Frequently asked questions from the previous class survey

- Yes. But what really is a second?
  - 1 second == time for a cesium 133 atom to make 9,192,631,770 transitions
- Even though we are assuming processes do not fail, how would you cope with tokens that are lost in transit?

Topics covered in this lecture

- Distributed Mutual Exclusion
  - Multicast & logical clocks (Aparwala & Ricart)
  - Maekawa's voting based algorithm
  - Election algorithms

Requirements for distributed mutual exclusion

- ME1: At most one process may execute in the critical section at a time
- Safety
- ME2: Requests to enter and exit the critical section eventually succeed
- Liveness: Freedom from deadlocks and starvation
- ME3: If one request happened-before another, then entry to the CS is granted in that order

Evaluation of the algorithms

- Bandwidth consumed
  - Proportional to number of messages sent in each entry and exit operation
- Client delay incurred by the process for each entry or exit operation
- Effect on throughput of the system
  - Synchronization delay between one process exiting critical section and next process entering it
  - Throughput is greater when synchronization delay is shorter

Mutual Exclusion using Multicast & Logical Clocks (Ricart & Agrawala's Algorithm)
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

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. 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
2. When a process requests entry?
   - Defers all processing requests from other processes until its own request has been sent

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

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: $p_j$ not interested in entering critical section
- $p_j$ and $p_k$ request entry concurrently
- Timestamp of $p_j$'s request: 41
- Timestamp of $p_k$'s request: 34

$p_j$ enters the critical section
Achieving the properties ME1, ME2 and ME3

- If two processes \( p_i \) and \( p_j \) enter critical section at the same time?
  - Both these processes would have replied to each other; but the pairs \( \langle T_i, p_i \rangle \) 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

Some observations

- One of the problems with the central server algorithm was that it was a single point of failure
- Here, the single point of failure has been replaced by \( N \) points of failure
- If any process crashes, it will fail to respond to requests
  - This silence is interpreted (incorrectly) as a denial of permission
  - Blocks all subsequent processes from entering the critical section
- Solution: To have timeout mechanisms in place

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 $p_i$, $i=1,2,\ldots,N$.
  $$V_i \subseteq \{p_1, p_2, \ldots, p_N\}$$

The optimal solution to 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>$K$</th>
<th>$N$</th>
<th>$R_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>$R_{12}$</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>$R_{13}$</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?
  - Removes the head of its queue of outstanding requests and sends 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.

Resolving the deadlock issue
- Processes queue requests in the happened-before order.
  - This also allows ME3 to be satisfied besides ME2.

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:
  - \( p_i \) to reply to itself and hold-off \( p_i \)
  - \( p_j \) to reply to itself and hold-off \( p_i \)
  - \( p_j \) to reply to itself and hold-off \( p_i \)
  - Each process receives one of two replies; none can proceed.
Analyzing the performance of the algorithm

- Bandwidth utilization
  - $2\sqrt{N}$ messages per entry into the critical section
  - $\sqrt{N}$ messages per exit
  - Total of $3\sqrt{N}$ is superior to $2(N-1)$ required by the previous algorithm (Ricart and Agarwala)
- 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 $<1/load, i>$ as the identifier
  - Process $i$ is used to order identifiers with some load
Managing the identity of the elected process

1. Each process $p_i (i=1, 2, \ldots, N)$ has a variable $elected_i$.
   - Contains identifier of the elected process.
2. 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

1. **E1 (safety)**
   - Participant processes have $elected_i = \perp$ or $elected_i = P$
   - $P$ is a non-crashed process at the end of the run with the largest identifier.
2. **E2 (liveness)**
   - All processes $p_i$ participate and eventually either set $elected_i \neq \perp$ or crash.

Measuring performance of election algorithms

1. **Network bandwidth utilization**
   - How many messages are sent.
2. **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: