CS 455: INTRODUCTION TO DISTRIBUTED SYSTEMS
[DISTRIBUTED MUTUAL EXCLUSION]

Permissions and Critical Sections
A process holding off on replies is a cue
For the critical section, there’s a queue
You either collect permissions from all
Or from curated subsets that are small
Because messages these subsets curtail
An added perk is that the system will scale

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

▪ What do the tokens actually contain?
  ▪ Typically, digitally signed metadata
Topics covered in this lecture

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

LAMPORT’S CLOCK WRAP-UP

April 16, 2019
An example of Lamport’s algorithm:

Each clock runs at a constant (but different rate)

April 16, 2019
CS455: Introduction to Distributed Systems [Spring 2019]
Dept. Of Computer Science, Colorado State University
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
   $C_j \leftarrow \max\{C_i, ts(m)\}$
   do step (1) and deliver message

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

Add $100 to account
San Francisco

Update with 1% interest
New York

Add $100 ... Total: $1100
Give 1% interest on total: $11
Balance: $1111

Give 1% interest ... Total: $1010
Add $100
Balance: $1110
There is a difference when the orders are reversed

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

- Situations like this 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
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 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 & Agarwala'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
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:**
- $p_3$ not interested in entering critical section
- $p_1$ and $p_2$ request entry concurrently
- Timestamp of $p_1$'s request: 41
- Timestamp of $p_2$'s request: 34

$p_2$ enters the critical section

Achieving the properties ME1, ME2 and ME3

- If two processes $p_i$ and $p_j$ ($i \neq j$) enter critical section at the same time?
  - Both these processes would have replied to each other; but the pairs $<T_i, p_i>$ 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 [1/2]

- 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
Some observations

- Another problem with the central server algorithm was that making it handle all requests can lead to a bottleneck.
- In this setup all processes are involved in all decisions.
- Improvements?
  - Getting permission from everyone is an overkill.
  - All we need is to prevent two processes from entering the CS at the same time.

---

**MAEKAWA’S VOTING ALGORITHM FOR DISTRIBUTED MUTUAL EXCLUSION**
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 sets $V_i$ are chosen such that, for all $i, j = 1, 2, \ldots, N$

$$p_i \in V_i$$

$$V_i \cap V_j \neq \emptyset$$

$$|V_i| = K$$  \hspace{1em} To be fair, each process has a voting set of the same size

Each process $p_j$ is contained in $M$ of the voting sets $V_i$
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

Calculation of voting sets

- Is not trivial
- As an approximation
  - Place processes in a \( \sqrt{N} \) by \( \sqrt{N} \) matrix
  - Voting set \( V_i \) is the union of the row and column containing \( p_i \)
  - Voting set size is then \( \sim 2\sqrt{N} \)
Maekawa’s voting sets

Example

<table>
<thead>
<tr>
<th>$R_i$</th>
<th>{1, 2, 3, 4}</th>
<th>{1, 2, 3}</th>
<th>{1, 4, 5}</th>
<th>{1, 4, 5}</th>
<th>{1, 2, 3, 4, 5}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_2$</td>
<td>{1, 5, 6, 7}</td>
<td>{1, 5, 6}</td>
<td>{2, 5, 8, 11}</td>
<td>{2, 5, 7}</td>
<td>{2, 5, 10, 12}</td>
</tr>
<tr>
<td>$R_3$</td>
<td>{1, 8, 9, 10}</td>
<td>{2, 6, 9, 12}</td>
<td>{2, 7, 10, 13}</td>
<td>{3, 4, 7}</td>
<td>{3, 6, 8, 13}</td>
</tr>
<tr>
<td>$R_4$</td>
<td>{1, 11, 12, 13}</td>
<td>{2, 6, 9, 12}</td>
<td>{2, 7, 10, 13}</td>
<td>{3, 6, 8, 13}</td>
<td>{3, 7, 9, 11}</td>
</tr>
<tr>
<td>$R_5$</td>
<td>{2, 7, 11, 15, 19}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 7, 11, 15, 19}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_6$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_7$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_8$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_9$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_{10}$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_{11}$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 13}</td>
</tr>
<tr>
<td>$R_{12}$</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{2, 7, 10, 13}</td>
<td>{2, 8, 12, 16, 20}</td>
<td>{4, 5, 9, 13}</td>
<td>{4, 5, 9, 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 $p_j$ 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

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 \( p_3 \)
  - \( p_3 \) 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
  - $2\sqrt{N}$ 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)
    - 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 $<\text{load}, i>$ 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 $\bot$ indicating that it is undefined
Requirements for the election algorithm

- **E1 (safety)**
  - Participant process has $elected_i = \bot$ 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 $elected_i \neq \bot$ 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
