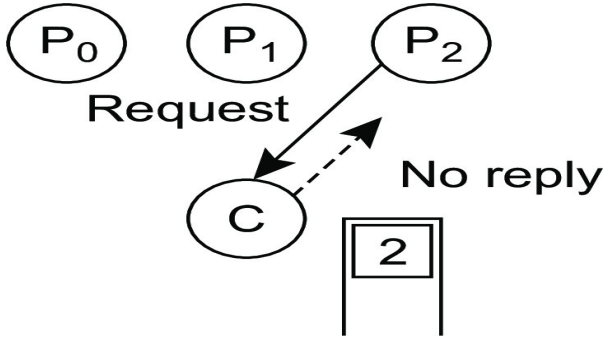
- Elect a central leader using election algorithm
- Leader keeps a queue of waiting requests from nodes who wish to access CS

Distributed Lock – Central Solution (contd…)



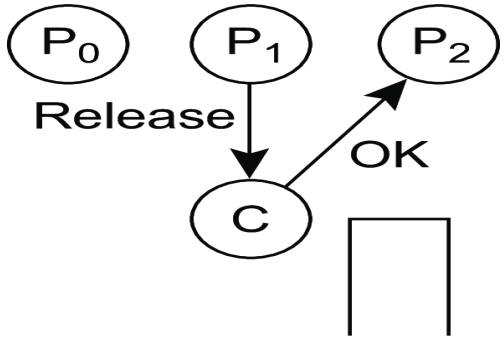
Step 1

Node P1 asks the coordinator for permission to access a shared resource. Permission is granted.



Step 2

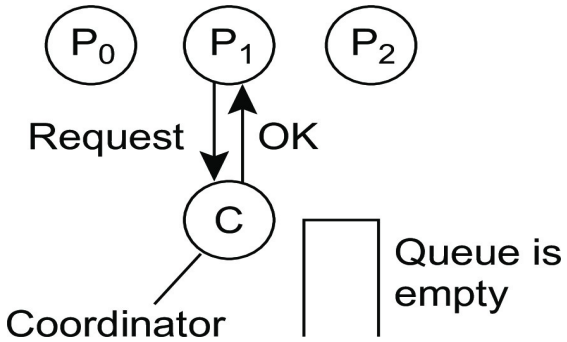
Node P2 then asks permission to access the same resource. The Coordinator does not reply.



Step 3

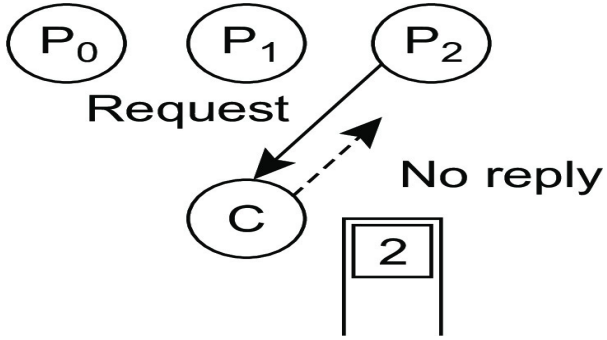
When P1 releases the resource, it tells the coordinator, which then replies to P2.

Distributed Lock – Central Solution (contd…)



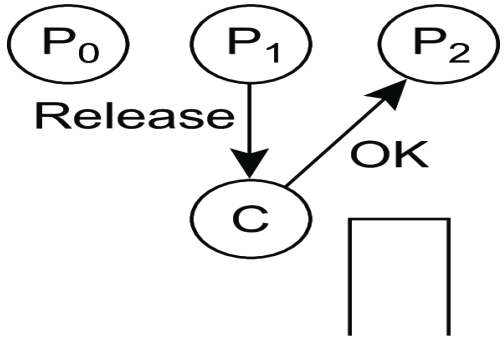
Step 1

Node P1 asks the coordinator for permission to access a shared resource. Permission is granted.



Step 2

Node P2 then asks permission to access the same resource. The Coordinator does not reply.



Step 3

When P1 releases the resource, it tells the coordinator, which then replies to P2.

Problems????

Distributed Solution

Decentralized approach

- All nodes involved in the decision making of who should access the resource

Ricart-Agarwala Algorithm

Use the notion of causality – rely on logical timestamps

Ricart-Agrawala Algorithm

Requestor

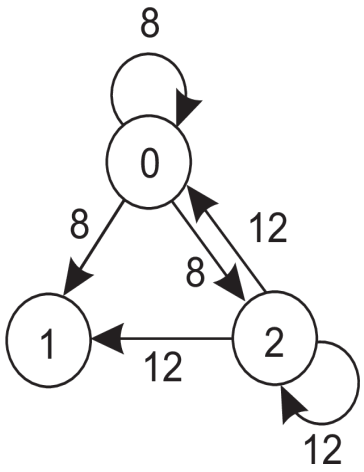
1. Broadcast a message to all receiver (including itself)
   <Resource-Name, Node-Name, Logical Timestamp>

3. Wait for all the OK messages.
4. Access resources once all receivers send OK message.
5. Release the resource; Send OK message to all queue entries

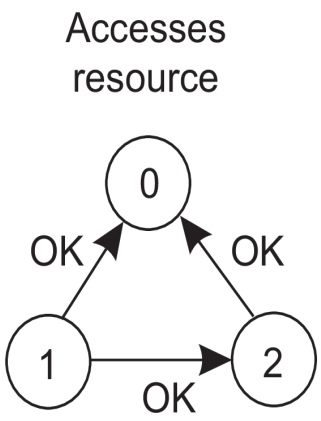
Receiver

2. If receiver not accessing the resource or does not want to access it, send OK message to the sender.
   If the receiver already has access to the resource. Do not reply. Queue the request.
   If receiver wants to access the resource but has not yet done, compare the timestamp
   - If incoming message has lower timestamp:
     - send OK message to the sender
   - Else:
     - Queue the incoming request and send nothing

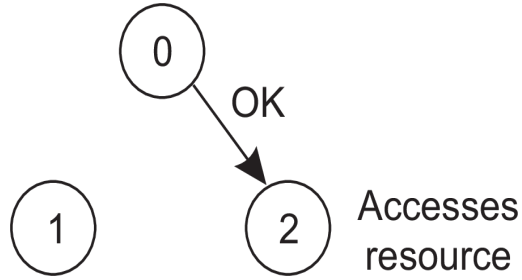
Ricart-Agrawala Algorithm Example



Step 1



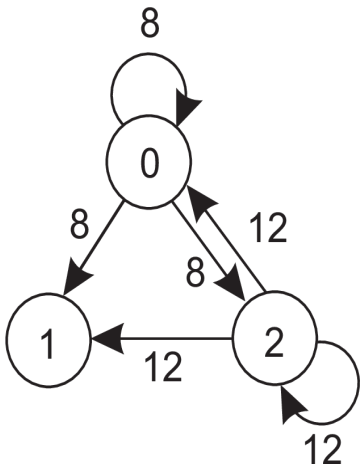
Step 2



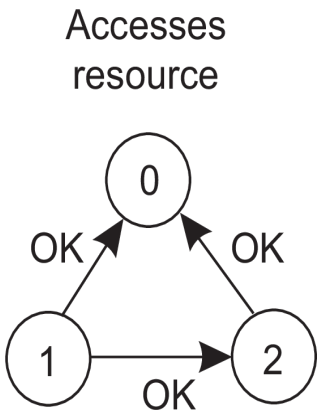
Step 3

Step 1: Two nodes want to access a shared resource at the same moment.
Step 2: P0 has the lowest timestamp, so it wins.
Step 3: When process P0 is done, it sends an OK message to P2. P2 can access the resource thereafter.

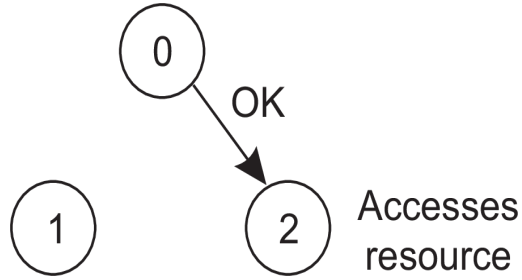
Ricart-Agrawala Algorithm Example



Step 1



Step 2



Step 3

Step 1: Two nodes want to access a shared resource at the same moment.
Step 2: P0 has the lowest timestamp, so it wins.
Step 3: When process P0 is done, it sends an OK message to P2. P2 can access the resource thereafter.

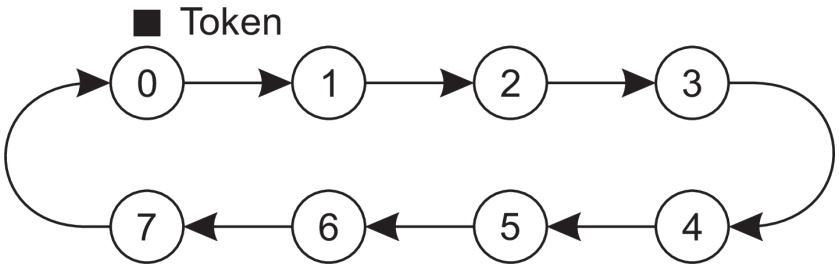
Problems????

Token Ring Algorithm

All nodes arranged in a ring fashion

Use token as a means of ownership

- Whosoever has the token can access the resource
- If no access needed, pass it on to the neighbor
- Token gets passed to all the nodes



Problems????