Distributed Mutual Exclusion Algorithms

Course: Distributed Computing
Faculty: Dr. Rajendra Prasath
About this topic

This course covers various concepts in Mutual Exclusion in Distributed Systems. We will also focus on different types of distributed mutual exclusion algorithms in distributed contexts and their analysis.
What did you learn so far?

- Challenges in Message Passing systems
- Distributed Sorting
- Space-Time Diagram
- Partial Ordering / Causal Ordering
- Concurrent Events
- Local Clocks and Vector Clocks
- Distributed Snapshots
- Termination Detection
- Topology Abstraction and Overlays
- Leader Election Problem in Rings
- Message Ordering / Group Communications
Topics to focus on ...

- Distributed Mutual Exclusion
- Deadlock Detection
- Check pointing and rollback recovery
- Self-Stabilization
- Distributed Consensus
- Reasoning with Knowledge
- Peer-to-peer computing and Overlays
- Authentication in Distributed Systems
Mutual Exclusion in Distributed Systems

Let us explore mutex algorithms proposed for various interconnection networks
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 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!
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.
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
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 !!

Rajendra, IIIT Sri City
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 \( Q \): \( Q_1 \cap Q_2 \neq \text{empty} \)

⇒ 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 $i, j$: $R_i \cap R_j \neq \emptyset$
- 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 processes 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: $3 \times \sqrt{N}$
- Synchronization delay
  - $2 \times$ (max message transmission time)
- Major problem: DEADLOCK possible
- Need three more types of messages (FAILED, INQUIRE, YIELD) to handle deadlock.
  - Message complexity can be $5 \times \sqrt{N \times N}$
- Important Issue:
  - How to build the request sets?
Token Ring Approach

- 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

- Previous holder of token
- Current holder of token
- Next holder of token
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 $\text{RN}_i[i]$ and send REQUEST( $i$, $\text{RN}_i[i]$) 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 $\text{RN}_j[i] = \max(\text{RN}_j[i], s_n)$
- If j has the token and the token is idle then
  - send it to i if $\text{RN}_j[i] = \text{LN}[i] + 1$
  - 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**
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-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 $T = \text{max. message delay}$
Summary

- Mutual Exclusion
- Various Types of MutEx algorithms
  - Non-Token based algorithm
  - Quorum based algorithm
- Token based algorithm
  - Suzuki – Kasami’s Algorithm
  - Raymond’s Tree based algorithm

Performance Metrics
- Stay tuned ... More to come up ... !!
How to reach me?

👉 Please leave me an email:
rajendra [DOT] prasath [AT] iiits [DOT] in

👉 Visit my homepage @

👉 http://www.iiits.ac.in/FacPages/index-rajendra.html

OR

👉 http://rajendra.2power3.com
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
Thanks ...

... Questions ???

Rajendra, IIIT Sri City