Frequently asked questions from the previous class survey

- Examples of distributed mutual exclusion?
- Central server grants “permission” by granting processes a “token”
- Is this permission based or token based?

Topics covered in this lecture

- Distributed Mutual Exclusion
  - Multicast & logical clocks [Agarwala & Ricart]
  - Maekawa’s voting based algorithm

Logical Clocks: If two processes do not interact with each other

- Their clocks need not be synchronized
- Lack of synchronization is not observable
  - Does not cause problems

Lamport’s logical clocks

- The happens-before relation
  - a and b are events in the process; and a occurs before b
    - Then a → b is true
  - a is event of message sent by one process; b is event of message being received in another process
    - Then a → b is true
Some more things about the happens-before relation

- If $a \Rightarrow b$ and $b \Rightarrow c$, then $a \Rightarrow c$
  - Transitive

- If events $x$ and $y$ occur in processes that do not exchange messages, then ...
  - $x \Rightarrow y$ is not true
  - But, neither is $y \Rightarrow x$
  - These events are said to be concurrent

An example of Lamport’s algorithm:

Implementing Lamport’s clocks

1. Before executing an event, $P_i$ executes
   
   $C_i \leftarrow C_i + 1$

2. When $P_i$ sends a message $m$ to $P_j$, it sets $m$’s timestamp $ts(m)$ to $C_i$ in previous step

3. Upon receipt of message $m$, $P_j$ adjusts its own local counter
   
   $C_j \leftarrow \max\{C_j, ts(m)\}$

   do step (1) and deliver message

An application of Lamport’s clock:
User has $1000 in bank account initially

- Add $100 to account
- Update with 1% interest

$100 + 1\% \text{ interest} = 100 + 0.01 \times 100 = 101$
Balance: $1000 + 101 = 1101$

There is a difference when the orders are reversed

- Our objective for now is consistency
- Both copies must be exactly the same

- Situations like these require totally-ordered multicast
  - All messages are delivered in the same order to each receiver
  - Lamport’s logical clocks allow us to accomplish this in a completely distributed fashion
Using Lamport’s clock to order messages

- Process puts received messages into local queue
  - Ordered according to the message’s timestamp
- Message can be delivered only if it is acknowledged by all the other processes
- If a message is at the head of the queue, and acknowledged by all processes
  - It is delivered and processed

Other types of logical clocks

- Vector clocks
- Matrix clocks

Getting back to distributed mutual exclusion ...

Agarwala & Ricart’s algorithm using multicast and logical clocks

- Processes that require entry to a critical section multicast a request message
  - Enter it only when all other processes have replied to request
- Process’ replies to a request are designed to ensure that ME1, ME2, and ME3 are met

The setting

- Processes \( p_1, p_2, \ldots, p_N \) have distinct identifiers
- Processes have communication channels to each other
- Each process \( p_i \) keeps a Lamport clock
- Messages requesting entry are of the form \(<T, p_i>\)
  - \( T \) is the sender’s timestamp and \( p_i \) is the sender’s identifier

Each process records its state

- Released
  - Outside the critical section
- Wanted
  - Wanting entry into the critical section
- Held
  - Being in the critical section
Entering the critical section [1/2]

- If a process requests entry and the state of all other processes is Released
  - All processes respond immediately and the entry is granted

- If a process requests entry and some process is in the state Held
  - That holding process will not reply to requests until it has finished with the critical section
  - All other processes respond

Entering the critical section [2/2]

- If two or more processes request entry at the same time?
  - Request with the lowest timestamp will be first to collect N-1 replies
  - If the Lamport timestamps are the same?
    - Requests are ordered based on their identifiers
  - When a process requests entry?
    - Defers all processing requests from other processes until its own request has been sent

Multicast synchronization

Initial Condition:
- p₂ not interested in entering critical section
- p₁ and p₂ request entry concurrently
- Timestamp of p₁’s request: 41
- Timestamp of p₂’s request: 34

p₂ enters the critical section

Achieving the properties ME1, ME2 and ME3

- If two processes pᵢ and pⱼ (i ≠ j) enter critical section at the same time?
  - Both these processes would have replied to each other; but the pairs <Tᵢ, pᵢ> are totally ordered
  - So it’s impossible
  
- Requests to enter and exit the critical section eventually succeed because requests are served based on timestamps
  - Satisfies ME2 and ME3 (order)

Evaluation of the algorithm

- Gaining entry takes 2(N-1) messages
  - N-1 to multicast the request
  - Followed by N-1 replies
  - Expensive in terms of bandwidth utilization

- Synchronization delay
  - Just one message transmission time
  - Previous algorithms incurred round-trip delays

Maekawa’s Voting Algorithm for Distributed Mutual Exclusion

Slides Created By: Shrideep Pallickara
Maekawa's solution to distributed mutual exclusion

- In order for a process to enter a critical section it is not necessary for all peers to grant access
- Obtain permission from subsets of peers
- Subsets used by any two peers must overlap
- Candidate process must collect sufficient votes to enter critical section

How mutual exclusion is achieved

- Processes at the intersection of two sets of voters ensure this
- Cast votes for only one candidate

Voting sets

- There is a voting set \( V_i \) associated with each process \( p_i \) (i=1,2,...,N)
- \( V_i \subseteq \{ p_1, p_2, ..., p_N \} \)

The optimal solution to the Maekawa's algorithm

\[
K \sim \sqrt{N} \\
M = K
\]

Each process is in as many of the voting sets as there are elements in one of the sets

Maekawa’s voting sets

Example

<table>
<thead>
<tr>
<th>K = 4</th>
<th>N = 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_{1} = {1, 2, 3, 4}</td>
<td>R_{2} = {2, 5, 6, 9}</td>
</tr>
<tr>
<td>R_{3} = {1, 4, 5}</td>
<td>R_{4} = {2, 6, 10, 11}</td>
</tr>
<tr>
<td>R_{5} = {1, 6, 7}</td>
<td>R_{6} = {2, 7, 10, 12}</td>
</tr>
<tr>
<td>R_{7} = {1, 8, 9}</td>
<td>R_{8} = {3, 6, 13}</td>
</tr>
<tr>
<td>R_{8} = {2, 5, 7}</td>
<td>R_{9} = {3, 7, 12}</td>
</tr>
<tr>
<td>R_{9} = {2, 8, 7, 13}</td>
<td>R_{10} = {4, 6, 13}</td>
</tr>
<tr>
<td>R_{10} = {2, 9, 5, 14}</td>
<td>R_{11} = {5, 7, 12}</td>
</tr>
<tr>
<td>R_{11} = {2, 10, 6, 15}</td>
<td>R_{12} = {6, 7, 14, 15}</td>
</tr>
<tr>
<td>R_{12} = {2, 11, 7, 16}</td>
<td>R_{13} = {7, 8, 12, 17}</td>
</tr>
<tr>
<td>R_{13} = {2, 12, 8, 18}</td>
<td>R_{14} = {8, 9, 13, 19}</td>
</tr>
<tr>
<td>R_{14} = {2, 13, 9, 20}</td>
<td>R_{15} = {9, 10, 14, 21}</td>
</tr>
<tr>
<td>R_{15} = {2, 14, 10, 22}</td>
<td>R_{16} = {10, 11, 15, 23}</td>
</tr>
</tbody>
</table>

Entering the critical section

- To obtain entry into the critical section, each $p_i$ sends request message to all $K$ members of $V_i$, including itself.
- $p_i$ cannot enter critical section till it has received all $K$ reply messages.

The reply message

- When a process $p_i$ in $V_i$ receives $p_i$’s request message, it sends a reply message immediately unless...
  - Its state is HELD
  - It has replied (voted) since it last received a release message.

The release message

- To leave the critical section, $p_i$ sends release message to all $K$ members of $V_i$ (incl. itself).
- When a process receives a release message:
  - Remove the head of its queue of outstanding requests and send a reply (vote) in response to it.

Satisfying the safety property

- If it were possible for $p_i$ and $p_j$ to enter the critical section at the same time, then...
  - Processes in $V_i \cap V_j \neq \emptyset$ would have voted for both $p_i$ and $p_j$.
  - But a process can make at most one vote between successive receipts of a release message.
  - So it is impossible for $p_i$ and $p_j$ to both enter the critical section.

But the basic algorithm is deadlock prone

- Consider three processes $p_1$, $p_2$, and $p_3$ with $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, and $V_3 = \{p_3, p_1\}$.
- If 3 processes concurrently request entry to the critical section, it is possible for:
  - $p_1$ to reply to itself and hold-off $p_2$.
  - $p_2$ to reply to itself and hold-off $p_1$.
  - $p_3$ to reply to itself and hold-off $p_1$.
  - Each process receives one of two replies; none can proceed.

Resolving the deadlock issue

- Processes queue requests in the happened-before order.
  - This also allows ME3 to be satisfied besides ME2.
Analyzing the performance of the algorithm

- Bandwidth utilization
  - \( \frac{2}{N} \) messages per entry into the critical section
  - \( \frac{2}{N} \) messages per exit
  - Total of \( \frac{2}{N} \) is superior to \( 2(N-1) \) required by the previous algorithm (Ricart and Agarwala)
    - if \( N \geq 3 \)
- Synchronization delay
  - Round-trip time

Election Algorithms

Election algorithms
- Algorithm for choosing a unique process to play a particular role
- When an elected process wants to retire, another election is needed

Calling an election
- When a process calls an election it initiates a particular run of the election algorithm
- A given process does not call more than one election at a time
  - With \( N \) processes there could be \( N \) concurrent elections
- At any point a process \( p_i \) is either:
  - A participant: Engaged in the election algorithm
  - Non-participant: Not engaged in the election algorithm

The choice of the elected process must be unique
- Even in cases where several processes call the election simultaneously
- E.g., 2 processes see a coordinator has failed and they both call elections

The elected process is the one with the largest identifier
- The identifier is any value with the provision that the identifiers are unique and totally ordered
- E.g., electing process with the lowest computational load
  - Use \( \langle i/\text{load}, i \rangle \) as the identifier
  - Process \( i \) is used to order identifiers with same load
Managing the identity of the elected process

- Each process $p_i$ (i=1, 2, ..., N) has a variable $elected_i$, contains identifier of the elected process.
- When a process first becomes a participant in an election, set this variable to $\perp$ indicating that it is undefined.

Requirements for the election algorithm

- **E1** (safety):
  - Participant process has $elected_i = \perp$ or $elected_i = P$
  - $P$ is a non-crashed process at the end of run with the largest identifier.

- **E2** (liveness):
  - All processes $p_i$ participate and eventually either set $elected_i \neq \perp$ or crash.

Measuring performance of election algorithms

- Network bandwidth utilization: How many messages are sent?
- Turnaround time for the algorithm: Number of message transmissions between the initiation and termination of a run.

The contents of this slide set are based on the following references