Distributed Mutual Exclusion

Slides are based on the book chapter from Distributed Computing: Principles, Algorithms and Systems (Chapter 9) by Kshemkalyani and Singhal

Pradipta De
pradipta.de@sunykorea.ac.kr
Problem Definition

• Mutual exclusion problem: allow only one process to access a shared resource at any point in time
• Cannot assume shared memory (or shared variables) in the system
• All coordination must be executed using message passing
Solution Approaches

• Three basic approaches
  – Token based
    • A unique token shared among processes decides who gets access to the CS
  – Non-token based
    • Two or more successive rounds of message passing to determine who accesses the CS
  – Quorum based
    • Each process requests permission from a subset of processes to enter the CS
    • Any two quorum contains a common process, who ensures that only one process enters CS
Requirements (Properties)

• Safety: One process executes the CS

• Liveness: No deadlock, no starvation (enters CS within bounded time)

• Fairness: each process gets a fair chance to execute CS \( \Rightarrow \) requests for CS are executed in order of their arrival
Performance Metrics

• **Message Complexity**: The number of messages required per CS execution by a site.

• **Synchronization Delay**: After a site leaves the CS, it is the time required and before the next site enters the CS.
**Performance Metrics**

**Response time**: The time interval a request waits for its CS execution to be over after its request messages have been sent out.

**System Throughput**: The rate at which the system executes requests for the CS. System throughput $= \frac{1}{SD+E}$, where $SD$ is the synchronization delay and $E$ is the average critical section execution time.
Token Based Algorithm

• Assume the processes are organized in a unidirectional ring

• A token represents the permission to enter the CS
  – Token hops from one process to another
  – Process waits in entry (T) till it receives the token
  – On exit (E), forwards the token to the next neighbor in the ring

• Message Complexity: Max number of messages between a T and CS is n (no. of nodes)
Suzuki-Kasami Algorithm

- Consider a fully connected network
  - Wait time for token can be reduced
- If a site wants to enter the CS and it does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
- If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

**Issues to resolve:**
- Which sites have pending request
- Which site to give the token to if multiple requests
- How to distinguish outdated and current request message
Suzuki-Kasami
Handling outdated request

• Uses sequence numbers to mark each request from a process $i$
  – $m^{th}$ CS request from process $j$ is $\text{REQ}(j, m)$

• Receiving process $i$ maintains an array of integers $\text{RN}_i[1, ..., n]$ where $\text{RN}_i[j] = \max$ seq no recv from $j$.
  – $\text{RN}_i[j] := \max(\text{RN}_i[j], m)$.

• When process $i$ receives a message $\text{REQ}(j,m)$, request is outdated if $\text{RN}_i[j] > m$
Suzuki-Kasami
Outstanding requests

• Token:
  – FIFO queue (Q) of requesting processes
  – LN[1…n] : sequence number of request that j executed most recently

• After finishing CS, i updates LN[i]=RN_i[i]
  – Indicates that req corresponding to RN_i[i] is complete

• If at i, RN_i[j]=LN_i[j] + 1, then j is requesting a token
Suzuki-Kasami

**Requesting the critical section**
(a) If proc i does not have the token at T, then it increments its seq no, RN\(_i\)[i], and sends a REQUEST(i, sn) message to all other sites.
(b) When j receives this message, it sets RN\(_j\)[i] to max(RN\(_j\) [i], sn).
If j has the idle token, then it sends the token to i if RN\(_j\)[i]=LN\(_j\)[i]+1.

**Executing the critical section**
(c) Process i executes the CS after it has received the token.

**Releasing the critical section:** After i has finished CS,
(d) *Mark token Q to mark completion of i req:* Set LN[i] = RN\(_i\)[i].
(e) *Update token Q:* For every j whose id is not in the token queue, it appends its id to the token queue if RN\(_i\)[j]= LN[j]+1.
(f) *Pop the top id:* If the token queue is nonempty after the above update, i deletes the top site id from the token queue and sends the token to the site indicated by the id.
Initial State

RN=[1,0,0,0,0]
LN=[0,0,0,0,0]
Q=[]
1 and 2 send request to enter CS
0 exits CS ➔ token idle

RN = [1, 1, 1, 0, 0]
LN = [1, 0, 0, 0, 0]
Q = [1, 2]
0 passes token to 1
0 and 3 request tokens

RN = [2, 1, 1, 1, 0]

0

1

2

3

4

RN = [2, 1, 1, 1, 0]

RN = [2, 1, 1, 1, 0]

RN = [2, 1, 1, 1, 0]

RN = [2, 1, 1, 1, 0]

LN = [1, 0, 0, 0, 0]

Q = [2, 0, 3]
1 sends token to 2

RN=[2,1,1,1,0]  RN=[2,1,1,1,0]
0

RN=[2,1,1,1,0]  RN=[2,1,1,1,0]
1

RN=[2,1,1,1,0]  RN=[2,1,1,1,0]
2

RN=[2,1,1,1,0]  LN=[1,1,0,0,0]  Q=[0,3]
3

RN=[2,1,1,1,0]
4
Suzuki-Kasami

- **Message complexity:**
  - If node holds the token, then messages = 0
  - Otherwise, can be worst case n

- **Synchronization**
  - If node has the token, then 0
  - Else, maximum message passing delay

- **What about starvation?**
Lamport’s Algorithm

• Non-token based approach
• Every \( i \) has a request queue, \( Q_i \)
  – Requests are sorted by logical timestamps

• Communication channels are FIFO
Lamport’s Algorithm

• In entry block (T), when i wants to enter CS
  – Broadcast REQUEST(tsi, i) to all other procs
  – Put (tsi, i) in Qi
• On receiving request at j,
  – Place i’s request in Qj
  – Send timestamped REPLY to i
• Execute CS by i:
  – (tsi, i) is at the top if its own queue Qi
  – Process i has received any message with timestamp larger than (tsi, i) from ALL other nodes.
• Release CS by i:
  – Remove request from top of Qi
  – Send timestamped RELEASE msg to all other nodes
  – When j receives RELEASE from i, then remove i’s from Qj
S1, S2 req for CS
Broadcast REQ

S1 enters the CS;
Total order enforced by sorting using process id

S1 completes CS;
Broadcasts REPLY message

S1 completes CS;
Broadcasts REPLY message
Lamport’s Algo

• Message Complexity
  – N-1 REQUEST msg; N-1 REPLY msg; N-1 RELEASE msg
  ➞ 3(N-1) messages per CS invocation

• Synchronization delay
  – Max message transmission time

• How to reduce the message complexity?
  – Can be between 3(N-1) and 2(N-1)
Ricart-Agrawala Algorithm

- Optimization over Lamport’s algorithm
  - Do not need to send the RELEASE message
  - Does not require FIFO channel

- Main Idea
  - Process $i$ sends a broadcast REQUEST to all others
  - If [not executing CS] OR [not requesting CS] OR [$i$'s REQ timestamp is lower than $j$'s REQ timestamp]
    - Process $j$ sends a REPLY
  - Else [defer REPLY] if proc $j$ has a REQUEST with a timestamp lower than $i$'s REQUEST
    - Send REPLY after completing CS

- Proci $i$ cannot enter before $j$ if the request has higher timestamp
Illustration

No need of RELEASE message

Message Complexity: $2(N-1)$

Can the message complexity be even lower?
Quorum-based ME

- A coterie \( C \) is defined as a set of sets, where each set \( g \in C \) is called a quorum.

- Properties of quorum:
  - Intersection: For every quorum \( g, h \in C \), \( g \cap h \neq \emptyset \)
    - \( \{1,2,3\}, \{2,5,7\} \) and \( \{5,7,9\} \) cannot be quorums in a coterie
    - \( \{1,2,3\}, \{3,5,6\} \) and \( \{2,4,5\} \) can be quorums in a coterie
  - Minimality: There should be no quorums \( g, h \) in coterie \( C \) such that \( g \supseteq h \)
    - \( \{1,2,3\} \) and \( \{1,3\} \) are not quorums in a coterie
Basic Idea: Quorum based ME

- P1 wants to access CS \(\Rightarrow\) requests access from the quorum A.
- A contains at least one site common to the quorum of every other site.
- The common sites send permission to one site at a time.
Maekawa’s Algorithm

- First quorum based algorithm
- Satisfies the following conditions

  M1: (\( \forall i \, \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset \) )

  M2: (\( \forall i : 1 \leq i \leq N :: S_i \in R_i \) )

  M3: (\( \forall i : 1 \leq i \leq N :: |R_i| = K \) )

  M4: Any site \( S_j \) is contained in \( K \) number of \( R_i \)s, \( 1 \leq i, j \leq N \).

- Size of \( R_i \): \( |R_i| = \sqrt{N} \)
- What is the use of each condition?
Maekawa’s Algorithm

• Requesting CS:
  – i sends message to all processes in $R_i$

• On receiving REQUEST msg:
  – Send REPLY if no REPLY sent since last RELEASE; Update REPLY sent status
  – Else, queue up REQUEST

• Enter CS:
  – After receiving REPLY from all nodes in $R_i$

• Release CS:
  – Send RELEASE to all nodes in $R_i$
  – Update status; send the next REPLY for the queued up REQUEST
Problem in Maekawa’s Algo

Requests are NOT prioritized by timestamp

P1 \rightarrow \{1,2,3\}
   p1, p2 replies 1, but p3 replies 2
P2 \rightarrow \{3,5,6\}
   p3, p6 replies 2, but p5 replies 3
P3 \rightarrow \{2,4,5\}
   p5, p4 replies 3, but p2 replies 1

Waiting for reply:
1 waits for reply from p3
2 waits for reply from p5
3 waits for reply from p2

Liveness is not guaranteed
Avoiding Deadlock: Maekawa

• One way to avoid is to ensure that messages are received in order of timestamp
  – Too strong assumption with variable message delays

• Introduce 3 additional messages
  – Failed; inquire; relinquish

• Handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock.

• A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.