Distributed Mutual Exclusion

CS60002: Distributed Systems

Pallab Dasgupta
Professor,
Dept. of Computer Sc. & Engg.,
Indian Institute of Technology Kharagpur

Mutual Exclusion

- Very well-understood in shared memory systems

- Requirements:
  - at most one process in critical section (safety)
  - if more than one requesting process, someone enters (liveness)
  - a requesting process enters within a finite time (no starvation)
  - requests are granted in order (fairness)

Types of Dist. Mutual Exclusion Algorithms

- **Non-token based / Permission based**
  - Permission from all processes: e.g. Lamport, Ricart-Agarwala, Raicourol-Carvalho etc.
  - Permission from a subset: ex. Maekawa

- **Token based**
  - ex. Suzuki-Kasami

Some Complexity Measures

- No. of messages/critical section entry
- Synchronization delay
- Response time
- Throughput

Lamport's Algorithm

- Every node i has a request queue q_i
  - keeps requests sorted by logical timestamps (total ordering enforced by including process id in the timestamps)

- To request critical section:
  - send timestamped REQUEST(tsi, i) to all other nodes
  - put (tsi, i) in its own queue

- On receiving a request (tsi, i):
  - send timestamped REPLY to the requesting node i
  - put request (tsi, i) in the queue

Lamport's Algorithm contd..

- **To enter critical section:**
  - **Process *i* enters critical section if:**
    - **(*tsi*, *i*) is at the top if its own queue, and**
    - **Process *i* has received a message (any message) with timestamp larger than (*tsi*, *i*) from ALL other nodes.**

- **To release critical section:**
  - **Process *i* removes its request from its own queue and sends a timestamped RELEASE message to all other nodes**
  - **On receiving a RELEASE message from *i*, *i*'s request is removed from the local request queue**

Some notable points

- **Purpose of REPLY messages from node *i* to *j* is to ensure that *j* knows of all requests of *i* prior to sending the REPLY (and therefore, 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 mesg transmission time**
- **Requests are granted in order of increasing timestamps**

The Ricart-Agrawala Algorithm

- Improvement over Lamport's
- Main Idea:
  - node *j* need not send a REPLY to node *i* if *j* has a request with timestamp lower than the request of *i* (since *i* cannot enter before *j* anyway in this case)
- Does not require FIFO
- 2(n – 1) messages per critical section invocation
- Synchronization delay = max. message transmission time
- Requests granted in order of increasing timestamps

The Ricart-Agrawala Algorithm

- To request critical section:
  - send timestamped REQUEST message (*tsi*, *i*)
- On receiving request (*tsi*, *i*) at *j*:
  - send REPLY to *i* if *j* is neither requesting nor executing critical section or
  - if *j* is requesting and *i*'s request timestamp is smaller than *j*'s request timestamp. Otherwise, defer the request.
- To enter critical section:
  - *i* enters critical section on receiving REPLY from all nodes
- To release critical section:
  - send REPLY to all deferred requests

Roucairol-Carvalho Algorithm

- Improvement over Ricart-Agarwala
- Main idea
  - Once i has received a REPLY from j, it does not need to send a REQUEST to j again unless it sends a REPLY to j (in response to a REQUEST from j)
  - Message complexity varies between 0 and $2(n - 1)$ depending on the request pattern
  - worst case message complexity still the same

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

- 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 D

- $K = D = \sqrt{N}$ for Maekawa's

A Simple Version

- **To request critical section:**
  - 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:**
  - i enters critical section after receiving REPLY from all nodes in R_i

A Simple Version 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.

Features

- **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)
- **Building the request sets?**

Token based Algorithms

- 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

Suzuki Kasami 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

Suzuki Kasami Algorithm

- **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 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, sn) at j:
  - Set $RN_j[\ i\ ] = \max(RN_j[\ i\ ], sn)$
  - 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 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

Notable features

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

Raymond's Algorithm

- **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

- 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-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-empty, send a REQUEST message to the parent (node pointed at by *Holder* variable)

Notable features

- Average message complexity: O(log n)

- Sync. delay = (T log n)/2, where T = max. message delay