Mutual Exclusion (ME)

- In a single-processor system, ME can be achieved with semaphores, lock variables, monitors, etc.
- In dist systems, ME is more complex due to no sh-mem, timing (comm delays and clocks) and ordering of events
- Two basic approaches of ME in dist systems can be identified:
  - Centralized: Master and slaves, master dictates actions
  - Distributed: each site decides on actions, based on own state
- Distributed algorithms can be subdivided into:
  - **Token based**: a site is allowed in the CS if has a token. Tokens are passed from site to site, in some priority order
  - **Non-token based**: site enters CS when an assertion becomes TRUE. A site communicates with others to get information about the other sites' states, and based on this, decides whether assertions is TRUE or FALSE. This communication is usually based on timestamps

ME requirements

- For a solution to be correct, the algorithms must have the following properties
  - Deadlock freedom
  - Starvation freedom
  - Fairness: give everyone a chance, in certain systems it also means to execute the requests as they are made (e.g., Logical FIFO)
- Performance metrics
  - Number of messages sent, received, or both
  - Synchronization delay
  - Response time
  - System throughput
- Conditions under which must test performance
  - High vs. low load behaviors
  - Worst vs best vs average case behaviors
Simple Solution: Centralized ME

- Centralized approach: Master holds a list of processes requesting the CS; grants requests in some order (random, FIFO, etc)

  ![Diagram of Centralized ME]

  - Master
  - Site 1
  - Site 2
  - Site 3
  - Site 1 has critical section; will yield to the master; master will grant to site x
  - List of requests

- Advantages: fair, correct, no starvation, simple to implement
- Disadvantages: single point of failure and bottleneck

  ![Diagram of Advantages and Disadvantages]

  - Bottleneck
  - Network congestion
  - Timeout

Distributed ME

- Problem: if messages are not ordered correctly, the CS can be acquired improperly (remember that if there are no global timestamps, the “earliest” request should gain the CS)
- In the example below, P3 thinks that P1 should have the CS (P3 got P1’s message first), while both P1 and P2 think that P2 should have it.
- In a distributed algorithm, the decision must made independently from other nodes in the system, AND the decision is the same.

  ![Diagram of Distributed ME]

  - Time
  - P1
  - P2
  - P3
  - Request for CS
Distributed ME

- Lamport’s algorithm solves the problem above, with the logical clocks [Lamport 78: Clocks, Messages, and the Pursuit of Happiness]
- To request CS:
  - send req message M to all processes;
  - enQ M in its own queue
- Upon receiving request from Pi: enQ and send ack
- Upon receiving release from Pi: deQ
- To release CS:
  - send ack message to all procs
  - remove it from Q
- To acquire CS: enter CS when got a message with a larger timestamp from every other proc AND has own message with smallest TS in own Q
  Note that to enter the CS, checks must only be made locally

Distributed ME

- Ricart and Agrawala’s Algorithm
- To request CS:
  - send req message to M to all processes;
  - enQ M in its own queue
- Upon receiving M from Pi:
  - If it doesn’t have/want CS send ack
  - If it has CS, enQ request Pi:M
  - If it wants CS, either enQ or ack
- To release CS:
  - send ack message to all procs in the Q
  - remove them (procs) from Q
- To acquire CS: enter CS when got ack from every other proc
Ricart and Agrawala’s Algorithm (Cont)

- The main idea is that lowest timestamp wins
- Also, the acks AND permission messages are combined into acks, which are only send out after a proc used the CS (if it has smaller timestamp)
- Example:
  - Node A req CS with ts=8, Node C with ts=12.
  - B acks both A&C
  - C acks A (smaller ts).
  - A uses CS, then sends ack to C.

Maekawa’s Algorithm

- Maekawa presents an algorithm where a subset of nodes can give permission to a node to enter the CS
- It is sufficient for a majority of the nodes to give permission
- In Maekawa’s algorithm, $|G| = \sqrt{n} + 1$, G is the subset.
- There are $\sqrt{n}$ subsets. By design, there is always a node in the intersection of 2 subsets.
- To request CS: send request message M to all processes in G(i)
- Upon receiving a request M from Pi: if it hasn’t sent a reply since last release, send a reply. Otherwise, queue it
- To release CS: send release to all procs in G(i)
- Upon receiving a release from Pi: send a reply to head of Q and remove it from Q; if Q is empty, update state (no reply sent)
- To acquire CS: enter CS when got ack from all other proc in G(i)
Maekawa’s Algorithm

- For example, $G(a) = \{b, d, e\}$; $G(c) = \{a, b, f\}$; $G(g) = \{c, d, f\}$
- Say sites $a, c, g$ make requests simultaneously
- If site $b$ responds positively to $a$, what happens?
- Site $b$ queues the response to site $c$ until it gets a release from $a$, but it could send positive responses to site $a$
- Meanwhile site $f$ responds positively to site $g$, but then must queue the request for site $c$
- Eventually, site $a$ will send a release to its subset of nodes, and site $b$ will then respond positively to $c$

Maekawa’s Alg Problem

- The biggest problem in Maekawa’s algorithm is that it may lead to deadlocks. HOW? **Homework: show a sequence of events that may lead to deadlocks; due next class; simple answer.**
- Note that there is no ordering in the messages that are sent to the subsets of authorizing nodes. Also, there are communication delays that may cause the deadlocks.
- Solution: One of the initiates a deadlock resolution phase.
- New types of messages are needed:
  - FAILED: message sent to a site (after a REPLY) when a higher-priority request was received. It’s a
  - INQUIRE: message sent to a site to check whether the site was able to acquire all locks
  - YIELD: a message sent in reply to INQUIRE message, releasing the locks (in case it didn’t get all locks)
  - GRANT: a message sent to a site, granting that site a lock
Maekawa’s Alg Modifications

• Upon receiving a request M form Pi; if it hasn’t sent a reply since last release, send a reply. Ow, if new req. has a larger timestamp, send a FAILED message; ow, if new req has a smaller timestamp, send a INQUIRE message to the site of the previous REPLY.
• Upon receiving an INQUIRE message, a site ignores it if it already got all grants from its subset. If the site received a FAILED message from any site, it sends a YIELD message. IF a site sent a YIELD message, but got no GRANT message from another site in the request set, it sends a YIELD message.
• Upon receiving a YIELD from a site, a member of a request site assumes that the site has release the locks and processes the request normally.
• Number of messages required for the algorithm to succeed?
• Homework: show a sequence of events that solves the deadlocks with new messages; due next class; simple answer.

Token Based Dist ME

• In the token based approaches to distributed ME, the general correctness of the algorithm is easier to prove, since the site will only go into the CS if it holds a token.
• On the other hand, there are more problems to prove freedom of starvation (since a site may retain a token forever)
• Another problem is when a site fails: there has to be a regeneration of the tokens, which is similar to leader election
• One representative algorithm for token-based tree-based ME is Raymond’s algorithm
Other Approaches

- Rings: physical or logical (any physical interconnection, eg, BUS, point-to-point nets, etc)
- Usually uses a **token**: the proc that has the token enters the CS
- It’s fair, since CS is round-robin
- It’s correct, since there is a single token
- Problems: lost-token, duplicated token. Need to be able to detect and regenerate/delete token
- If a process crashes/fails, what to do? Ack?

<table>
<thead>
<tr>
<th></th>
<th>Msg/CS</th>
<th>Delay</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2(n-1)</td>
<td>2(n-1)</td>
<td>N points of failure</td>
</tr>
<tr>
<td>Token ring</td>
<td>0 to n-1</td>
<td>0 to n-1</td>
<td>Lost/duplic token</td>
</tr>
</tbody>
</table>