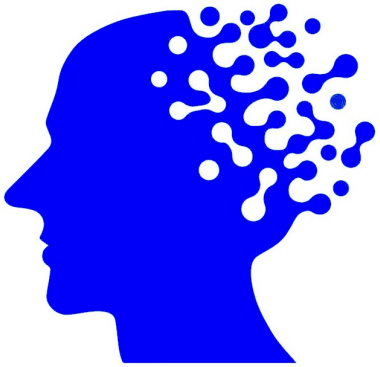


Spring 2024

Distributed Computing

- Basics of Mutual Exclusion algorithms

Dr. Rajendra Prasath

Indian Institute of Information Technology Sri City, Chittoor

11th March 2024 (http://rajendra.2power3.com)

> Distributed Computing?

- How will you design a Distributed Algorithm?

- Learn to Solve using Distributed Algorithms

Recap: Distributed Systems

A Distributed System:

- A collection of independent systems that appears to its users as a single coherent system
- A system in which hardware and software components of networked computers communicate and coordinate their activity only by passing messages
- A computing platform built with many computers that:
  - Operate concurrently
  - Are physically distributed (have their own failure modes)
  - Are linked by a network
  - Have independent clocks

Recap: Characteristics

- Concurrent execution of processes:
  - Non-determinism, Race Conditions, Synchronization, Deadlocks, and so on
- No global clock
  - Coordination is done by message exchange
  - No Single Global notion of the correct time
- No global state
  - No Process has a knowledge of the current global state of the system
- Units may fail independently
  - Network Faults may isolate computers that are still running
  - System Failures may not be immediately known

What did you learn so far?

- Goals / Challenges in Message Passing systems
- Distributed Sorting / Space-Time diagram
- Partial Ordering / Total Ordering
- Concurrent Events / Causal Ordering
- Logical Clocks vs Physical Clocks
- Global Snapshot Detection
- Termination Detection Algorithm
- Leader Election in Rings
- Topology Abstraction and Overlays
- Message Ordering and Group Communication

- Mutual Exclusion Algorithm

[Now] → → →

Recap

> About this Lecture

What do we learn today?

- Recap: Message Ordering and GC
  - Models of Communication
  - Design Issues / Good / Bad Ordering

- Mutual Exclusion Algorithms
  - Centralized Algorithm
  - Token-Based / Permission-Based Algorithms
  - Quorum-Based Algorithm
  - Tree-Based Algorithm

Let us explore these topics ➔ ➔ ➔

Today's Focus

Distributed Mutual Exclusion Algorithms

Why do we need MutEx?

- Mutual Exclusion
  - Operating systems: Semaphores
    - In a single machine, you could use semaphores to implement mutual exclusion
    - How to implement semaphores?
      - Inhibit interrupts
      - Use clever instructions (e.g. test-and-set)
    - On a multiprocessor shared memory machine, only the latter works

Characteristics

- Processes communicate only through messages – no shared memory or no global clocks
- Processes must expect unpredictable message delays
- Processes coordinate access to shared resources (printer, file, etc.) that should only be used in a mutually exclusive manner.

Race Conditions

- Consider Online systems – For example, Airline reservation systems maintain records of available seats
- Suppose two people buy the same seat, because each checks and finds the seat available, then each buys the seat
- Overlapped accesses generate different results than serial accesses – race condition

Distributed Mutual Exclusion

- Needs
  - Only one process should be in critical section at any point of time
  - What about resources?

Distributed Mutual Exclusion

- No Deadlocks – no set of sites should be permanently blocked, waiting for messages from other sites in that set
- No starvation – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
- Fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation.)
- Fault Tolerance – the algorithm is able to survive a failure at one or more sites

Distributed MutEx – An overview

Token-based solution: Processes share a special message known as a token

- ➔ Token holder has right to access shared resource
- ➔ Wait for/ask for (depending on algorithm) token; enter Critical Section (CS) when it is obtained, pass to another process on exit or hold until requested (depending on algorithm)
- ➔ If a process receives the token and doesn't need it, just pass it on

Distributed MutEx – A Few Issues

- Who can access the resource?
- When does a process to be privileged to access the resource?
- How long does a process access the resource? Any finite duration?
- How long can a process wait to be privileged?
- Computation complexity of the solution

Types of Distributed MutEx

- Token-based distributed mutual exclusion algorithms
  - Suzuki – Kasami's Algorithm
- Non-token based distributed mutual exclusion algorithms
  - Lamport's Algorithm
  - Ricart-Agartala's Algorithm

Token Based Methods

Advantages:

- Starvation can be avoided by efficient organization of the processes
- Deadlock is also avoidable

Disadvantage: Token Loss

- Must initiate a cooperative procedure to recreate the token
- Must ensure that only one token is created!

Permission-Based Methods

- **Non-Token Based solutions:** a process that wishes to access a shared resource must first get permission from one or more other processes.

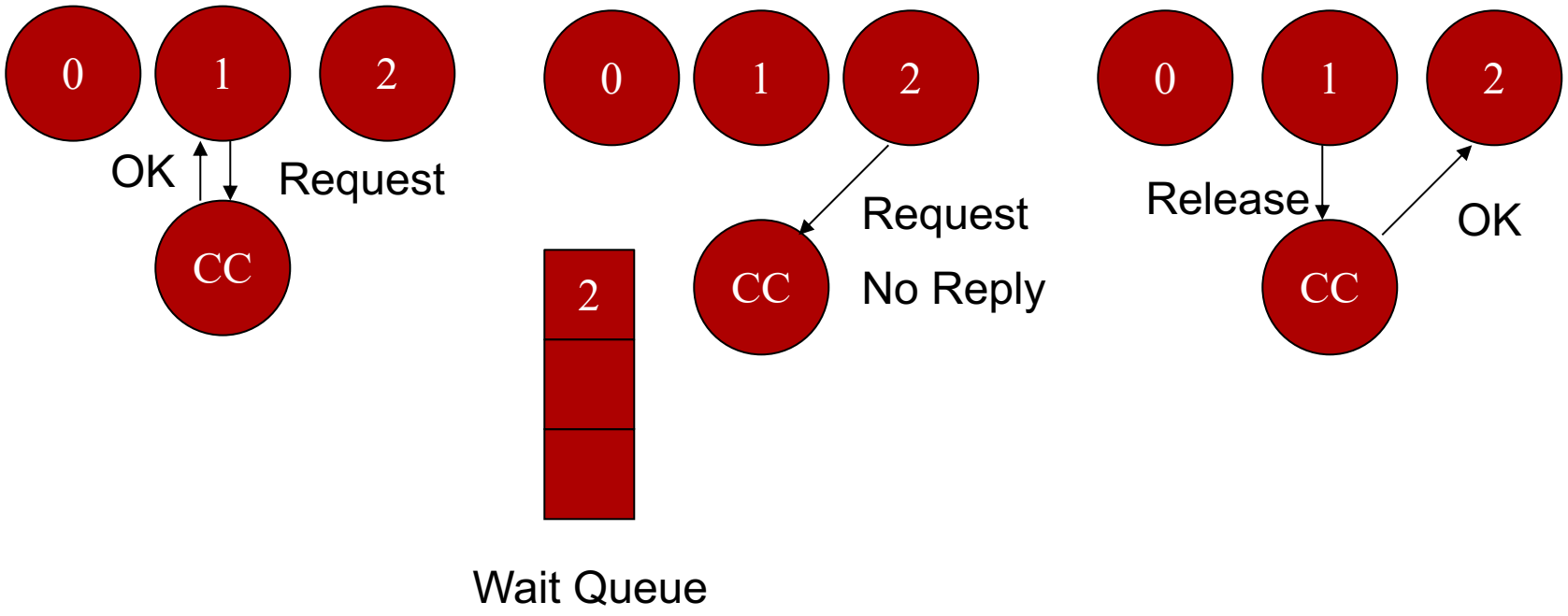
- Avoids the problems of token-based solutions, but is more complicated to implement

Basic Algorithms

- Centralized
- Decentralized
- Distributed
  - Distributed with "voting" – for increased fault tolerance
- Token Ring

Centralized Mutual Exclusion

➔ Central coordinator manages the FIFO queue of requests to guarantee "no starvation"



Centralized Control of MutEx

- A central coordinator (master or leader)
- Is elected (which algorithm?)
- Grants permission to enter CS & keeps a queue of requests to enter the CS.
- Ensures only one process at a time can access the CS
- Has a special token message, which it can give to any process to access CS

Centralized Control - Operations

- To enter a CS, send a request to the coordinator & wait for token.
- On exiting the CS, send a message to the coordinator to release the token.
- Upon receipt of a request, if no other process has the token, the coordinator replies with the token; otherwise, the coordinator queues the request
- Upon receipt of a release message, the coordinator removes the oldest entry in the queue (if any) and replies with a token

Centralized Control - Features

- Safety, Liveness are guaranteed
- Ordering also guaranteed (what kind?)
- Requires 2 messages for entry + 1 messages for exit operation.
- Client delay: one round trip time (request + grant)
- Synchronization delay: 2 message latencies (release + grant)
- The coordinator becomes performance bottleneck and single point of failure

Decentralized MutEx

- More fault-tolerant than centralized approach
- Uses the Distributed Hash Table (DHT) approach to locate objects/replicas
  - Object names are hashed to find the node where they are stored (succ function)
- n replicas of each object are placed on n successive nodes
  - Hash object name to get addresses
- Now every replica has a coordinator that controls access

The Decentralized Algorithm

- Coordinators respond to requests at once:

  Yes OR No

- Majority: To use the resource, a process must receive permission from m > n/2 coordinators
  - If the requester gets fewer than m votes, it will wait for a random time and then ask again
- If a request is denied, or when the CS is completed, notify the coordinators who have sent OK messages, so they can respond again to another request. (Why is this important?)

The Decentralized Algo - Analysis

- More robust than the central coordinator approach. If one coordinator goes down others are available.
  - If a coordinator fails and resets then it will not remember having granted access to one requestor, and may then give access to another.
  - It is highly unlikely that this will lead to a violation of mutual exclusion

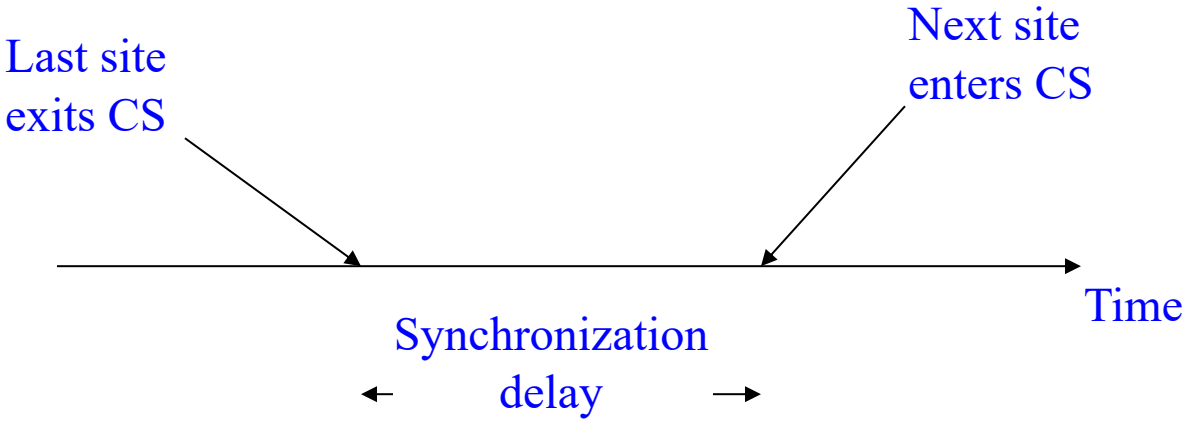
The Decentralized Algorithm - Issues

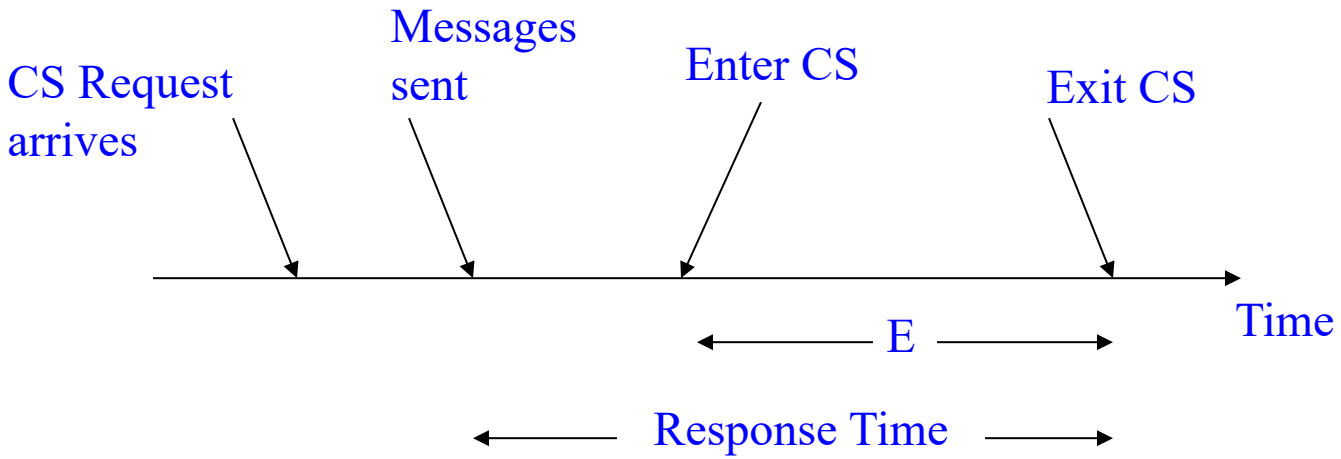
- If a resource is in high demand, multiple requests will be generated by different processes.
- High level of contention
- Processes may wait a long time to get permission - Possibility of starvation exists
- Resource usage drops.

Performance Analysis

- Guarantees mutual exclusion
- No starvation: Only if requests served in order
- No deadlock
- Fault tolerant?
  - Single point of failure
  - Blocking requests mean client processes have difficulty distinguishing crashed coordinator from long wait
  - Bottlenecks
- The solution is simple and ease

Performance Metrics





Performance - Analysis

- Number of messages per CS invocation: should be minimized
- Synchronization delay, i.e., time between the leaving of CS by a site and the entry of CS by the next one: should be minimized
- Response time: time interval between request messages transmissions and the exit of CS
- System throughput, i.e., rate at which system executes the requests for CS: should be maximized
- If d is the synchronization delay, e the average CS execution time:

$$\text{System Throughput} = 1 / (d + e)$$

Performance – Analysis (contd)

- Low and High Load:
  - Low load: No more than one request at a given point in time
  - High load: Always a pending mutual exclusion request at a site
- Best and Worst Case:
  - Best Case (low loads): Round-trip message delay + Execution time - 2T + E
  - Worst case (high loads)
- Message traffic: low at low loads, high at high loads
- Average performance: when load conditions fluctuate widely

Token Ring Approach

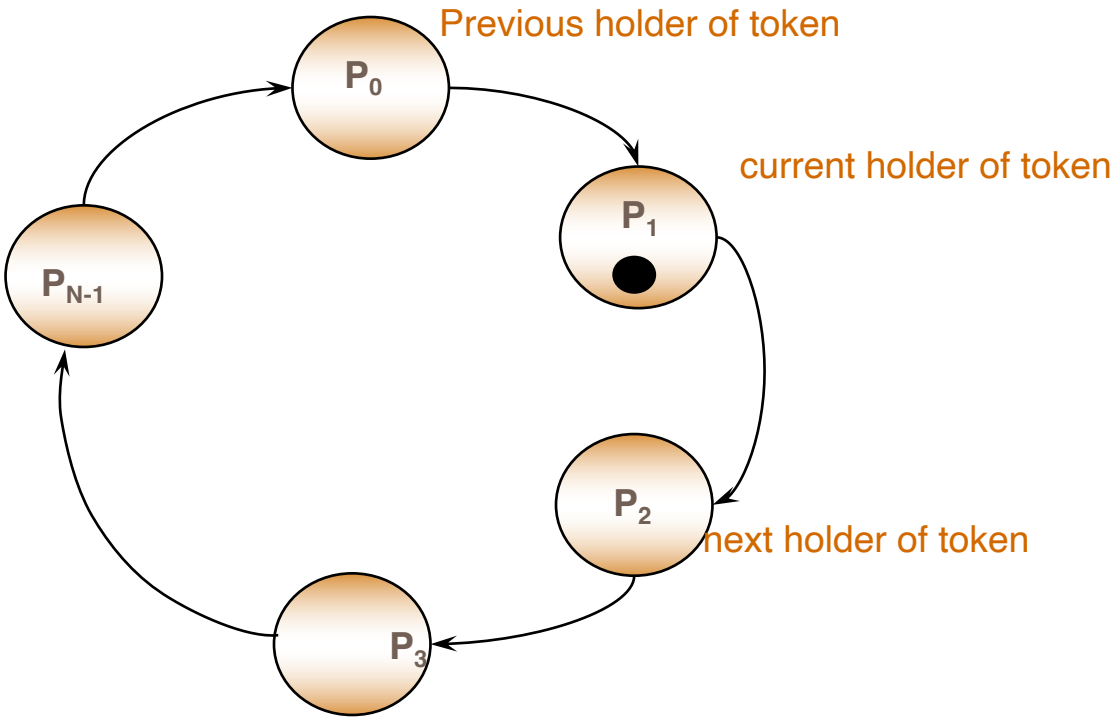
- Processes are organized in a logical ring: P_i has a communication channel to $P_{(i+1) mod\ N}$

Operations:

- Only the process holding the token T can enter the CS
- To enter the critical section, wait passively for T When in CS, hold on to T and don't release it
- To exit the CS, send T onto your neighbor
- If a process does not want to enter the CS when it receives T, it simply forwards T to the next neighbor

Token Rings – Illustration

➔ **Request movements in an unidirectional ring network**



Token Rings - Features

- Safety & Liveness are guaranteed
- Ordering is not guaranteed
- Bandwidth: 1 message per exit
- Client delay: 0 to N message transmissions
- Synchronization delay between one process's exit from the CS and the next process's entry is between 1 and N-1 message transmissions

Summary

- Racap: Message Ordering & Group Communications
  - Good / Bad Ordering
  - Causal Ordering

- Distributed Mutual Exclusion Algorithms
  - Mutual Exclusion Problem
    - Basics of MutEx algorithms
    - Types of MutEx algorithms
    - Centralized Approach
    - Token-based / Permission-based algorithms
    - Token RingsPerformance Metrics

Many more to come up … ! Stay tuned in !!

Penalties

- Every Student is expected to strictly follow a fair Academic Code of Conduct to avoid penalties

- Penalties is heavy for those who involve in:
  - Copy and Pasting the code
  - Plagiarism (copied from your neighbor or friend – in this case, both will get "0" marks for that specific take home assignments)
  - If the candidate is unable to explain his own solution, it would be considered as a "copied case"!!
  - Any other unfair means of completing the assignments

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

How to reach me?

➔ Please leave me an email:

rajendra [DOT] prasath [AT] iiits [DOT] in

➔ Visit my homepage @

➔ https://www.iiits.ac.in/people/regular-faculty/dr-rajendra-prasath/

(OR)

➔ http://rajendra.2power3.com

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- You may ask for one-to-one meeting

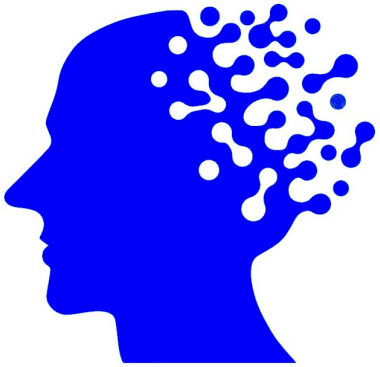
Best Approach

- You may leave me an email any time
  (email is the best way to reach me faster)

Questions
It's Your Time

Contact Information

Dr. Rajendra Prasath
IIIT Sri City, Chittoor



Spring 2024

Distributed Computing

- Token-Based Mutual Exclusion algos

Dr. Rajendra Prasath

Indian Institute of Information Technology Sri City, Chittoor

12th March 2024 (http://rajendra.2power3.com)

> Distributed Computing?

- How will you design a Distributed Algorithm?

- Learn to Solve using Distributed Algorithms

Recap: Distributed Systems

A Distributed System:

- A collection of independent systems that appears to its users as a single coherent system
- A system in which hardware and software components of networked computers communicate and coordinate their activity only by passing messages
- A computing platform built with many computers that:
  - Operate concurrently
  - Are physically distributed (have their own failure modes)
  - Are linked by a network
  - Have independent clocks

Recap: Characteristics

- Concurrent execution of processes:
  - Non-determinism, Race Conditions, Synchronization, Deadlocks, and so on
- No global clock
  - Coordination is done by message exchange
  - No Single Global notion of the correct time
- No global state
  - No Process has a knowledge of the current global state of the system
- Units may fail independently
  - Network Faults may isolate computers that are still running
  - System Failures may not be immediately known

What did you learn so far?

- Goals / Challenges in Message Passing systems
- Distributed Sorting / Space-Time diagram
- Partial Ordering / Total Ordering
- Concurrent Events / Causal Ordering
- Logical Clocks vs Physical Clocks
- Global Snapshot Detection
- Termination Detection Algorithm
- Leader Election in Rings
- Topology Abstraction and Overlays
- Message Ordering and Group Communication

- Mutual Exclusion Algorithm

[Now] → → →

Recap

> About this Lecture

What do we learn today?

- Mutual Exclusion Algorithms
  - Centralized Algorithm
  - Token-Based / Permission-Based Algorithms
  - Quorum-Based Algorithm
  - Tree-Based Algorithm

Let us explore these topics ➔ ➔ ➔

Today's Focus

Distributed Mutual Exclusion – Token-Based MutEx Algorithms

Recap: The need for MutEx?

- Mutual Exclusion
  - Operating systems: Semaphores
    - In a single machine, you could use semaphores to implement mutual exclusion
    - How to implement semaphores?
      - Inhibit interrupts
      - Use clever instructions (e.g. test-and-set)
    - On a multiprocessor shared memory machine, only the latter works

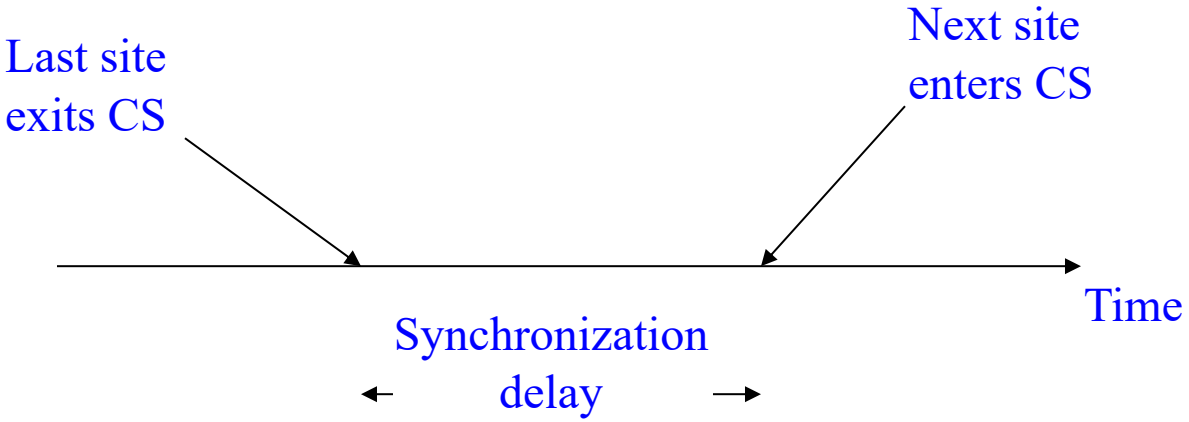
Characteristics

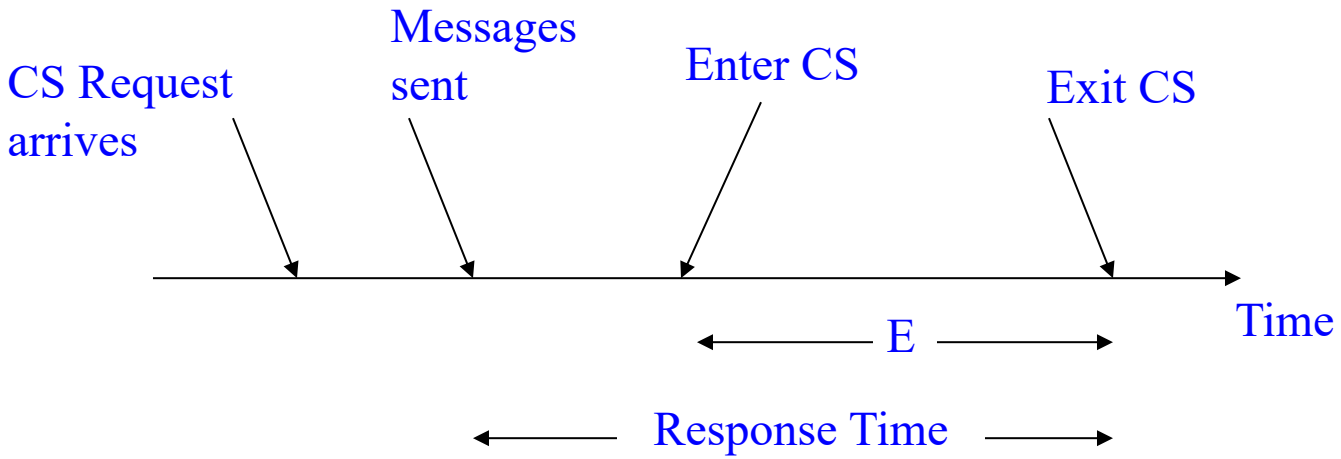
- Processes communicate only through messages – no shared memory or no global clocks
- Processes must expect unpredictable message but finite delays
- Processes coordinate access to shared resources that should only be used in a mutually exclusive manner.

Recap: Distributed MutEx

- No Deadlocks – no set of sites should be permanently blocked, waiting for messages from other sites in that set
- No starvation – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
- Fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation.)
- Fault Tolerance – the algorithm is able to survive a failure at one or more sites

Recap: Performance Metrics





Token-Based MutEx Algorithms

Token-based solution: Processes share a special message known as a token

- ➔ Token holder has right to access shared resource
- ➔ Wait for/ask for (depending on algorithm) token; enter Critical Section (CS) when it is obtained, pass to another process on exit or hold until requested (depending on algorithm)
- ➔ If a process receives the token and doesn't need it, just pass it on

Distributed MutEx – A Few Issues

- Who can access the resource?
- When does a process to be privileged to access the resource?
- How long does a process access the resource? Any finite duration?
- How long can a process wait to be privileged?
- Computation complexity of the solution

Types of Distributed MutEx

- Token-based distributed mutual exclusion algorithms
  - Suzuki – Kasami's Algorithm
  - Feuerstein et al's Algorithm

- Non-token based distributed mutual exclusion algorithms
  - Lamport's Algorithm
  - Ricart-Agrawala's Algorithm

Token Based Methods

Advantages:

- ➔ Starvation can be avoided by efficient organization of the processes
- ➔ Deadlock is also avoidable

Disadvantage: Token Loss

- ➔ Must initiate a cooperative procedure to recreate the token
- ➔ Must ensure that only one token is created!

Token Ring Approach

- Processes are organized in a logical ring: P_i has a communication channel to $P_{(i+1) mod\ N}$

Operations:

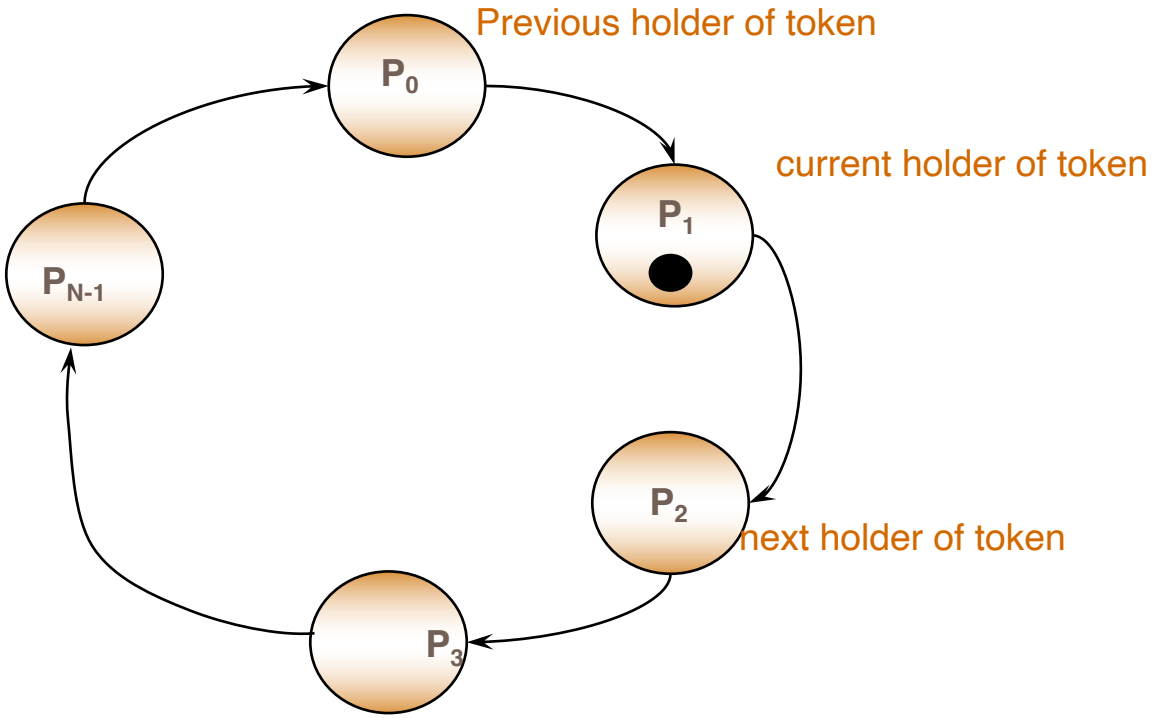
- Only the process holding the token T can enter the CS
- To enter the critical section, wait passively for T When in CS, hold on to T and don't release it
- To exit the CS, send T onto your neighbor
- If a process does not want to enter the CS when it receives T, it simply forwards T to the next neighbor

Token Ring - Idea

- Single token circulates, enter CS when token is present
- Mutual exclusion obvious
- Algorithms differ in how to find and get the token
- Uses sequence numbers rather than timestamps to differentiate between old and current requests

Token Rings – Illustration

➔ **Request movements in an unidirectional ring network**



Suzuki – Kasami's Algorithm

- Broadcast a request for the token
- Process with the token sends it to the requestor if it does not need it

- Issues:
  - Current versus outdated requests
  - Determining sites with pending requests
  - Deciding which site to give the token to

Data Structures

The token:

- Queue (FIFO) Q of requesting processes
- LN[1.. n] : sequence number of request that j executed most recently

The request message:

- REQUEST(i, k): request message from node i for its k^{th} critical section execution

Other data structures:

- RN_i [1.. n] for each node i, where , RN_i [j] is the largest sequence number received so far by i in a REQUEST message from j

Suzuki-Kasami's algorithm

To request critical section:

- If i does not have token, increment $RN_i[\ i\ ]$ and send REQUEST(i, $RN_i[\ i\ ]$) to all nodes
- If i has token already, enter critical section if the token is idle (no pending requests), else follow rule to release critical section

On receiving REQUEST(i, s_n) at) j:

- Set $RN_j[\ i\ ] = \max(RN_j[\ i\ ], s_n)$
- If j has the token and the token is idle then
  - send it to i if $RN_j[\ i\ ] = LN[\ i\ ] + 1$
  - If token is not idle, follow rule to release critical section

Suzuki-Kasami's algorithm

To enter critical section:

- Enter CS if token is present

To release critical section:

- Set $LN[i] = RN_i[i]$
- For every node j which is not in Q (in token), add node j to Q if $RN_i[j] = LN[j] + 1$
- If Q is non empty after the above, delete first node from Q and send the token to that node

Complexity

- No. of messages:
  - 0 if node holds the token already,
  - n otherwise

- Synchronization delay:
  - 0 (node has the token) or
  - max. message delay (token is elsewhere)

- No starvation

Token Based Control in Rings

- Requests move in either Clockwise or Anticlockwise
- Proposed by Feuerstein et al. (1996)
- There are 3 steps:
  - P needs the resource
    - P has T: enter CS
    - P has no T: send the request to the next P
  - P receives a request
    - P has T: increase TC by 1 and send T to the next P
    - P has no T: send request to the next P
  - P receives Token
    - P has a pending request: enter CS and decrease TC by 1 and send T to next P if TC > 0
    - P has no pending request: send T to the next P

Refer to: Feuerstein et al., Efficient token-based control in rings, Information Processing Letters, 66 (4) (1998) pp. 175-180

Token Based Control In Rings

- When p needs the resource then
  - If p has T then it enters the critical section.
  - If p has not T then it sends a request message to the next processor
- When p receives a request message then:
  - If p has T then it increases the counter and passes T to the next processor.
  - If p has not T then the request message is passed to the next processor.
- When p receives T then:
  - If p has a pending request then it enters the critical section in which the token counter is decreased by 1; at the exit of the critical section p passes T to the next processor if the token counter is greater than 0
  - If p has not a pending request then T is passed to the next processor in the ring.

Read from: E. Feuerstein et al. Efficient token-based control in rings, Information Processing Letters, 66 (1998) pp. 175-180, doi: 10.1016/S0020-0190(98)00054-4

Token Rings - Features

- Safety & Liveness are guaranteed
- Ordering is not guaranteed
- Bandwidth: 1 message per exit
- Client delay: 0 to N message transmissions
- Synchronization delay between one process's exit from the CS and the next process's entry is between 1 and N-1 message transmissions

Non-Token Based Methods

- **Permission-based solutions:** a process that wishes to access a shared resource must first get permission from one or more other processes.

- Avoids the problems of token-based solutions, but is more complicated to implement

Non-Token Based Algorithms

- Notations:
  - P_i: i th Process
  - R_i: Request set, containing IDs of all P_i s from which permission must be received before accessing CS
  - Non-token based approaches use time stamps to order requests for CS
  - Smaller time stamps get priority over larger ones
- Lamport's Algorithm
  - $R_i = \{P_1, P_2, \ldots, P_n\}$, i.e., all processes.
  - Request queue: maintained at each P_i ordered by time stamps.
  - Assumption: message delivered in FIFO

Lamport's Algorithm

- **Requesting CS:**
  - Send REQUEST(ts_i, i) where (ts_i, i) - Request time stamp; Place REQUEST in $request\_queue_i$
  - On receiving the message; P_j sends time-stamped REPLY message to P_i; Pi 's request placed in $request\_queue_j$
- **Executing CS:**
  - P_i has received a message with time stamp larger than (ts_i, i) from all other sites
  - P_i's request is the top most one in $request\_queue_i$
- **Releasing CS:**
  - Exiting CS: send a time stamped RELEASE message to all sites in its request set
  - Receiving RELEASE message: P_j removes P_i 's request from its queue

Notable Points

- Purpose of REPLY messages from i to j is to ensure that j knows of all requests of i prior to sending the REPLY (possibly any request of i with timestamp lower than j 's request)
- Requires FIFO channels
- 3(n – 1) messages per critical section invocation
- Synchronization delay = max msg transmission time
- Requests are granted in order of increasing timestamps

Performance Improvements

- 3(n-1) messages per Critical Section invocation
  - (n - 1) REQUEST messages
  - (n - 1) REPLY messages
  - (n - 1) RELEASE messages
- Synchronization delay: T
- Optimization:
  - Suppress reply messages: For example, P_j receives a REQUEST message from P_i after sending its own REQUEST message with time stamp higher than that of P_i 's then Do NOT send a REPLY message
  - Messages reduced to between 2(n-1) and 3(n-1)

Ricart & Agrawala's Algorithm

- A time-stamp based approach
- Originally proposed by Lamport using logical clocks
- Modified by Ricart & Agrawala

Ricart & Agrawala's Algorithm

Main Idea:

- Process j need not send a REPLY to Process i if j has a request with timestamp lower than the request of i (since i cannot enter before j here)
- Does not require FIFO
- 2(n – 1) messages per critical section invocation
- Synchronization delay = maximum message transmission time
- Requests granted in order of increasing timestamps

Ricart & Agrawala (contd)

- Processes need entry to critical section multicast a request, and can enter it only when all other processes have replied positively
- Messages requesting entry are of the form $<T, P_i>$
  - T - sender's timestamp (Lamport clock)
  - P_i the sender's identity

Ricart & Agrawala – Algorithm

To enter the Critical Section (CS):

- ➔ Set state = wanted
- ➔ multicast "request" to all processes (including timestamp)
- ➔ wait until all processes send back "reply"
- ➔ change state to held and enter the CS

On receipt of a request <T_j, P_j> at P_i:

- ➔ if (state == held) or (state == wanted & (T_i, P_i) < (T_j, P_j)) then enqueue the request
- ➔ else "reply" to P_j

On exiting the CS:

- ➔ change state to release and "reply" to all queued requests

Ricart & Agrawala – Simplified

To request Critical Section:

- ➔ send timestamped REQUEST message (ts_i, i)

On receiving request (ts_i, i) at j:

- ➔ if j is neither requesting nor executing critical section then send REPLY to i
- ➔ if j is requesting and i's request timestamp is smaller than j's request timestamp then
  - ➔ enqueue the request; Otherwise, defer the request

To enter Critical Section:

- ➔ Process i enters critical section on receiving REPLY messages from all processes

To release Critical Section:

- ➔ send REPLY to all deferred requests

Summary

- Recap: Distributed Mutual Exclusion Algorithms
  - Mutual Exclusion Problem
    - Basics of MutEx algorithms
    - Types of MutEx algorithms
      - Token-based Algorithm
        - Suzuki-Kasami's Algorithm
        - Feuerstein et al's Algorithm
      - Non-Token based Algorithm
        - Lamport's Algorithm
        - Ricart – Agrawala's Algorithm
    - Performance Metrics

Many more to come up … ! Stay tuned in !!

Penalties

- Every Student is expected to strictly follow a fair Academic Code of Conduct to avoid penalties

- Penalties is heavy for those who involve in:
  - Copy and Pasting the code
  - Plagiarism (copied from your neighbor or friend – in this case, both will get "0" marks for that specific take home assignments)
  - If the candidate is unable to explain his own solution, it would be considered as a "copied case"!!
  - Any other unfair means of completing the assignments

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

How to reach me?

➔ Please leave me an email:

rajendra [DOT] prasath [AT] iiits [DOT] in

➔ Visit my homepage @

➔ https://www.iiits.ac.in/people/regular-faculty/dr-rajendra-prasath/

(OR)

➔ http://rajendra.2power3.com

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- You may ask for one-to-one meeting

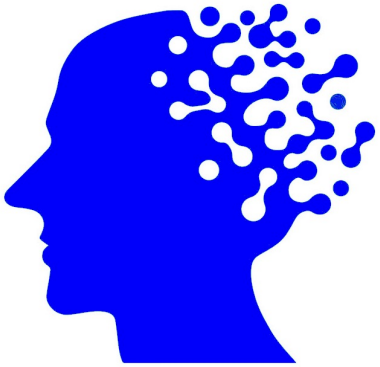
Best Approach

- You may leave me an email any time
  (email is the best way to reach me faster)

Questions
It's Your Time

Contact Information

Dr. Rajendra Prasath
IIIT Sri City, Chittoor



Spring 2024

Distributed Computing

- Non-Token-Based MutEx algorithms

Dr. Rajendra Prasath

Indian Institute of Information Technology Sri City, Chittoor

20th March 2024 (http://rajendra.2power3.com)

> Distributed Computing?

- How will you design a Distributed Algorithm?

- Learn to Solve using Distributed Algorithms

Recap: Distributed Systems

A Distributed System:

- A collection of independent systems that appears to its users as a single coherent system
- A system in which hardware and software components of networked computers communicate and coordinate their activity only by passing messages
- A computing platform built with many computers that:
  - Operate concurrently
  - Are physically distributed (have their own failure modes)
  - Are linked by a network
  - Have independent clocks

Recap: Characteristics

- Concurrent execution of processes:
  - Non-determinism, Race Conditions, Synchronization, Deadlocks, and so on
- No global clock
  - Coordination is done by message exchange
  - No Single Global notion of the correct time
- No global state
  - No Process has a knowledge of the current global state of the system
- Units may fail independently
  - Network Faults may isolate computers that are still running
  - System Failures may not be immediately known

What did you learn so far?

- Goals / Challenges in Message Passing systems
- Distributed Sorting / Space-Time diagram
- Partial Ordering / Total Ordering
- Concurrent Events / Causal Ordering
- Logical Clocks vs Physical Clocks
- Global Snapshot Detection
- Termination Detection Algorithm
- Leader Election in Rings
- Topology Abstraction and Overlays
- Message Ordering and Group Communication

- Mutual Exclusion Algorithm

[Now] → → →

Recap

> About this Lecture

What do we learn today?

- Mutual Exclusion Algorithms
  - Centralized Algorithm
  - Token-Based / Permission-Based Algorithms
  - Quorum-Based Algorithm
  - Tree-Based Algorithm

Let us explore these topics ➔ ➔ ➔

Today's Focus

Distributed Mutual Exclusion – Token / Non-Token-Based MutEx Algorithms

Recap: The need for MutEx?

- Mutual Exclusion
  - Operating systems: Semaphores
    - In a single machine, you could use semaphores to implement mutual exclusion
    - How to implement semaphores?
      - Inhibit interrupts
      - Use clever instructions (e.g. test-and-set)
    - On a multiprocessor shared memory machine, only the latter works

Characteristics

- Processes communicate only through messages – no shared memory or no global clocks
- Processes must expect unpredictable message but finite delays
- Processes coordinate access to shared resources that should only be used in a mutually exclusive manner.

Recap: Distributed MutEx

- No Deadlocks – no set of sites should be permanently blocked, waiting for messages from other sites in that set
- No starvation – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
- Fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation.)
- Fault Tolerance – the algorithm is able to survive a failure at one or more sites

Quorum Based algorithms

Why Quorum based algorithm?

- Lamports and Ricard-Agrawala' algorithm requires permission from all processes to enter into the critical section.

Modifications:

- Is it necessary to obtain permission from all processes before entering into the CS?
- How to reduce the message exchanges and increase the performance of MutEx algorithm?

Quorum Based algorithms

What is a Quorum?

- There are n requesting processes in a distributed system and any process may request for CS.
- Can we form such a subset of processes who request for Critical Section? YES !!
  - Such a set is said to be a Request Set or Quorum
  - In fact, we will have a separate Request set for each process P_i

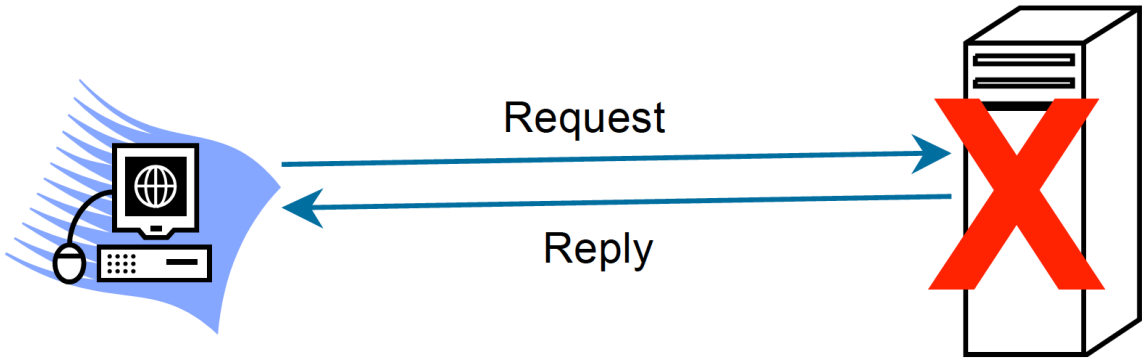
Quorum - Definition

- A quorum system is a collection of subsets of processes, called quorums, such that each pair of quorums have a non-empty intersection

- How do we formally define a quorum of processes in a distributed system?
- Let us look at some examples

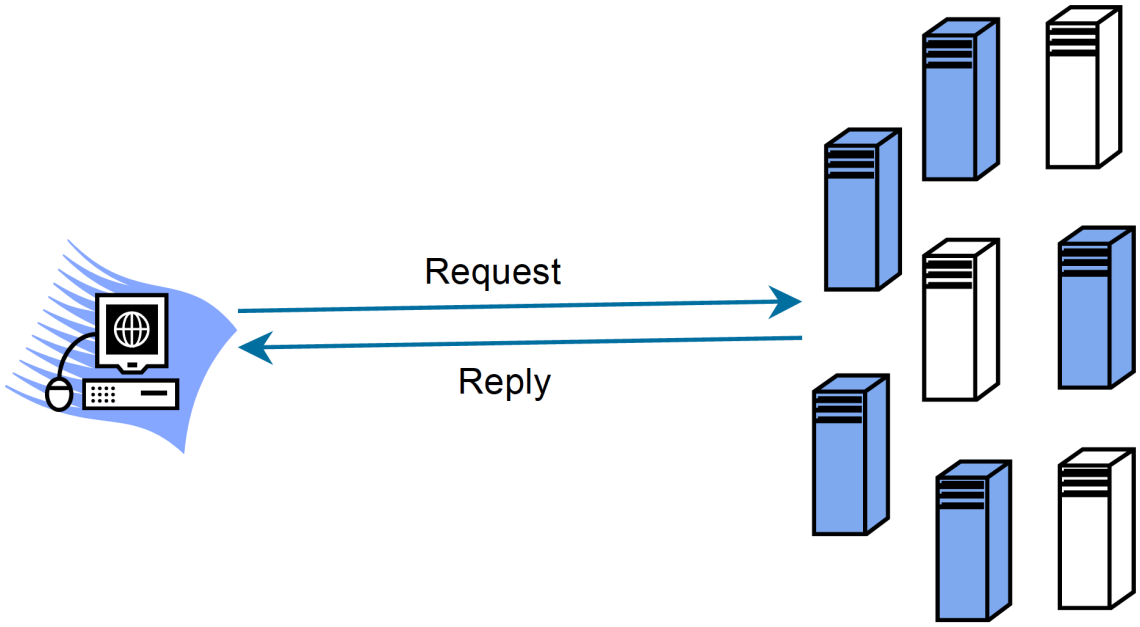
Quorum – Why?

- Process may not respond or may go down (any kind of failure)
- The requesting process can not get REPLY from all remaining processes
- It would infinitely wait for CS !!



Quorum – Why?

➔ Can the requesting process get permission from a quorum of processes to enter into CS?



Quorum - Definition

More Formally,

- Given a set of processes

$$P = \{P_1, P_2, \ldots, P_n\}$$

- A quorum system $Q \subseteq 2^P$ is a set of subsets of P such that

  for all Q_1, Q_2 in Q: $Q_1 \cap Q_2 \neq empty$

- Each Q_i in Q is called a quorum

Maekawa's Algorithm

- Permission obtained from only a subset of other processes, called the Request Set (or Quorum)

- Separate Request Set R_i, for each process i

Maekawa's Algorithm

Requirements

- For all i, j: $R_i \cap R_j \neq \Phi$
- For all i: $i \in R_i$
- For all i: $| R_i | = K$, for some K
- Any node i is contained in exactly D Request Sets, for some Request set D

- $K = D = sqrt(N)$ for Maekawa's algorithm

Maekawa's Algorithm - Steps

To Request Critical Section:

- P_i sends REQUEST message to all process in R_i

On receiving a REQUEST message:

- Send a REPLY message if no REPLY message has been sent since the last RELEASE message is received.
- Update status to indicate that a REPLY has been sent.
- Otherwise, queue up the REQUEST

To enter critical section:

- P_i enters critical section after receiving REPLY from all nodes in R_i

Maekawa's Algorithm – Steps (contd)

To release critical section:

- Send RELEASE message to all nodes in R_i
- On receiving a RELEASE message, send REPLY to next node in queue and delete the node from the queue.
- If queue is empty, update status to indicate no REPLY message has been sent

Computation Complexity

- Message Complexity: 3 * sqrt (N)
- Synchronization delay
  - 2*(max message transmission time)
- Major problem: DEADLOCK possible
- Need three more types of messages (FAILED, INQUIRE, YIELD) to handle deadlock.
  - Message complexity can be 5* sqrt(N N)
- Important Issue:
  - How to build the request sets?

Raymond's Algorithm

- Forms a directed tree (logical) with the token token-holder as root
- Each node has variable "Holder" that points to its parent on the path to the root.
  - Root's Holder variable points to itself
- Each node P_i has a FIFO request queue Q_i

Raymond's Algorithm

- To request critical section:
  - Send REQUEST to parent on the tree, provided i does not hold the token currently and Q_i is empty. Then place is request in Q_i

- When a non-root node j receives a request from k
  - place request in Q_j
  - send REQUEST to parent if no previous REQUEST sent

Raymond's Algorithm (contd)

When the root receives a REQUEST:

- ➔ send the token to the requesting node
- ➔ set Holder variable to point to that node

When a node receives the token:

- ➔ delete first entry from the queue
- ➔ send token to that node
- ➔ set Holder variable to point to that node
- ➔ if queue is non non-empty, send a REQUEST message to the parent (node pointed at by Holder variable)

Raymond's Algorithm (contd)

- To execute critical section:
  - enter if token is received and own entry is at the top of the queue; delete the entry from the queue
- To release critical section:
  - if queue is non non-empty, delete first entry from the queue, send token to that node and make Holder variable point to that node
  - If queue is still non non-empty, send a REQUEST message to the parent (node pointed at by Holder variable)

Features of Raymond's Algo

- Average message complexity:
  - O(log n)
- Sync. Delay
  - (T log n)/2, where T = max. message delay

Summary

- Recap: Distributed Mutual Exclusion Algorithms
  - Mutual Exclusion Problem
    - Basics of MutEx algorithms
    - Types of MutEx algorithms
      - Token-based Algorithms
        - Raymond's Tree based algorithm
      - Non-Token based Algorithms
        - Quorum based algorithm
    - Performance Metrics

Many more to come up … ! Stay tuned in !!

Penalties

- Every Student is expected to strictly follow a fair Academic Code of Conduct to avoid penalties

- Penalties is heavy for those who involve in:
  - Copy and Pasting the code
  - Plagiarism (copied from your neighbor or friend – in this case, both will get "0" marks for that specific take home assignments)
  - If the candidate is unable to explain his own solution, it would be considered as a "copied case"!!
  - Any other unfair means of completing the assignments

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

How to reach me?

➔ Please leave me an email:

rajendra [DOT] prasath [AT] iiits [DOT] in

➔ Visit my homepage @

➔ https://www.iiits.ac.in/people/regular-faculty/dr-rajendra-prasath/

(OR)

➔ http://rajendra.2power3.com

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- You may ask for one-to-one meeting

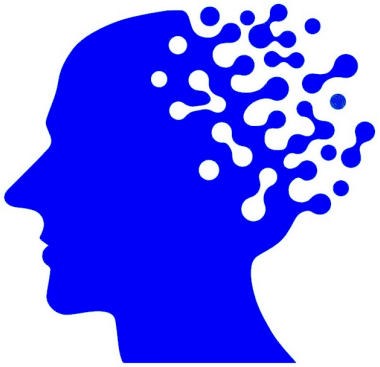
Best Approach

- You may leave me an email any time
  (email is the best way to reach me faster)

Questions
It's Your Time

Contact Information

Dr. Rajendra Prasath
IIIT Sri City, Chittoor



Spring 2024

Distributed Computing

- Deadlock Detection Algorithms

Dr. Rajendra Prasath

Indian Institute of Information Technology Sri City, Chittoor

21st and 22nd March 2024 (http://rajendra.2power3.com)

> Distributed Computing?

- How will you design a Distributed Algorithm?

- Learn to Solve using Distributed Algorithms

Recap: Distributed Systems

A Distributed System:

- A collection of independent systems that appears to its users as a single coherent system
- A system in which hardware and software components of networked computers communicate and coordinate their activity only by passing messages
- A computing platform built with many computers that:
  - Operate concurrently
  - Are physically distributed (have their own failure modes)
  - Are linked by a network
  - Have independent clocks

Recap: Characteristics

- Concurrent execution of processes:
  - Non-determinism, Race Conditions, Synchronization, Deadlocks, and so on
- No global clock
  - Coordination is done by message exchange
  - No Single Global notion of the correct time
- No global state
  - No Process has a knowledge of the current global state of the system
- Units may fail independently
  - Network Faults may isolate computers that are still running
  - System Failures may not be immediately known

What did you learn so far?

- Goals / Challenges in Message Passing systems
- Distributed Sorting / Space-Time diagram
- Partial Ordering / Total Ordering
- Concurrent Events / Causal Ordering
- Logical Clocks vs Physical Clocks
- Global Snapshot Detection
- Termination Detection Algorithm
- Leader Election in Rings
- Topology Abstraction and Overlays
- Message Ordering and Group Communication
- Mutual Exclusion Algorithms

- Deadlock Detection Algorithms

[Now] → → →

Recap

> About this Lecture

What do we learn today?

- Deadlock Detection
  - Prevention
  - Avoidance
  - Detection
    - Resource Allocation Graph
    - Banker's algorithm
  - Recovery from Deadlocks
  - Performance Metrics

Let us explore these topics ➔ ➔ ➔

Today's Focus

Deadlocks

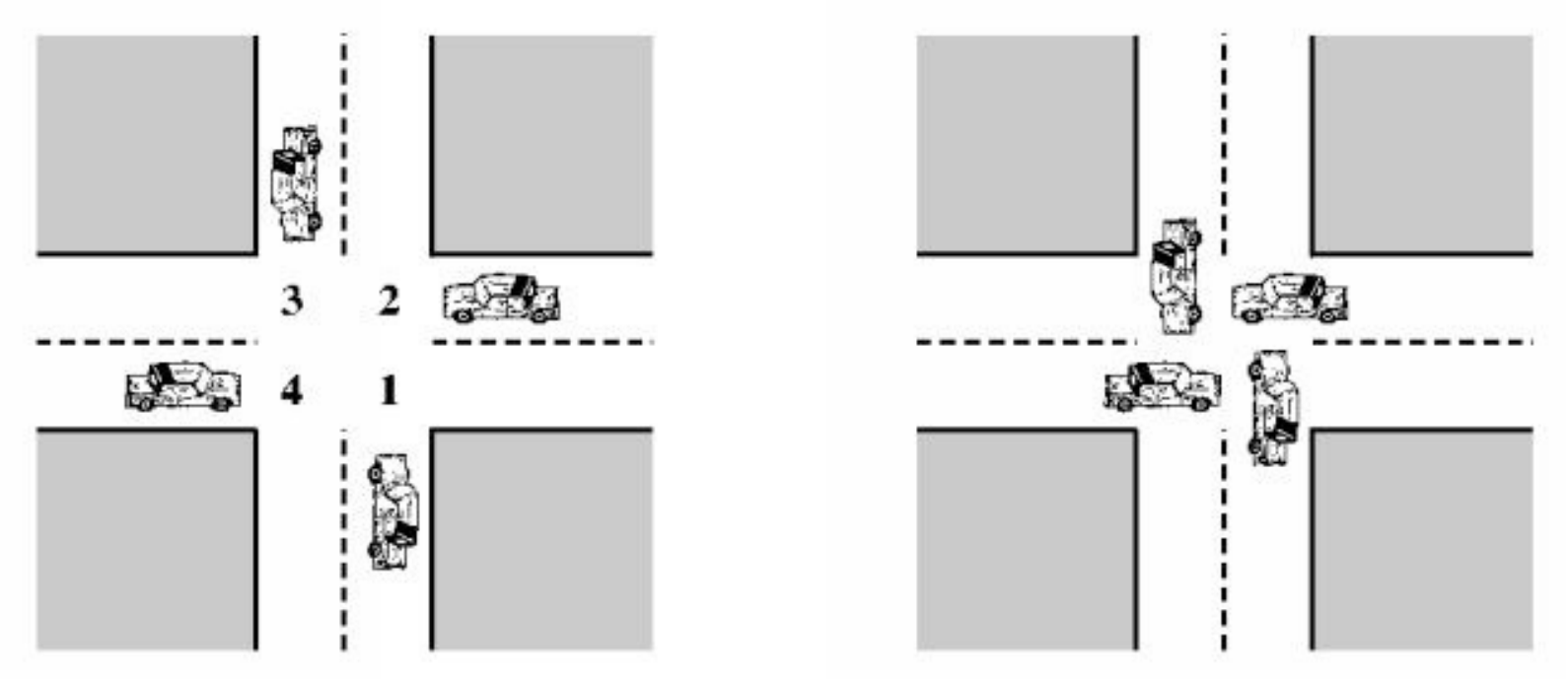
Let us explore deadlock detection, prevention and avoidance algorithms in distributed systems

Distributed Mutual Exclusion (recap)

- No Deadlocks – No processes should be permanently blocked, waiting for messages (Resources) from other sites
- No starvation – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
- Fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation)
- Fault Tolerance – the algorithm is able to survive a failure at one or more sites

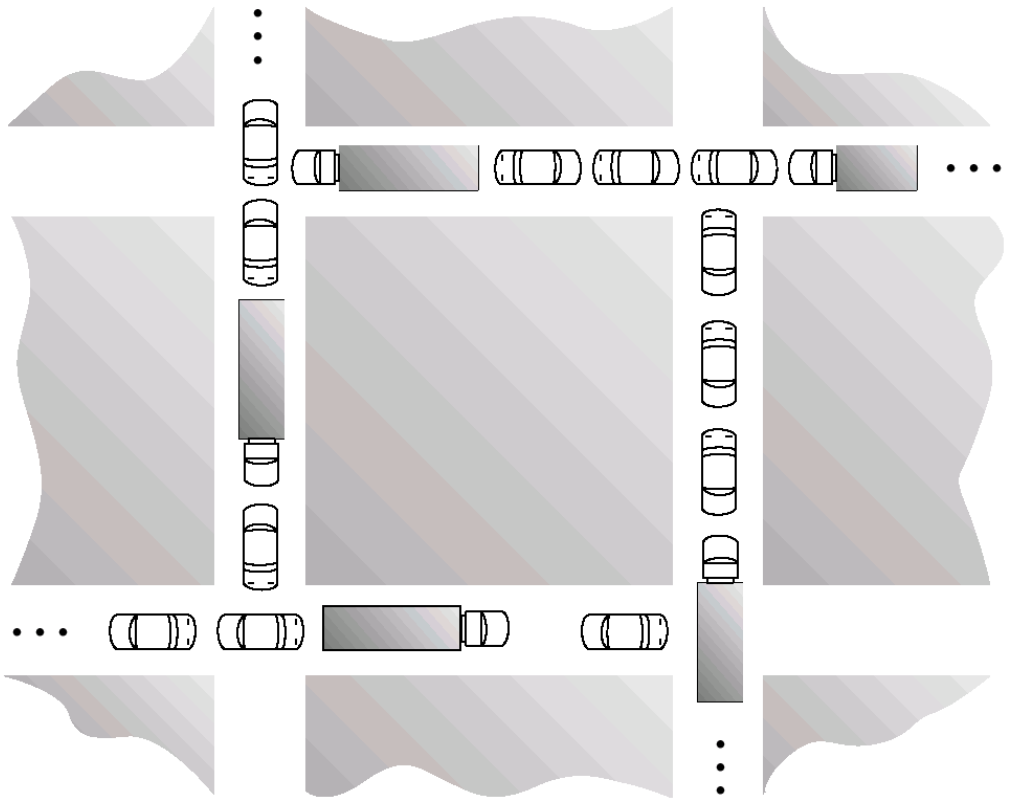
Deadlock – A Simple Example

➔ Vehicular Traffic at a signal



Deadlock – Another Example

➔ Vehicular Traffic – Another Scenario



Deadlock – Illustrated

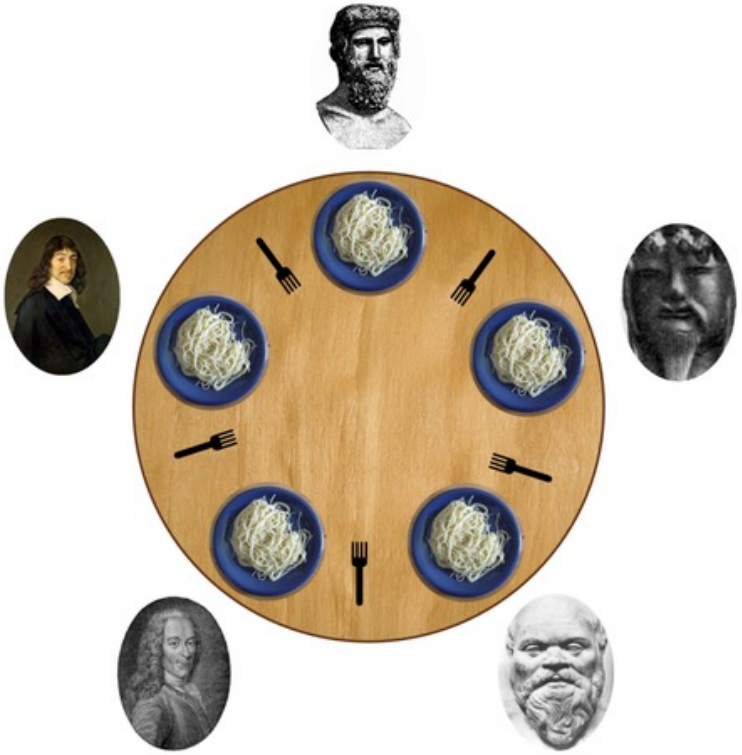
➔ Vehicular Traffic – A real-time scenario

Dining Philosophers' Problem

- Each philosopher must alternately think and eat
- A philosopher can only eat when they have both left and right forks

- **Problem:** How to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve?

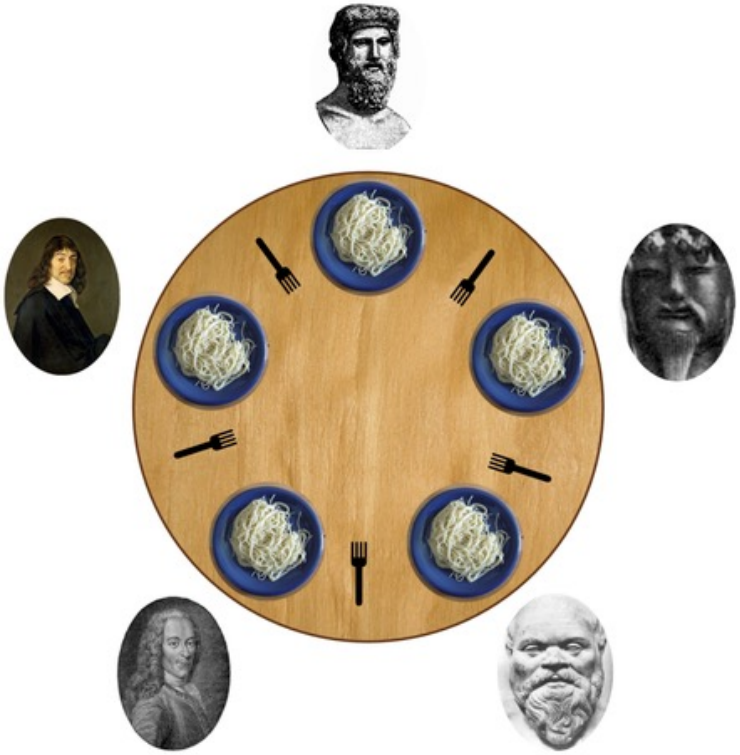
- Suggest a Simple Solution ??



Dining Philosophers' Problem

- Soln - 1: Forks will be numbered 1 through 5 and each philosopher will always pick up the lower-numbered fork first, and then the higher-numbered fork
- Soln - 2: Use Arbitrator (waiter) to grant permission to pick up both forks

➔ Deadlock-Free Solutions !!



Deadlocks in Distributed Sytems

Definition

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- No process can progress in the system
- Competing processes may WAIT indefinitely for resources
- How do we manage resources among the competing tasks efficiently?

Deadlocks – A few more examples

Tape Drives

- Assume that a system has two Tape Drives
- There are two processes P_1 and P_2 each hold one drive
- Now each process needs access to another tape drive
- P_1 does not get access to the resource held by P_2 and vice versa.
- This implies DEADLOCK … neither P_1 nor P_2 succeeds in its attempt

Deadlocks – A few more examples

Semaphores

➔ Semaphores A and B

P_1	P_2
wait (A)	wait(B)
OR	
wait (B)	wait(A)

➔ This implies DEADLOCK … neither P_1 nor P_2 succeeds in its attempt

Deadlock – Characterization

- Mutual exclusion - only one process at a time can use a resource
- Hold and wait - a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption - a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait - there exists a set $\{P_0, P_1, \bullet\bullet\bullet, P_0\}$ of waiting processes such that P_i is waiting for a resource that is held by $P_j \ (mod\ n)$ where n is the total number of resources

System Model

- Resource types: $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - REQUEST
  - USE (Critical Section)
  - RELEASE

- Recall - Distributed Exclusion Algorithms

Resource Allocation Graph (RAG)

- A set of vertices V and a set of edges E

- V is partitioned into two types:
  - Set consisting of all processes

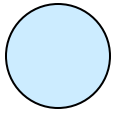
$$P = \{P_1, P_2, \ldots, P_n\}$$

  - Set consisting of all resource types

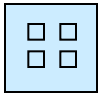
$$R = \{R_1, R_2, \ldots, R_m\}$$

- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph (contd)

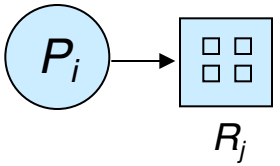


➔ Process

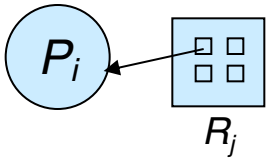


➔ Resource type with 4 instances

➔ P_i requests an instance of R_j

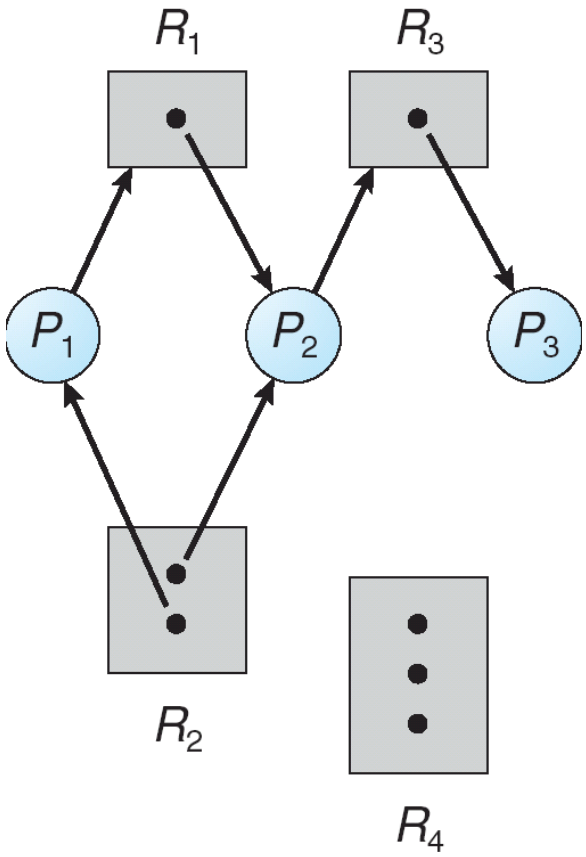


➔ P_i is holding an instance of R_j



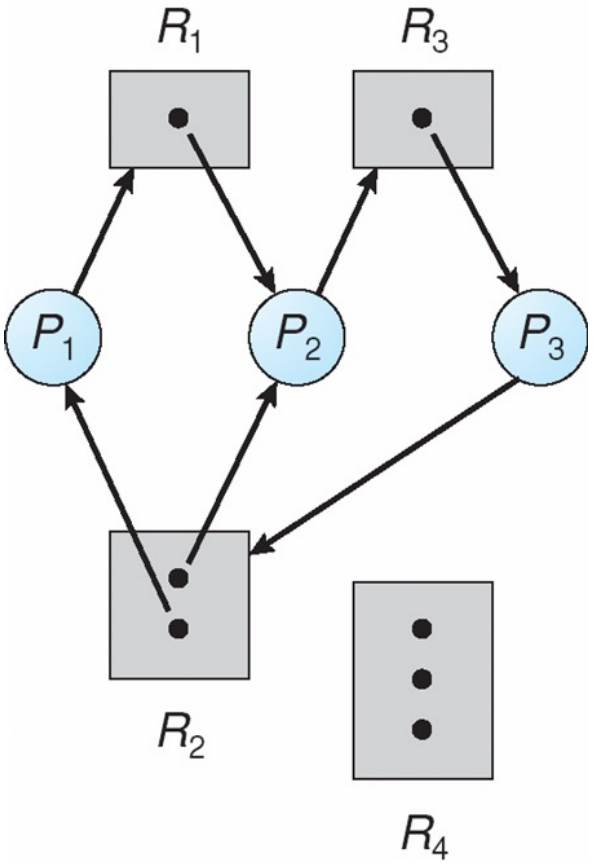
RAG – An example

- Look at this graph
- Resources:
  - R_1 – 1 unit
  - R_2 – 2 units
  - R_3 – 1 unit
  - R_4 – 3 units
- Requests:
  - P_1, P_2, P_3



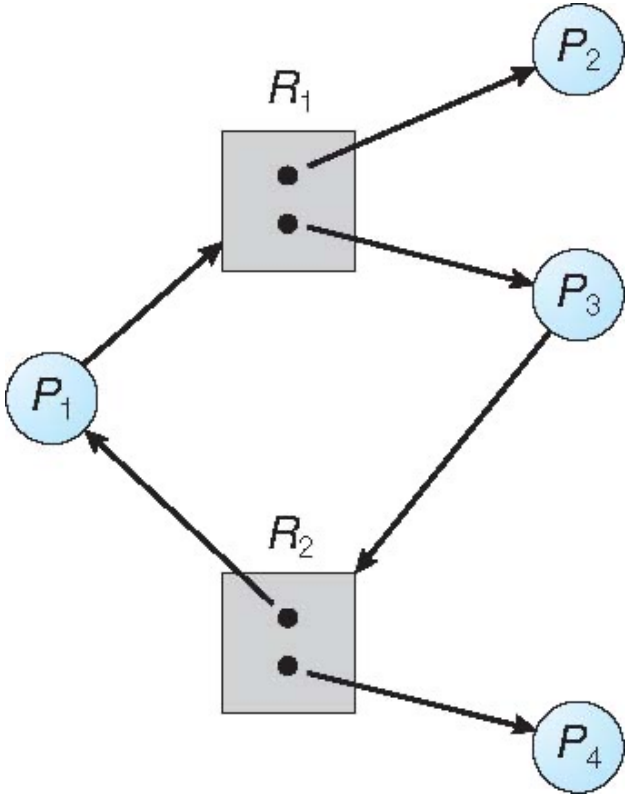
RAG with a Deadlock

- Look at this graph
  - P_1 needs R_1 which in turn used by P_2 and P_2 is requesting R_3 which is currently being accessed by P_3 and P_3 needs R_2 which is being locked by P_1 and P_2
- This implies Deadlock



RAG with a cycle but NO Deadlock

- Look at this graph
  - P_2 and P_4 may release the resource R_3 in finite time as they do not depend on other competing processes
  - There exists a cycle but may not be a deadlock !!



Basic Facts

- ➔ If graph contains no cycles ➔ no deadlock
- ➔ If graph contains a cycle ➔
  - ➔ if only one instance per resource type, then deadlock
  - ➔ if several instances per resource type, possibility of deadlock

How to handle Deadlocks?

- Ensure that the system will never enter a deadlock state
  - Deadlock Prevention – Stop before it happens!
  - Deadlock Avoidance – Precautions !!
  - Deadlock Detection – How to overcome?
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

4 Conditions to occur Deadlocks:

- ➔ Mutual Exclusion – Exclusive access – when a process accesses a resource, it is granted exclusive use of that resource
- ➔ Hold and wait – a process is allowed to hold onto some resources while waiting for other resources
- ➔ No preemption – a process cannot preempt or take away the resources held by another process
- ➔ Cyclical wait – There is a circular chain of waiting processes, each waiting for a resource held by the next process in the chain

Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Safe State

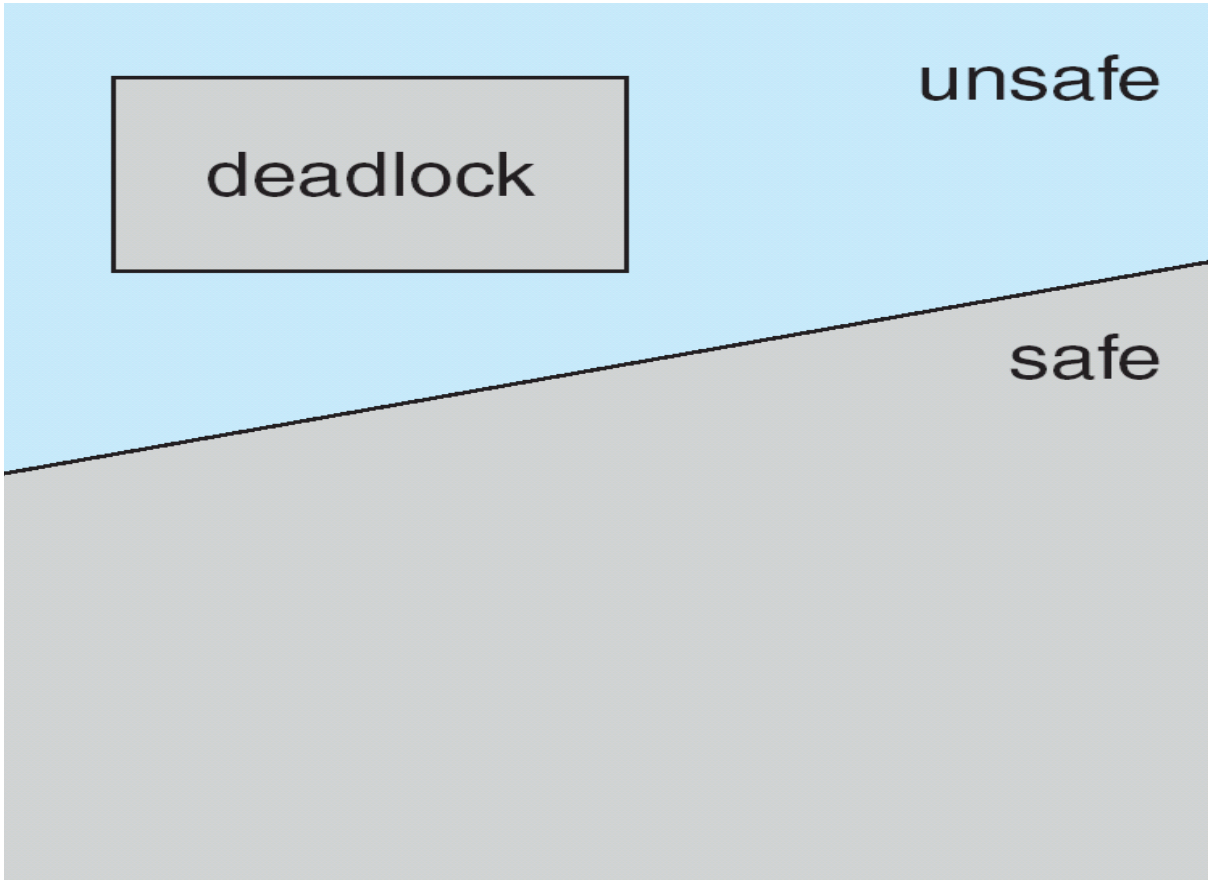
- When a process requests an available resource, system must decide whether the allocation immediate leaves the system in a Safe State?
- System is in Safe State if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL processes such that for each P_i, the resources that P_i can still request, can be satisfied by available resources + resources held by all P_j, $j < I$
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can get resources and so on

Basic Facts

- If a system is in safe state
  - no deadlocks
- If a system is in unsafe state
  - possibility of deadlock
- Avoidance
  - ensure that a system will never enter an unsafe state

Safe / Unsafe / Deadlock State

➔ Illustration of safe, unsafe and deadlock state



Deadlock Avoidance Algorithms

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's algorithm

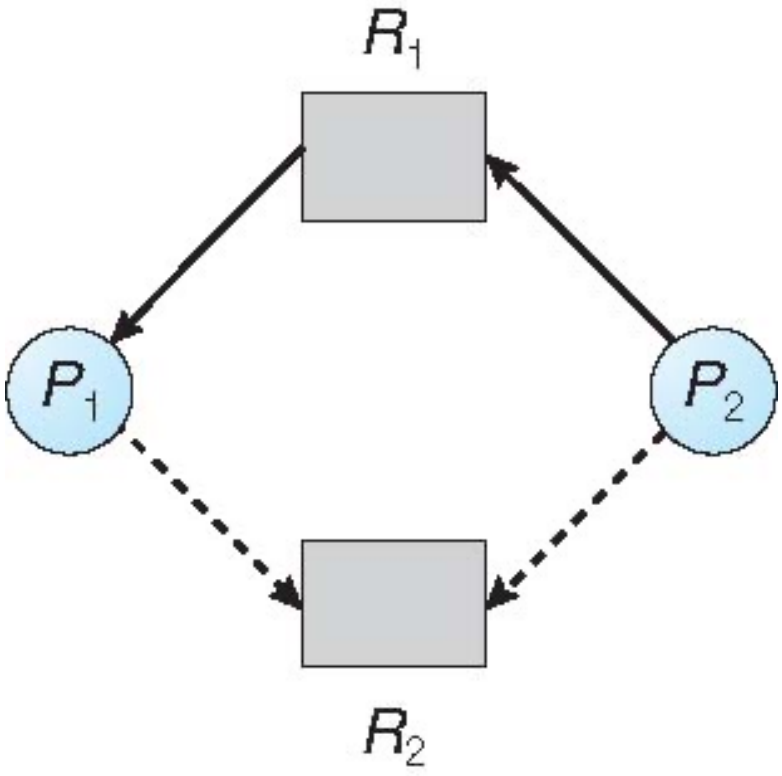
Resource-Allocation Graph Scheme

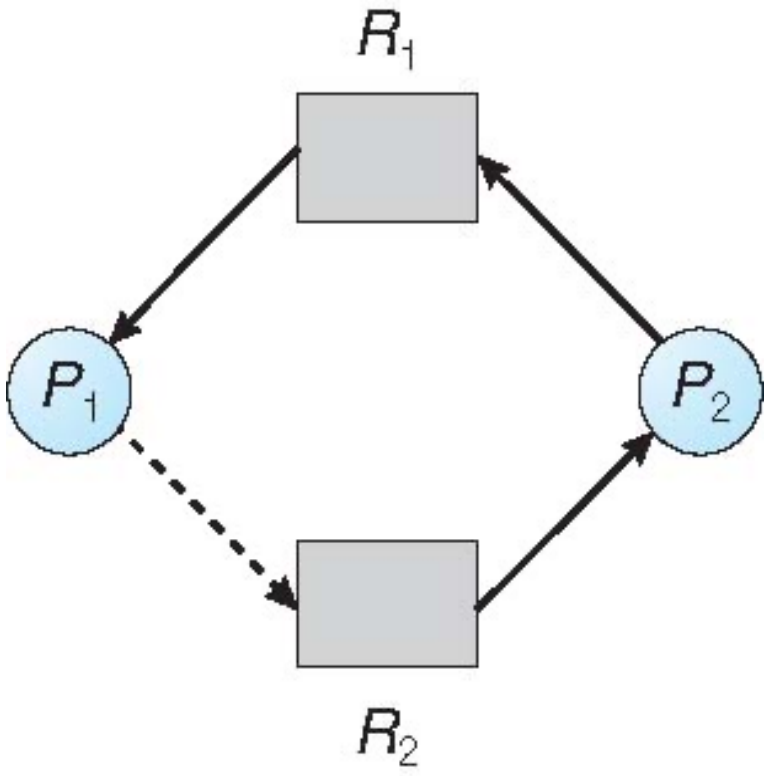
- Claim edge $P_i \rightarrow R_j$ indicate:
  - (i) process P_j may request resource R_j
  - (ii) represented by a dashed line
- Claim edge converts to Request edge when a process requests for a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Resource-Allocation Graph

➔ An Example

Unsafe state





Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances of Resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures - Banker's Algorithm

Let n = number of processes;
m = number of resources types;

- **Available:** Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max:** n x m matrix. If Max[i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation:** n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need:** n x m matrix. If Need[i,j] = k, then P_i may need k more instances of R_j to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

- (1) Let Work and Finish be vectors of length m and n respectively. Initialize: Work = Available and Finish [i] = false for i = 0, 1, …, n- 1
- (2) Find an i such that both:

  (a) Finish [i] = false; (b) $Need_i$? Work.

  If no such i exists, go to step 4
- (3) Work = Work + $Allocation_i$

  Finish[i] = true; go to step 2
- (4) If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algo for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ then go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$ then go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

➔ If safe ➔ the resources are allocated to P_i

➔ If unsafe ➔ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types:
  - A (10 instances), B (5instances), and C (7 instances)
- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (contd)

- Need matrix is defined to be as follows:

  Need = Max – Allocation

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available
  (that is, (1,0,2) (3,3,2) - Is true?

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection Algorithm
- Recovery Scheme

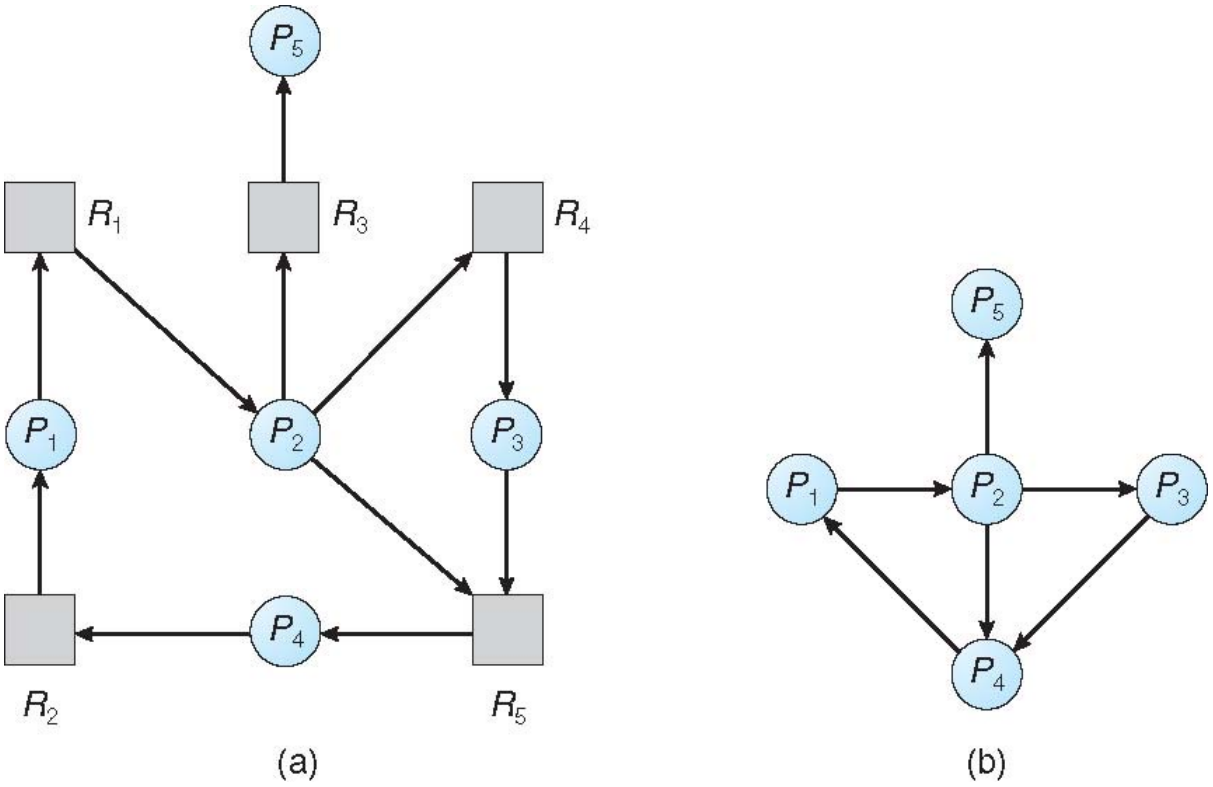
Single Instance of Each Resource Type

- Maintain wait-for graph
- Nodes are processes
- $P_i \rightarrow P_j$ if P_i is waiting for Resources from P_j

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation / Wait-for Graph



Resource allocation Graph

Wait For Graph (WFG)

Several Instances of a Resource Type

- **Available:** A vector of length m indicates the number of available resources of each type

- **Allocation:** An n x m matrix defines the number of resources of each type currently allocated to each process

- **Request:** An n x m matrix indicates the current request of each process.

  If Request [i, j] = k, then process P_i is requesting k more instances of resource type R_j.

Detection Algorithm

- (1) Let Work and Finish be vectors of length m and n, respectively Initialize:
  - (a) Work = Available
  - (b) For i = 1,2, …, n,
    Finish[i] = false
- (2) Find an index i such that both:
  - (a) Finish[i] == false
  - (b) $Request_i \leq Work$
    If no such i exists, go to step 4

Detection Algorithm (contd)

- (3) Work = Work + $Allocation_i$
  Finish[i] = true; go to step 2

- (4) If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish[i] == false, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Detection Algorithm - Example

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in Finish[i] = true for all i

Detection Algo – Example (contd)

- P_2 requests an additional instance of type C

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
- How often a deadlock is likely to occur?
- How many processes will need to be rolled back?
  - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Resource Links

- Distributed Deadlock Detection
  - http://www.cse.scu.edu/~jholliday/dd_9_16.htm

- Coffman et. al., System Deadlocks, ACM Computing Surveys. 3 (2) (1971): 67–78. DOI:10.1145/356586.356588

- Havender, James W., Avoiding deadlock in multitasking systems, IBM Systems Journal. 7 (2) (1968): 74. DOI:10.1147/sj.72.0074

- Knapp, Edgar, Deadlock detection in distributed databases, ACM Computing Surveys, 19 (4) (1987): 303–328. DOI:10.1145/45075.46163. ISSN 0360-0300

Summary

- Recap: Distributed Mutual Exclusion Algorithms

- Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
    - Resource Allocation Graphs
    - Banker's Algorithm
  - Recovery from Deadlocks
- Performance Metrics

Many more to come up … ! Stay tuned in !!

Penalties

- Every Student is expected to strictly follow a fair Academic Code of Conduct to avoid penalties

- Penalties is heavy for those who involve in:
  - Copy and Pasting the code
  - Plagiarism (copied from your neighbor or friend – in this case, both will get "0" marks for that specific take home assignments)
  - If the candidate is unable to explain his own solution, it would be considered as a "copied case"!!
  - Any other unfair means of completing the assignments

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

How to reach me?

➔ Please leave me an email:

rajendra [DOT] prasath [AT] iiits [DOT] in

➔ Visit my homepage @

➔ https://www.iiits.ac.in/people/regular-faculty/dr-rajendra-prasath/

(OR)

➔ http://rajendra.2power3.com

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- You may ask for one-to-one meeting

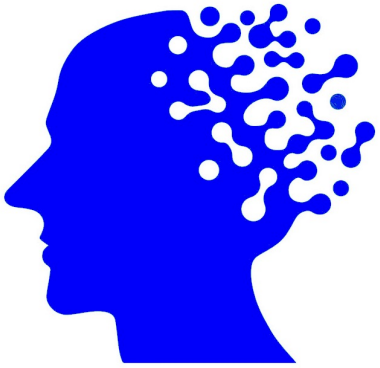
Best Approach

- You may leave me an email any time
  (email is the best way to reach me faster)

Questions
It's Your Time

Contact Information

Dr. Rajendra Prasath
IIIT Sri City, Chittoor



Spring 2024

Distributed Computing

- Self Stabilization Algorithms

Dr. Rajendra Prasath

Indian Institute of Information Technology Sri City, Chittoor

10th and 12th April 2024 (http://rajendra.2power3.com)

> Distributed Computing?

- How will you design a Distributed Algorithm?

- Learn to Solve using Distributed Algorithms

Recap: Distributed Systems

A Distributed System:

- A collection of independent systems that appears to its users as a single coherent system
- A system in which hardware and software components of networked computers communicate and coordinate their activity only by passing messages
- A computing platform built with many computers that:
  - Operate concurrently
  - Are physically distributed (have their own failure modes)
  - Are linked by a network
  - Have independent clocks

Recap: Characteristics

- Concurrent execution of processes:
  - Non-determinism, Race Conditions, Synchronization, Deadlocks, and so on
- No global clock
  - Coordination is done by message exchange
  - No Single Global notion of the correct time
- No global state
  - No Process has a knowledge of the current global state of the system
- Units may fail independently
  - Network Faults may isolate computers that are still running
  - System Failures may not be immediately known

What did you learn so far?

- ➔ Goals / Challenges in Message Passing systems
- ➔ Space-Time Diagram / Partial Ordering / Total Ordering
- ➔ Concurrent Events / Causal Ordering
- ➔ Logical Clocks vs Physical Clocks
- ➔ Global Snapshot Detection
- ➔ Termination Detection Algorithm
- ➔ Leader Election in Rings
- ➔ Topology Abstraction and Overlays
- ➔ Message Ordering and Group Communication
- ➔ Mutual Exclusion Algorithms
- ➔ Deadlock Detection Algorithms
- ➔ Checkpointing and Rollback Recover Algorithms
- ➔ Distributed Consensus Algorithms

- ➔ Self-Stabilizing Algorithms

[Now] ➔ ➔ ➔

Recap

> About this Lecture

What do we learn today?

- **Self-Stabilization Algorithms**
  - Problem Specification
  - Token Rings
  - Dijkstra's Algorithm
  - Minimum Spanning Tree
  - Breadth First Tree

Let us explore these topics ➔ ➔ ➔

Today's Focus

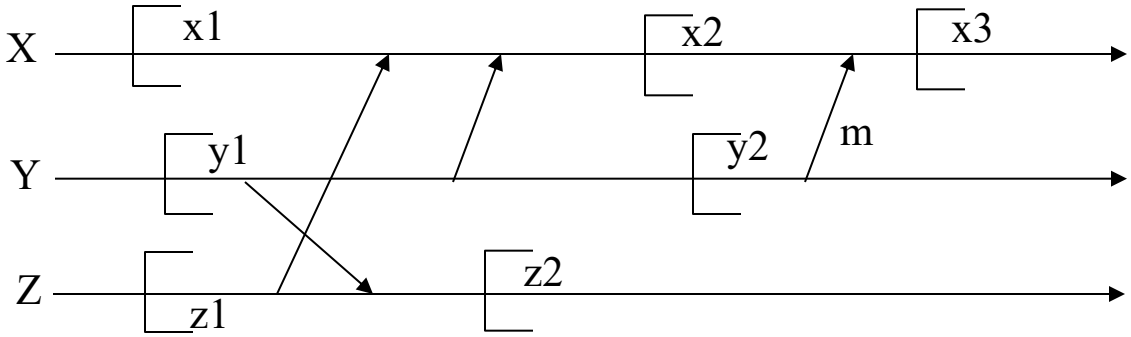
Self-Stabilization in Distributed Systems

Let us explore Self-Stabilization algorithms in Distributed Systems

Handling Failures / Recovery?

- Failure of a site/node in a distributed system causes inconsistencies in the state of the system.
- Recovery: bringing back the failed node in step with other nodes in the system.
- Failures:
  - Process failure:
    - Deadlocks, protection violation, erroneous user input, etc.
  - System failure:
    - Failure of processor/system. System failure can have full/partial amnesia.
    - It can be a pause failure (system restarts at the same state it was in before the crash) or a complete halt.
  - Secondary storage failure: data inaccessible.
  - Communication failure: network inaccessible.

Consistent Checkpoints



- Overcoming domino effect and livelocks: checkpoints should not have messages in transit.
- Consistent checkpoints: no message exchange between any pair of processes in the set as well as outside the set during the interval spanned by checkpoints.
- {x1,y1,z1} is a strongly consistent checkpoint

Types of CP-RR Algorithms

- Synchronous Algorithm
  - Two Phase algorithm proposed by Koo and Toueg
- Asynchronous Algorithm
  - A simple algorithm proposed by Juang & Venkatesan

Overview

- Self-Stabilizing (SS) Systems
  - Legitimate / Illegitimate states
  - System Model
  - Token Ring System
    - Dijkstra's Self-stabilizing Algorithm
    - Construct Breadth-First Trees (BFT)
  - Computational Cost
  - Fault Tolerance / Factors Preventing SS
  - Limitations of SS systems

Introduction

- Legitimate State – Systems behave correctly as it has expected to.
- Illegitimate State – inactive state or state in which the system misbehaves (Message is lost)
- Self – Stabilization – A concept of fault-tolerance in distributed computing

- Regardless of initial state, system is guaranteed to converge to a legitimate state in a finite amount of time without any outside intervention

- Problem – Nodes do not have a global memory

Definition

A system is self-stabilizing if and only if:

- Convergence: Starting from any state, it is guaranteed that the system will eventually reach a correct state
- Closure: Given that the system is in a correct state, it is guaranteed to stay in a correct state, provided that no fault happens

- A system is said to be randomized self-stabilizing if and only if it is self-stabilizing and the expected number of rounds needed to reach a correct state is bounded by some constant k

System Model

- An abstract computer model: state machine.
- A distributed system model comprises of a set of n state machines called processors that communicate with each other, which can be represented as a GRAPH
- Message passing communication model:
  - queue(s) Q_{ij}, for messages from P_i to P_j
- System configuration is set of states, and message queues.
- In any case it is assumed that the topology remains connected, i.e., there exists a path between any two nodes.

Token Rings

- Dijkstra's Self-Stabilizing Token Ring System
  - When a machine has a privilege, it is able to change its current state, which is referred to as a move.
  - A legitimate state must satisfy the following constraints:
  - There must be at least one privilege in the system (liveness or no deadlock).
  - Every move from a legal state must again put the system into a legal state (closure).
  - During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation)
  - Given any two legal states, there is a series of moves that change one legal state to the other (reachability). Dijkstra considered a legitimate (or legal) state as one in which exactly one machine enjoys the privilege

Dijkstra's Algorithm

- For any machine:
  - S – State of its own
  - L – State of the left neighbor and
  - R - State of the right neighbor on the ring

- The exceptional machine:
  - If L = S then S = (S+1) mod K;

- All other machines:
  - If L = S then S = L;

Dijkstra's Algorithm

- A Privilege of a machine is able to change its current state on a Boolean predicate that consists of its current state and the states of its neighbors
- When a machine has a privilege, it is able to change its current state, which is referred to as a move.

Second solution (K = 3)

- The bottom machine, machine 0:
  - If (S+1) mod 3 = R then S = (S−1) mod 3;
- The top machine, machine n−1:
  - If L = R and (L+1) mod 3 = S then S = (L+1) mod 3;
- The other machines:
  - If (S+1) mod 3 = L then S = L;

An Illustration

➔ 4 Machines: M0, M1, M2, and M3

State of machine 0	State of machine 1	State of machine 2	State of machine 3	Privileged machines	Machine to make move
0	1	0	2	0, 2, 3	0
2	1	0	2	1, 2	1
2	2	0	2	1	1
2	0	0	2	0	0
1	0	0	2	1	1
1	1	0	2	2	2
1	1	1	2	2	2
1	1	2	2	1	1
1	2	2	2	0	0
0	2	2	2	1	1
0	0	2	2	2	2
0	0	0	2	3	3
0	0	0	1	2	2

Fault Tolerance

A Self-Stabilizing System handles Transient faults:

- Inconsistent Initialization: Different processes initialized to local states that are inconsistent with one another.
- Mode of Change: There can be different modes of execution of a system. In changing the mode of operation, it is impossible for all processes to effect the change in same time.
- Transmission Errors: Loss, corruption, or reordering of messages
- Memory Crash

Factors Preventing Self-Stabilization

Transient faults:

- ➔ Symmetry: Processes should not be identical/symmetric because solution generally relies on a distinguished process.
- ➔ Termination: If any unsafe global state is a final state, system will not be able to stabilize
- ➔ Isolation: Inadequate communication among processes can lead to local states consistent, however, the resulting global state is not safe!
- ➔ Look-alike configurations: Such configurations result when the same computation is enabled in two different states with no way to differentiate between them. Then system cannot guarantee convergence from unsafe state

Limitations of Self-Stabilizing

- Need for an exceptional machine
- Convergence-response tradeoffs
  - Convergence span denotes the maximum number of critical transitions made before the system reaches a legal state
  - Response span denotes the maximum number of transitions to get from the starting state to some goal state
  - Critical Transitions. (ex. A process moves into a critical section, while another is already in!)

Limitations of Self-Stabilizing (contd)

- Pseudo-stabilization: Weaker, but less expensive with respect to self-stabilization.
  - Every computation only needs to have some state such that the suffix of the computation beginning at this state is in the set of legal computations.
- Verification of self-stabilizing system
  - Verification may be difficult.
  - Stair method developed; Proving the algorithm stabilizes in each step verifies correctness of the entire algorithm, where interleaving assumptions are relaxed

Costs of Self-Stabilization

- Assessment of cost factor
  - **Convergence Span:** The maximum number of transitions that can be executed in a system, starting from an arbitrary state, before it reaches a safe state.
  - **Response Span:** The maximum number of transitions that can be executed in a system to reach a specified target state, starting from some initial state. The choice of initial state and target state depends upon the application

Interesting Algorithms

- Breadth-First Trees (Huang and Chen, 1992)
- All-pairs shortest path problem (Chandrasekar and Srimani, 1994)
- Finding centers and medians of trees (Bruell et al. 1999)
- Shortest path problem (Huang and Lin, 2002)
- Shortest path problem assuming read/write atomicity (Huang, 2005)
- Connected minimal dominating sets (Turau and Hauck, 2009)
- Finding efficient sets of graphs and trees (Turau, 2013)
- Leader election (Altisen et al., 2017)
- Edge monitoring in wireless sensor networks (Neggazi et al., 2017)

Breadth-First Trees

- Proposed by Huang and Chen, 1992
- Breadth-First Tree (BFT): A Breadth-First Tree of a connected graph is a spanning tree of the graph in which each node has a minimum distance to the root along the tree edges
- How to construct a BFT from a given graph?
- How to develop a self-stabilizing algorithm for constructing the Breadth-First Tree?

Self-Stabilizing Algorithm for BFT

- Basics:
  - Model a distributed system as a connected graph G(V, E)
  - A specific node r is selected as the root.
  - How to build a breadth-first- tree rooted at r from G with each node knowing its level in the tree.
  - For each node i, let N_i be the set of i's neighbors
  - Each node i other than the root maintains the following two local variables:
    - L(i): the level of i,
    - P(i) : the parent of i,

    where 2 <= L(i) <= n and P(i) in N_i

Self-Stabilizing Algorithm for BFT

- From G, construct BFT rooted at node r
- In a tree:
  - $L(i) = (L(p_i) + 1)$ for any i other than r
  - $L(pi) = min(\{L(j) \mid j \text{ in } N_j\})$ based on the property of breadth-first trees.
- The system reaches a legitimate state, when the following predicate is true

$$BFT = (\forall i: i \neq r: L(i) = L(p_i) + 1 \wedge L(p_i) = min(\{L(j) \mid j \text{ in } N_i\}))$$

Self-Stabilizing Algorithm for BFT

- For any node i, if $L(i) <= L(p_i)$, we call node i an L-$turn$ node, or more specifically a k-$turn$ node, where $k = L(i)$. Also let t_k be the number of all the k-$turn$ nodes in the system
- Define F_1 as follows:

$$F_1 \equiv (t_2, t_3, \ldots, t_n)$$

- Compare the values of F_1 is by lexicographical order
- Based on lexicographical order,

$(a_1, a_2, \ldots) > (b_1, b_2, \ldots)$ if there exists some k such that $a_i = b_i$, $1 <= i < k$, and $a_k > b_k$

Self-Stabilizing Algorithm for BFT

- Define F_2 as follows:

$$F_2 \equiv \sum_{i,\ i \# r} (L(i) + L(p_i))$$

- That is, for each node i other than the root, it contributes two values to F_2: one is the level of itself and the other is the level of its parent

Summary

- Recap: Checkpointing
  - Consistent set of checkpoints
  - Synchronous Algo (Koo and Toueg)
  - Asynchronous Algo (Juang & Venkatesan)
- Distributed Consensus
  - State of Machines
  - Legitimate / Illegitimate States
  - Self-Stabilizing Algorithms
  - Dijkstra's algorithm (token rings)
  - Constructing a Breadth First Tree
  - Fault Tolerance
  - Costs of self-stabilization

Many more to come up … ! Stay tuned in !!

Penalties

- Every Student is expected to strictly follow a fair Academic Code of Conduct to avoid penalties

- Penalties is heavy for those who involve in:
  - Copy and Pasting the code
  - Plagiarism (copied from your neighbor or friend – in this case, both will get "0" marks for that specific take home assignments)
  - If the candidate is unable to explain his own solution, it would be considered as a "copied case"!!
  - Any other unfair means of completing the assignments

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

How to reach me?

➔ Please leave me an email:

rajendra [DOT] prasath [AT] iiits [DOT] in

➔ Visit my homepage @

➔ https://www.iiits.ac.in/people/regular-faculty/dr-rajendra-prasath/

(OR)

➔ http://rajendra.2power3.com

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- You may ask for one-to-one meeting

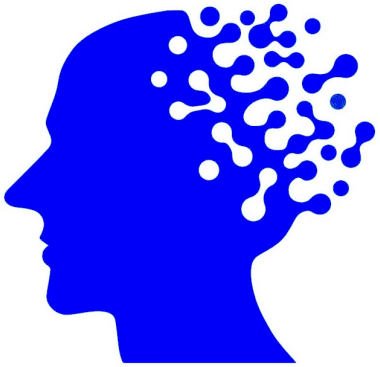
Best Approach

- You may leave me an email any time
  (email is the best way to reach me faster)

Questions
It's Your Time

Contact Information

Dr. Rajendra Prasath
IIIT Sri City, Chittoor



Spring 2024

Distributed Computing

- Distributed Consensus

Dr. Rajendra Prasath

Indian Institute of Information Technology Sri City, Chittoor

19th April 2024 (http://rajendra.2power3.com)

> Distributed Computing?

- How will you design a Distributed Algorithm?

- Learn to Solve using Distributed Algorithms

Recap: Distributed Systems

A Distributed System:

- A collection of independent systems that appears to its users as a single coherent system
- A system in which hardware and software components of networked computers communicate and coordinate their activity only by passing messages
- A computing platform built with many computers that:
  - Operate concurrently
  - Are physically distributed (have their own failure modes)
  - Are linked by a network
  - Have independent clocks

Recap: Characteristics

- Concurrent execution of processes:
  - Non-determinism, Race Conditions, Synchronization, Deadlocks, and so on
- No global clock
  - Coordination is done by message exchange
  - No Single Global notion of the correct time
- No global state
  - No Process has a knowledge of the current global state of the system
- Units may fail independently
  - Network Faults may isolate computers that are still running
  - System Failures may not be immediately known

What did you learn so far?

- ➔ Goals / Challenges in Message Passing systems
- ➔ Space-Time Diagram / Partial Ordering / Total Ordering
- ➔ Concurrent Events / Causal Ordering
- ➔ Logical Clocks vs Physical Clocks
- ➔ Global Snapshot Detection
- ➔ Termination Detection Algorithm
- ➔ Leader Election in Rings
- ➔ Topology Abstraction and Overlays
- ➔ Message Ordering and Group Communication
- ➔ Mutual Exclusion Algorithms
- ➔ Deadlock Detection Algorithms
- ➔ Checkpointing and Rollback Recover Algorithms

- ➔ Distributed Consensus Algorithms

[Now] ➔ ➔ ➔

Recap

> About this Lecture

What do we learn today?

- Consensus in Distributed Systems
  - Problem Specification
  - Asynchronous Consensus
  - Byzantine Generals
  - The OM(m) algorithm
  - The Final Theorem

Let us explore these topics ➔ ➔ ➔

Today's Focus

Consensus in Distributed Systems

Let us explore distributed consensus algorithms in Distributed Systems

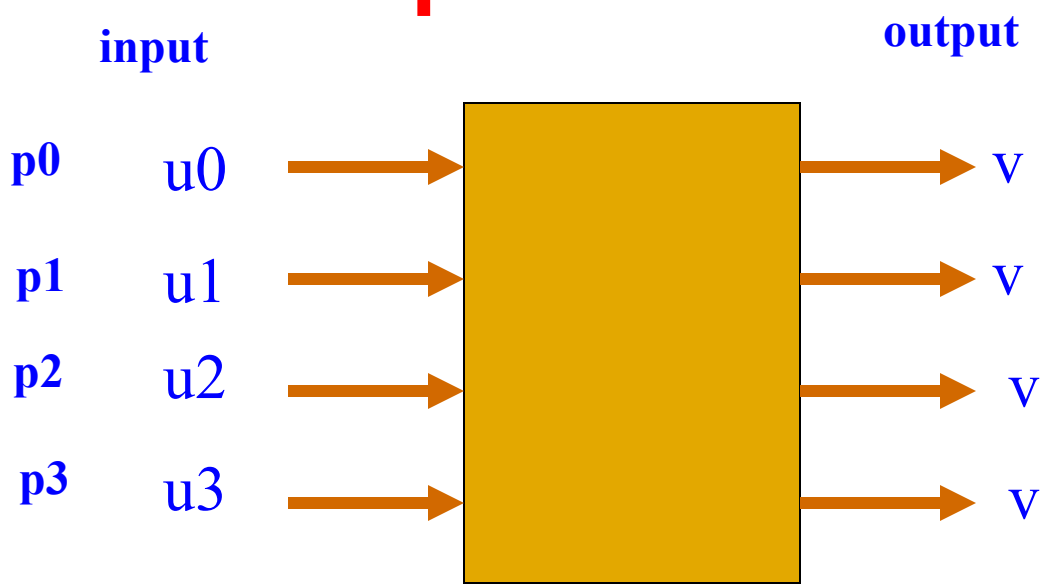
Distributed Consensus

Reaching agreement is a fundamental problem in distributed computing

Examples:

- ➔ Leader election / Mutual Exclusion
- ➔ Commit or Abort in distributed transactions
- ➔ Reaching agreement about which process has failed
- ➔ Clock phase synchronization
- ➔ Air traffic control system: all aircrafts must have the same view

Problem Specification



- Each process p_i has an input value u_i
- These processes exchanges their inputs, so that the outputs of all non-faulty processes become identical, even if one or more processes fail at any time
- Output v must be equal to the value of at least one process

Problem Specification

Termination	Every non-faulty process must eventually decide.
Agreement	The final decision of every non-faulty process must be identical.
Validity	If every non-faulty process begins with the same initial value v, then their final decision must be v

Observation

- If there is no failure, then reaching consensus is trivial. All-to-all broadcast followed by a applying a choice function …

- Consensus in presence of failures can however be complex. The complexity depends on the system model and the type of failures

Asynchronous Consensus

Seven members of a busy household decided to hire a cook, since they do not have time to prepare their own food

Each member *separately interviewed* every applicant for the cook's position. Depending on how it went, each member voted "yes" (means "hire") or "no" (means "don't hire").

These members will now have to communicate with one another to reach a uniform final decision about whether the applicant will be hired

The process will be repeated with the next applicant, until someone is hired. Consider various modes of communication like shared memory or message passing Also assume that one process may crash at any time

Asynchronous Consensus

Theorem:

In a purely asynchronous distributed system, the consensus problem is impossible to solve if even a single process crashes

Fischer, Lynch, Patterson (commonly known as FLP 85). Received the most influential paper award of ACM PODC in 2001

Proof

Bivalent and Univalent states

A decision state is bivalent, if starting from that state, there exist two distinct executions leading to two distinct decision values 0 or 1

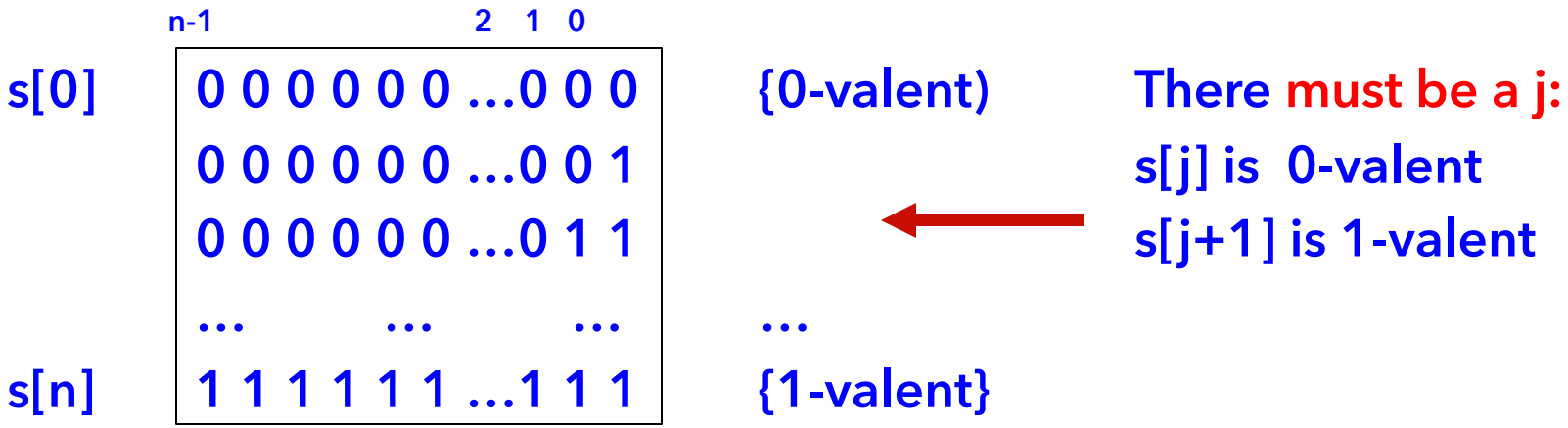
Otherwise it is univalent

An univalent state may be either 0-valent or 1-valent.

Proof

Lemma: Every consensus protocol must have a bivalent initial state.

Proof by contradiction: *Suppose not. Then consider the following input patterns:*



What if process j crashes at the first step?

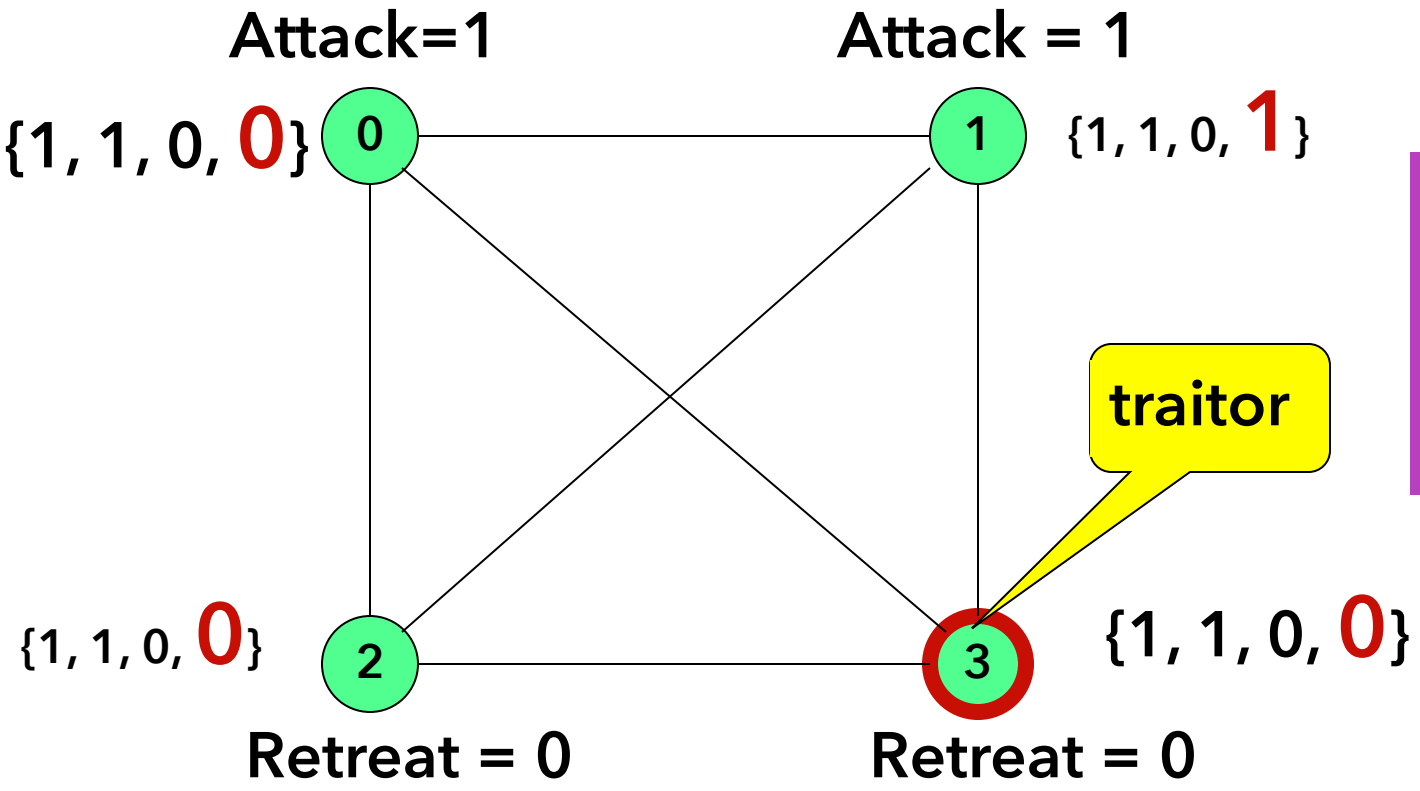
Consensus in Synchronous Systems: Byzantine Generals Problem

- Describes and solves the consensus problem on the synchronous Communication model
- Processor speeds have lower bounds and communication delays have upper bounds.
- The network is completely connected
- Processes undergo byzantine failures, the worst possible kind of failure

Byzantine Generals Problem

- n generals $\{0, 1, 2, \ldots, n-1\}$ decide about whether to "attack" or to "retreat" during a particular phase of a war. The goal is to agree upon the same plan of action
- Some generals may be "traitors" and therefore send either no input, or send conflicting inputs to prevent the "loyal" generals from reaching an agreement
- Devise a strategy, by which every loyal general eventually agrees upon the same plan, regardless of the action of the traitors

Byzantine Generals



The traitor may send out conflicting inputs

Every general will broadcast his judgment to everyone else. These are inputs to the consensus protocol.

Byzantine Generals

We need to devise a protocol so that every peer (call it a lieutenant) *receives the same value* from any given general (call it a commander)

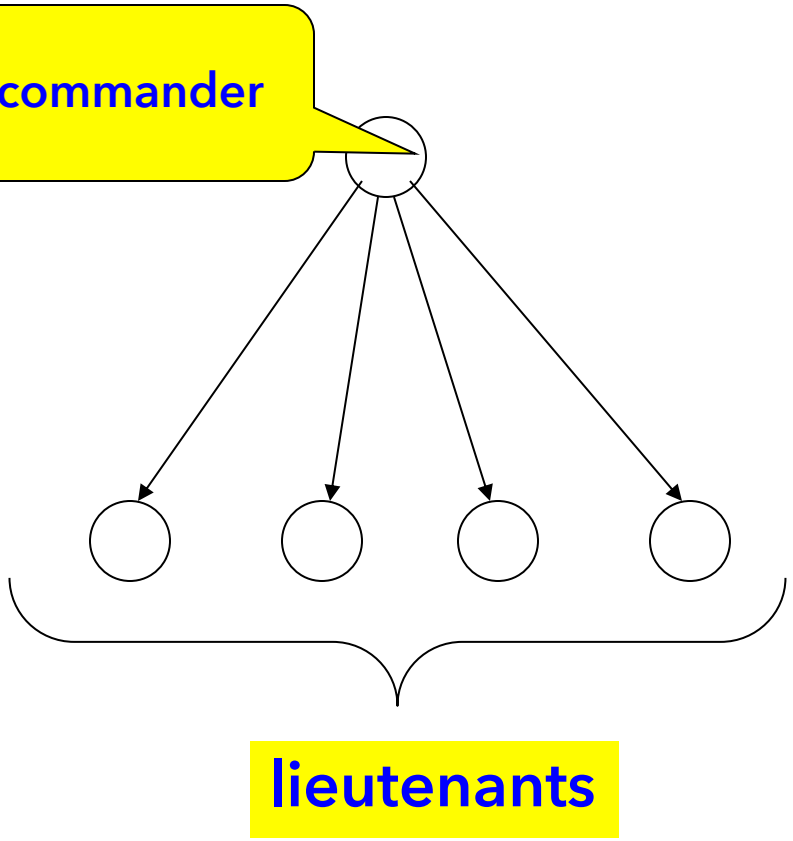
Clearly, the lieutenants will have to use secondary information

Note that the roles of the commander and the lieutenants will rotate among the generals

Interactive consistency specifications

IC1: Every loyal lieutenant receives the same order from the commander

IC2: If the commander is loyal, then every loyal lieutenant receives the order that the commander sends



The Communication Model

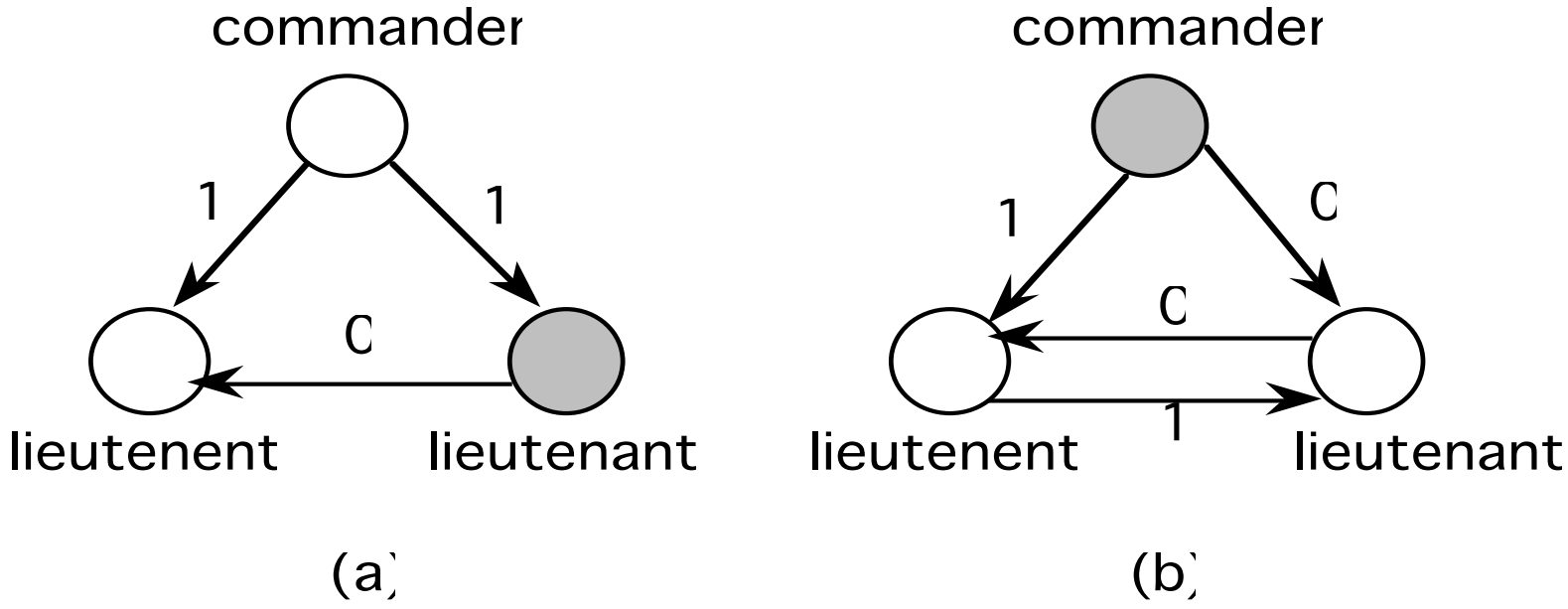
Oral Messages (OM)

1. Messages are not corrupted during the transit
2. Messages can be lost, but the absence of message can be detected
3. When a message is received (or its absence is detected), the receiver knows the identity of the sender (or the defaulter)

OM(m) represents an *interactive consistency protocol* in presence of at most m traitors.

An Impossibility Result

Using oral messages, no solution to the Byzantine Generals problem exists with three or fewer generals and one traitor. Consider the two cases:



In (a), to satisfy IC2, lieutenant 1 must trust the commander, but in IC2, the same idea leads to the violation of IC1.

Impossibility result

Using oral messages, no solution to the Byzantine Generals problem exists with 3m or fewer generals and m traitors (m > 0)

The proof is by contradiction: Assume that such a solution exists. Now, divide the 3m generals into three groups of m generals each, such that all the traitors belong to one group. Let one general simulate each of these three groups. This scenario is equivalent to the case of three generals and one traitor. We already know that such a solution does not exist.

Note: Be always suspicious about such an informal reasoning (refer to Lamport's original paper)

The OM(m) algorithm

Recursive algorithm

OM(m)

↓

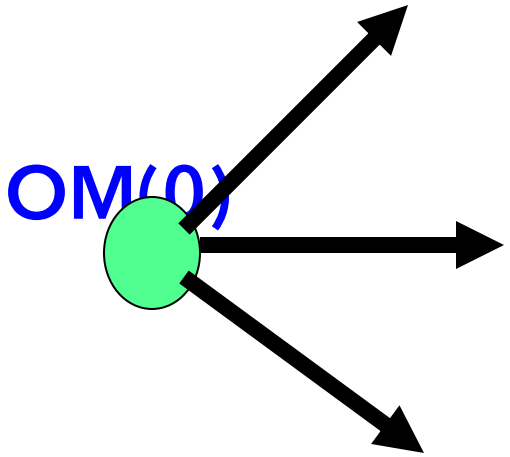
OM(m-1)

↓

OM(m-2)

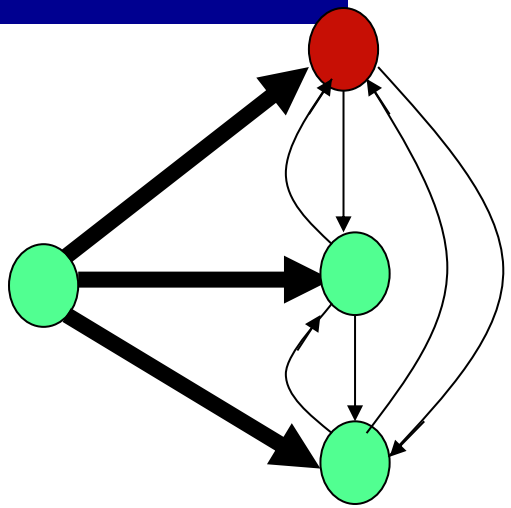
↓

OM(0)



OM(0) = Direct broadcast

The OM(m) algorithm



1. Commander i sends out a value v (0 or 1)

1. If $m > 0$, then every lieutenant $j \neq i$, after receiving v , acts as a commander and initiates OM(m-1) with everyone except i

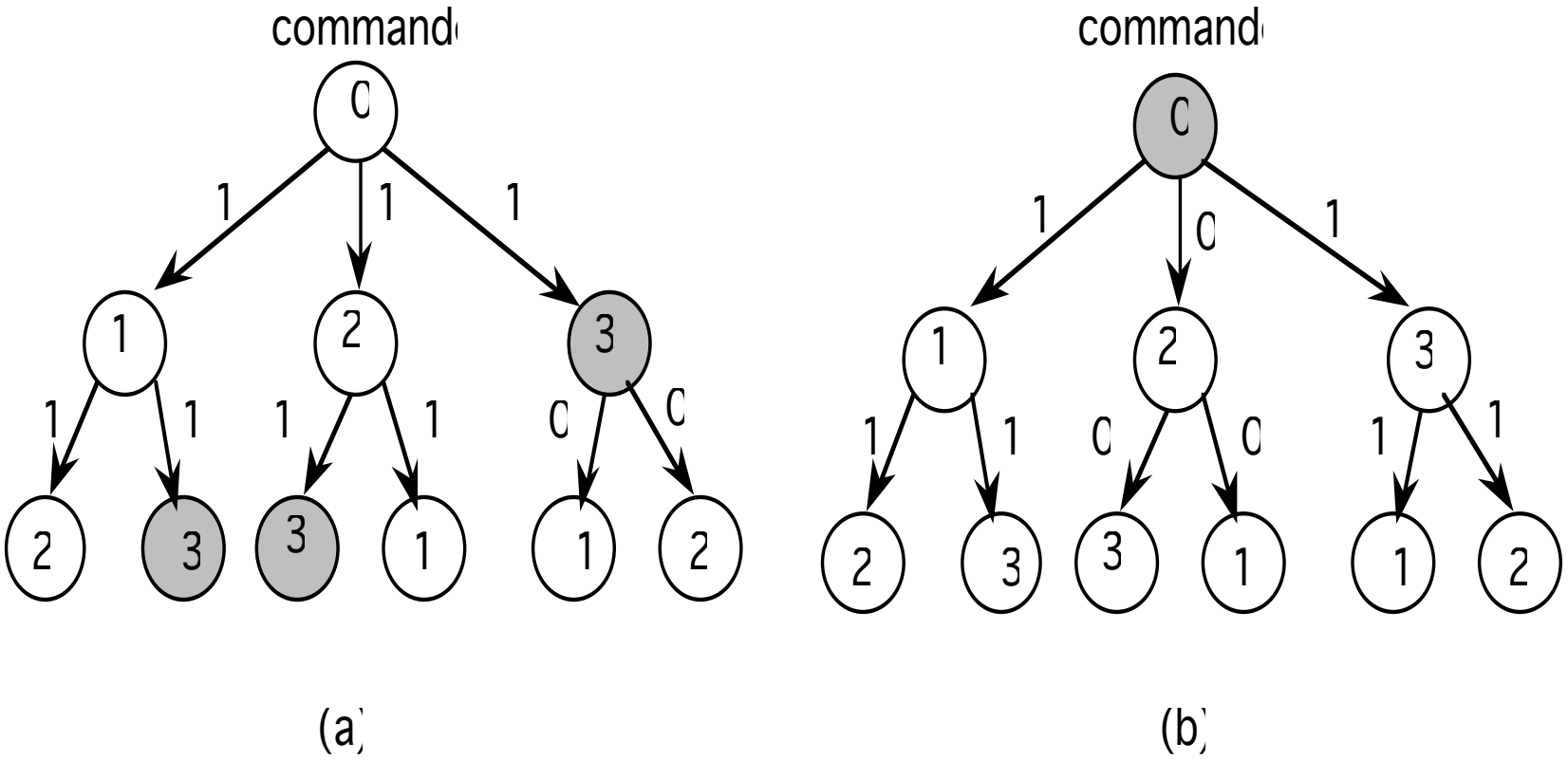
2. Every lieutenant, collects (n-1) values:

   → (n-2) values received from the lieutenants using OM(m-1) and

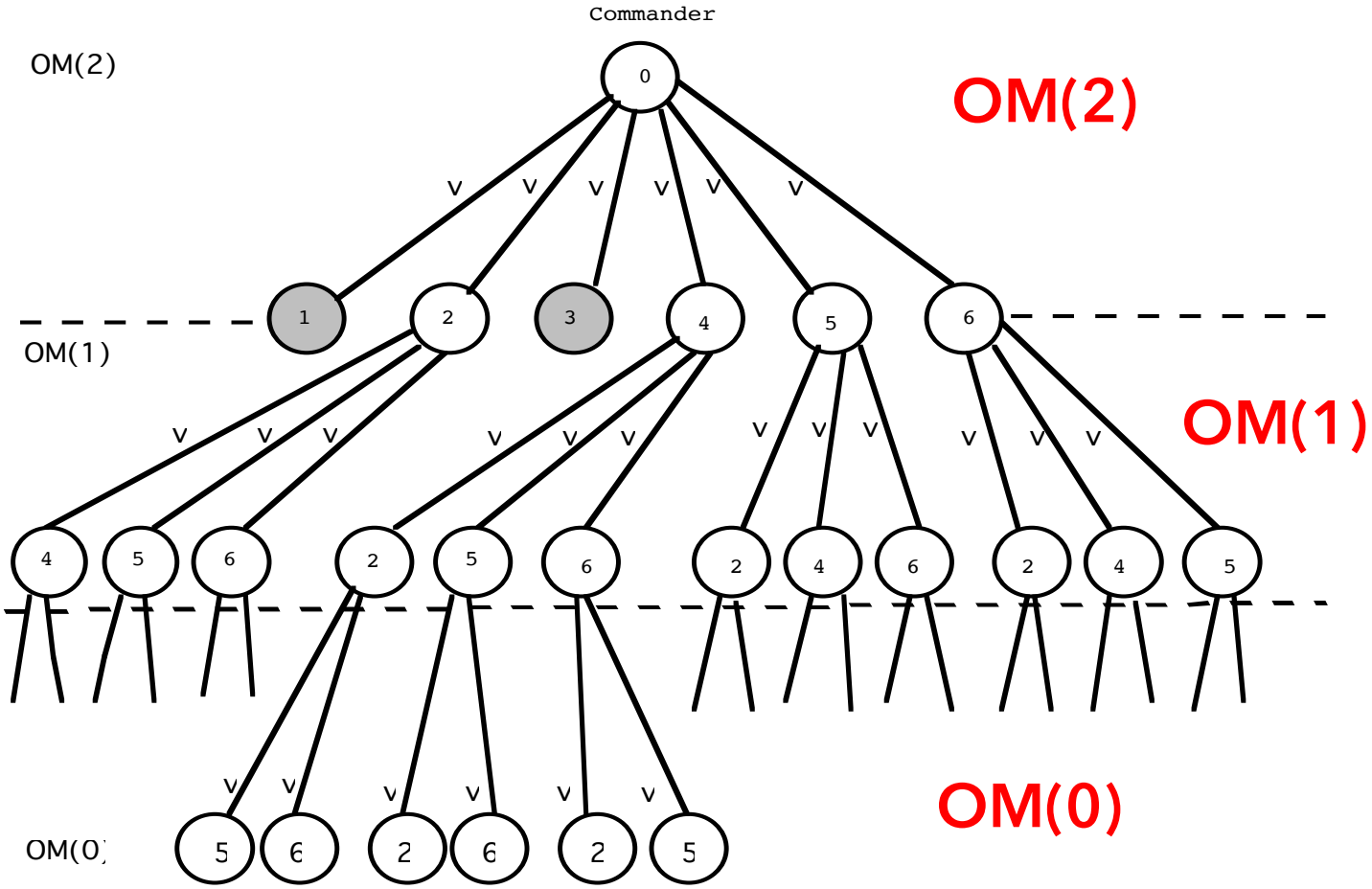
   → one direct value from the commander

   Then he picks the majority of these values as the order from i

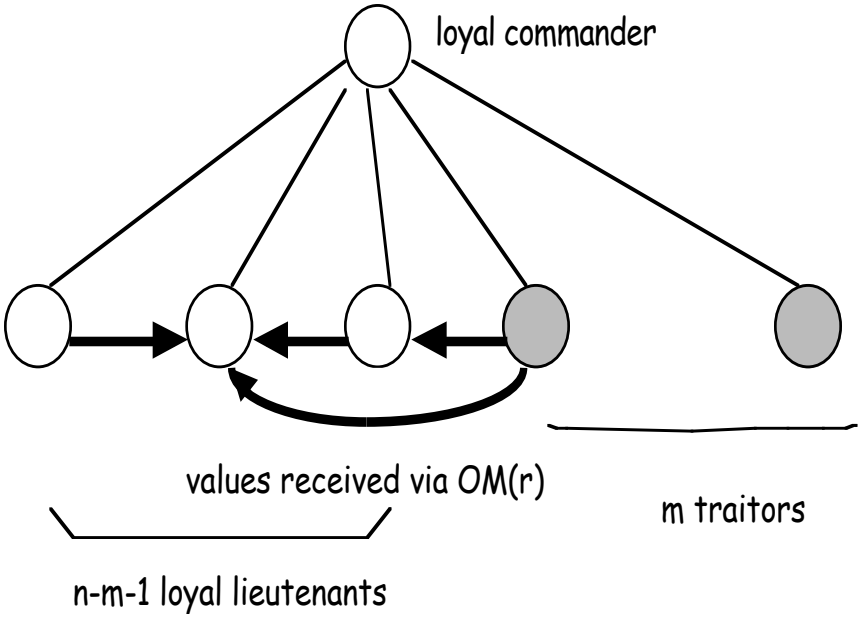
Example of OM(1)



Example of OM(2)



Proof of OM(m)



Lemma:

Let the commander be loyal, and n > 2m + k, where m = maximum number of traitors.

Then OM(k) satisfies IC2

Proof of OM(m)

Proof:

If k=0, then the result trivially holds.

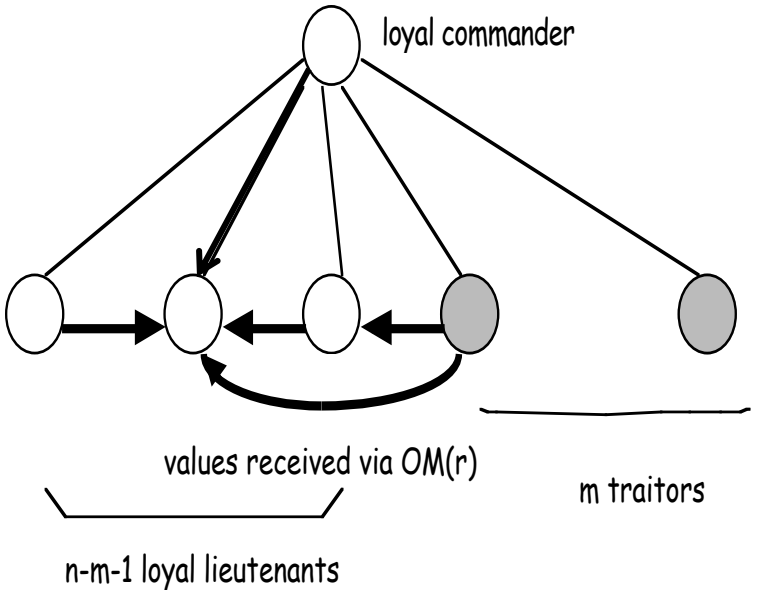
Let it hold for k = r (r > 0) i.e. OM(r) satisfies IC2. We have to show that it holds for k = r + 1 too.

By definition, n > 2m+ r+1, so n-1 > 2m+ r

So OM(r) holds for the lieutenants in the bottom row.

Each loyal lieutenant collects n-m-1 identical good values and m bad values.

So bad values are voted out (n-m-1 > m+r implies n-m-1 > m)



"OM(r) holds" means each loyal lieutenant receives identical values from every loyal commander

The Final theorem

Theorem: If $n > 3m$ where m is the maximum number of traitors, then OM(m) satisfies both IC1 and IC2

Proof: Consider two cases:

Case 1: Commander is loyal

The theorem follows from the previous lemma (substitute $k = m$).

Case 2: Commander is a traitor

We prove it by induction:

Base case: m=0 trivial. (*Induction hypothesis*)
Let the theorem hold for m = r

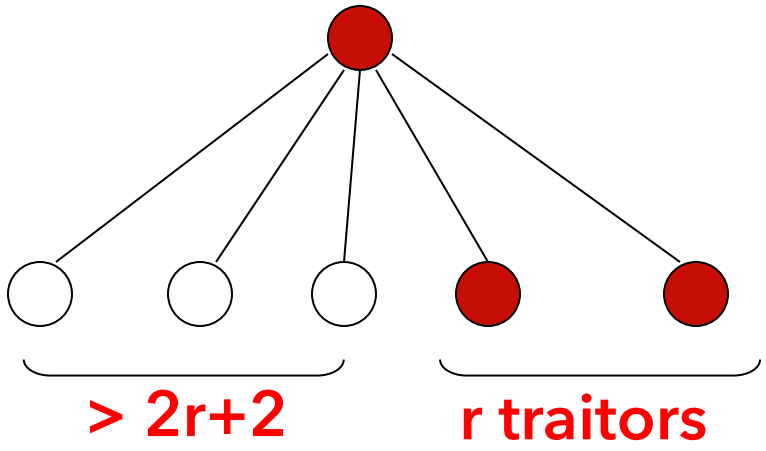
We have to show that it holds for m = r+1 too.

Proof (continued)

There are n > 3(r + 1) generals and r + 1 traitors

Excluding the commander, there are > 3r+2 generals of which there are r traitors. So > 2r+2 lieutenants are loyal.

Since 3r+ 2 > 3r ➔ OM(r) satisfies IC1 and IC2



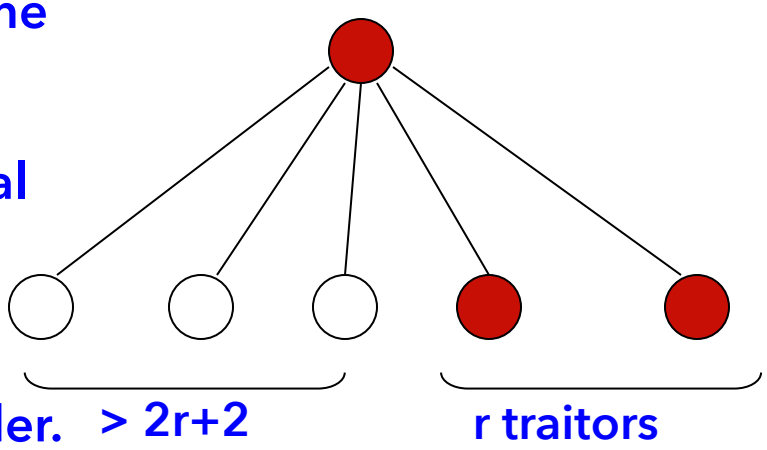
Proof (continued)

In OM(r+1), a loyal lieutenant chooses the majority from

(1) > 2r+1 values obtained from the loyal lieutenants via OM(r),

(2) the r values from the traitors, and

(3) the value directly from the commander.



The set of values collected in part (1) & (3) are the same for all loyal lieutenants - it is the same set of values that these lieutenants received from the commander

Also, by the induction hypothesis, in part (2) each loyal lieutenant receives identical values from each traitor. So every loyal lieutenant eventually collects the same set of values

Applications

The real world applications include

- Block Chain Transactions – Bit Coins
- Clock Synchronization
- PageRank
- Opinion Formation
- Smart Power Grids
- State Estimation
- Control of UAVs (and multiple robots / agents in general)
- Load Balancing and many other domains
- Block Chain
  - Distributed Consensus is really important

Summary

- Recap: Checkpointing
  - Consistent set of checkpoints
  - Synchronous Algo (Koo and Toueg)
  - Asynchronous Algo (Juang & Venkatesan)

- Distributed Consensus
  - Problem Specification
  - Asynchronous Consensus
  - Byzantine Generals / The OM(m) algorithm
  - The Final Theorem

Many more to come up … ! Stay tuned in !!

Penalties

- Every Student is expected to strictly follow a fair Academic Code of Conduct to avoid penalties

- Penalties is heavy for those who involve in:
  - Copy and Pasting the code
  - Plagiarism (copied from your neighbor or friend – in this case, both will get "0" marks for that specific take home assignments)
  - If the candidate is unable to explain his own solution, it would be considered as a "copied case"!!
  - Any other unfair means of completing the assignments

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

How to reach me?

➔ Please leave me an email:

rajendra [DOT] prasath [AT] iiits [DOT] in

➔ Visit my homepage @

➔ https://www.iiits.ac.in/people/regular-faculty/dr-rajendra-prasath/

(OR)

➔ http://rajendra.2power3.com

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- You may ask for one-to-one meeting

Best Approach

- You may leave me an email any time
  (email is the best way to reach me faster)

Questions
It's Your Time

Contact Information

Dr. Rajendra Prasath
IIIT Sri City, Chittoor