Mutual Exclusion

♦ Requirements for Mutual Exclusion Algorithms in Message-Passing Based Distributed Systems
  ‣ ME1: at most one process may execute in the critical section at any given point in time (safety)
  ‣ ME2: requests to enter or exit the critical section will eventually succeed (liveness)
    – impossible for one process to enter critical section more than once while other processes are awaiting entry
  ‣ ME3: if one request to enter the critical section is issued before another request (as per the → relation), then the requests will be served in the same order
Mutual Exclusion

♦ **Performance criteria to be used in the assessment of mutual exclusion algorithms**
  ‣ **bandwidth** consumed (corresponds to number of messages sent)
  ‣ **client delay** at each entry and exit
  ‣ **throughput**: number of critical region accesses that the system allows
    – here: measured in terms of the synchronization delay between one process exiting the critical section and the next process entering
Mutual Exclusion

♦ Central Server-based Algorithm
  ‣ central server receives access requests
    – if no process in critical section, request will be granted
    – if process in critical section, request will be queued
  ‣ process leaving critical section
    – grant access to next process in queue, or wait for new requests if queue is empty

♦ Properties
  ‣ satisfies ME1 and ME2, but not ME3 (network delays may reorder requests)
  ‣ two messages per request, one per exit, exit does not delay process
  ‣ performance and availability of server are the bottlenecks
Mutual Exclusion

♦ Ring-based Algorithm

- logical, not necessarily physical link: every process \( p_i \) has connection to process \( p_{(i+1) \mod N} \)
- token passes in one direction through the ring
- token arrival
  - only process in possession of token may access critical region
  - if no request upon arrival of token, or when exiting critical region, pass token on to neighbour
Mutual Exclusion

♦ Ring-based Algorithm

- satisfies ME1 and ME2, but not ME3
- performance
  - constant bandwidth consumption
  - entry delay between 0 and N message transmission times
  - synchronization delay between 1 and N message transmission times
Mutual Exclusion

On initialization
state := RELEASED;

To enter the section
state := WANTED;
Multicast request to all processes;
\( T := \) request’s timestamp;
Wait until (number of replies received = \( N - 1 \));
state := HELD;

On receipt of a request \( <T_i, p_i> \) at \( p_j \) \( (i \leq j) \)
if (state = HELD or (state = WANTED and \( (T, p_j) < (T_i, p_i) \)))
then
queue request from \( p_i \) without replying;
else
reply immediately to \( p_i \);
end if

To exit the critical section
state := RELEASED;
reply to any queued requests;

- Algorithm by Ricart and Agrawala
  - based on multicast
    - process requesting access multicasts request to all other processes
    - process may only enter critical section if all other processes return positive acknowledgement messages
  - assumptions
    - all processes have communication channels to all other processes
    - all processes have distinct numeric ID and maintain logical clocks
Mutual Exclusion

<table>
<thead>
<tr>
<th>Algorithm by Ricart and Agrawala</th>
</tr>
</thead>
<tbody>
<tr>
<td>if request is broadcast and state of all other processes is RELEASED, then all processes will reply immediately and requester will obtain entry</td>
</tr>
<tr>
<td>if at least one process is in state HELD, that process will not reply until it has left critical section, hence mutual exclusion</td>
</tr>
<tr>
<td>if two or more processes request at the same time, whichever process’ request bears lower timestamp will be the first to get N-1 replies</td>
</tr>
<tr>
<td>in case of equal timestamps, process with lower ID wins</td>
</tr>
</tbody>
</table>
Algorithm by Ricart and Agrawala

- p_3 not attempting to enter, p_1 and p_2 request entry simultaneously
- p_3 replies immediately
- p_2 receives request from p_1, timestamp(p_2) < timestamp(p_1), therefore p_2 does not reply
- p_1 sees its timestamp to be larger than that of the request from p_2, hence it replies immediately and p_2 is granted access
- p_2 will reply to p_1’s request after exiting the critical section
Algorithm by Ricart and Agrawala

- $p_3$ not attempting to enter, $p_1$ and $p_2$ request entry simultaneously
- $p_3$ replies immediately
- $p_2$ receives request from $p_1$, $\text{timestamp}(p_2) < \text{timestamp}(p_1)$, therefore $p_2$ does not reply
- $p_1$ sees its timestamp to be larger than that of the request from $p_2$, hence it replies immediately and $p_2$ is granted access
- $p_2$ will reply to $p_1$’s request after exiting the critical section
Mutual Exclusion

On initialization
  state := RELEASED;
To enter the section
  state := WANTED;
  Multicast request to all processes;  processing of incoming requests deferred here
  T := request’s timestamp;
  Wait until (number of replies received = (N - 1));
  state := HELD;
On receipt of a request <T_i, p_i> at p_j (i ≤ j)
  if (state = HELD or (state = WANTED and (T, p_j) < (T_i, p_i)))
    then
      queue request from p_i without replying;
    else
      reply immediately to p_i;
  end if
To exit the critical section
  state := RELEASED;
  reply to any queued requests;

♦ Algorithm by Ricart and Agrawala
  algorithms satisfies ME1
    – two processes p_i and p_j can only access critical section at the same time in case they would have replied to each other
    – since pairs <T_i, p_i> are totally ordered, this cannot happen
  algorithms also satisfies ME2 and ME3
Mutual Exclusion

On initialization
  state := RELEASED;

To enter the section
  state := WANTED;
  Multicast request to all processes; \{ processing of incoming requests deferred here \}
  \begin{align*}
  T &:= \text{request's timestamp;} \\
  \text{Wait until (number of replies received} &= (N - 1)); \\
  \text{state} &:={ \text{HELD}};
\end{align*}

On receipt of a request \(<T_i, p_i>\) at \(p_j\) \((i \leq j)\)
  \begin{align*}
  &\text{if (state = HELD or (state = WANTED and (}\color{blue}{T, p_j} < (T_i, p_i))))) \\
  &\text{then} \\
  &\text{queue request from } p_i \text{ without replying;} \\
  &\text{else} \\
  &\text{reply immediately to } p_i;
  \end{align*}
  \text{end if}

To exit the critical section
  state := RELEASED;
  reply to any queued requests;

\begin{itemize}
\item Algorithm by Ricart and Agrawala
  \begin{itemize}
  \item performance
    \begin{itemize}
    \item getting access requires \(2(N-1)\) messages per request
    \item synchronization delay: just one round-trip time (previous algorithms: up to \(N\))
    \end{itemize}
  \end{itemize}
\end{itemize}
Mutual Exclusion

On initialization
state := RELEASED;

To enter the section
state := WANTED;
Multicast request to all processes;

processing of incoming requests deferred here
T := request’s timestamp;

Wait until (number of replies received = (N – 1));
state := HELD;

On receipt of a request <T_i, p_i> at p_j (i ≤ j)
if (state = HELD or (state = WANTED and (T, p_j) < (T_i, p_i)))
then
queue request from p_i without replying;
else
reply immediately to p_i;
end if

To exit the critical section
state := RELEASED;
reply to any queued requests;

♦ Algorithm by Ricart and Agrawala
  ‣ protocol improvements
    ‒ repeated entry of same process without executing protocol
    ‒ optimization possible to N messages per request
Mutual Exclusion

♦ Maekawa’s Voting Algorithm
  
  ♦ observation
    – to get access, not all processes have to agree
    – suffices to split set of processes up into subsets (“voting sets”) that overlap
    – suffices that there is consensus within every subset
  
  ♦ model
    – processes $p_1, .., p_N$
    – voting sets $V_1, .., V_N$ chosen such that $\forall \ i,k$ and for some integer $M$:
      
      $p_i \in V_i$
      
      $V_i \cap V_k \neq \emptyset$ (some overlap in every voting set)
      
      $| V_i | = K$ (fairness: all voting sets have equal size)
      
      each process $p_k$, is contained in $M$ voting sets
Mutual Exclusion

♦ Maekawa’s Voting Algorithm
  ‣ protocol
    – to obtain entry to critical section, $p_i$ sends request messages to all $K-1$ members of voting set $V_i$
    – cannot enter until $K-1$ replies received
    – when leaving critical section, send release to all members of $V_i$
    – when receiving request
      • if state = HELD or already replied (voted) since last request
        * then queue request
      • else immediately send reply
    – when receiving release
      • remove request at head of queue and send reply
Mutual Exclusion

♦ Maekawa’s Voting Algorithm

On initialization
  state := RELEASED; voted := FALSE;
For \( p_i \) to enter the critical section
  state := WANTED;
  Multicast request to all processes in \( V_i - \{p_i\} \);
  Wait until (number of replies received = \( K - 1 \));
  state := HELD;
On receipt of a request from \( p_i \) at \( p_j \) (\( i \neq j \))
  if (state = HELD or voted = TRUE)
    queue request from \( p_i \) without replying;
  else
    send reply to \( p_i \);
    voted := TRUE;
  end if
For \( p_i \) to exit the critical section
  state := RELEASED;
  Multicast release to all processes in \( V_i - \{p_i\} \);
On receipt of a release from \( p_i \) at \( p_j \) (\( i \neq j \))
  if (queue of requests is non-empty)
    then
      remove head of queue – from \( p_k \), say;
      send reply to \( p_k \);
      voted := TRUE;
    else
      voted := FALSE;
  end if

© Addison-Wesley Publishers 2000
Mutual Exclusion

♦ Maekawa’s Voting Algorithm

› optimization goal: minimize K while achieving mutual exclusion
  – can be shown to be reached when $K \sim \sqrt{N}$ and $M=K$
› optimal voting sets: nontrivial to calculate
  – approximation: derive $V_i$ so that $|V_i| \sim 2\sqrt{N}$
    • place processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix
    • let $V_i$ the union of the row and column containing $p_i$
Mutual Exclusion

♦ Maekawa’s Voting Algorithm

- satisfies ME1
  - if possible for two processes to enter critical section, then processes in the non-empty intersection of their voting sets would have both granted access
  - impossible, since all processes make at most one vote after receiving request

- deadlocks are possible
  - consider three processes with
    - \( V_1 = \{p_1, p_2\}, V_2 = \{p_2, p_3\}, V_3 = \{p_3, p_1\} \)
  - possible to construct cyclic wait graph
    - \( p_1 \) replies to \( p_2 \), but queues request from \( p_3 \)
    - \( p_2 \) replies to \( p_3 \), but queues request from \( p_1 \)
    - \( p_3 \) replies to \( p_1 \), but queues request from \( p_2 \)
Mutual Exclusion

♦ Maekawa’s Voting Algorithm
  ‣ algorithm can be modified to ensure absence of deadlocks
    – use of logical clocks
    – processes queue requests in happened-before order
    – means that ME3 is also satisfied
  ‣ performance
    – bandwidth utilization
      • $2\sqrt{N}$ per entry, $\sqrt{N}$ per exit, total $3\sqrt{N}$ is better than Ricart and Agrawala for $N>4$
    – client delay
      • same as for Ricart and Agrawala
    – synchronization delay
      • round-trip time instead of single-message transmission time in Ricart and Agrawala
Mutual Exclusion

Notes on Fault Tolerance

- none of these algorithms tolerates message loss
- ring-algorithms cannot tolerate single crash failure
- Maekawa’s algorithm can tolerate some crash failure
  - if process is in a voting set not required, rest of the system not affected
- Central-Server: tolerates crash failure of node that has neither requested access nor is currently in the critical section
- Ricart and Agrawala algorithm can be modified to tolerate crash failures by the assumption that a failed process grants all requests immediately
  - requires reliable failure detector