Distributed Mutual Exclusion

Olivier Dalle (*)

olivier.dalle@sophia.inria.fr

(*) Large parts of this lecture borrowed from Sukumar Ghosh's book.
Distributed Mutual Exclusion

Mostly from Sukumar Ghosh's book and handsout:
1 – Introduction
2 – Solutions Using Message Passing
3 – Token Passing Algorithms
4 – The Group Mutual Exclusion Problem

Also in Ghosh's book (not covered by this lecture):
- Solution on the shared memory model
  - Peterson algorithm
- Mutual exclusion using special instruction
  - Solution using Test-and-Set
  - Solution using DEC LL and SC instruction
1 – **Introduction**
2 – Solutions Using Message Passing
3 – Token Passing Algorithms
4 – The Group Mutual Exclusion Problem
Why Do We Need Distributed Mutual Exclusion (DME)?

Atomicity exists only up to a certain level:

- Atomic instructions
- Defines the granularity of the computation
  - Types of possible interleaving
    - Assembly Language Instruction?
    - Remote Procedure Call?
1- Introduction

Why Do We Need Distributed Mutual Exclusion (DME) ?

Some applications are:

- Resource sharing
- Avoiding concurrent update on shared data
- Controlling the grain of atomicity
- Medium Access Control in Ethernet
- Collision avoidance in wireless broadcasts
Why Do We Need Distributed Mutual Exclusion (DME)?

Example: Bank Account Operations

shared n : integer

**Process P**
Account receives amount nP
Computation: \( n = n + nP \):
  P1. Load Reg_P, n
  P2. Add Reg_P, nP
  P3. Store Reg_P, n

**Process Q**
Account receives amount nQ
Computation: \( n = n + nQ \):
  Q1. Load Reg_Q, n
  Q2. Add Reg_Q, nQ
  Q3. Store Reg_Q, n
Possible Interleaves of Executions of P and Q:

2 give the expected result $n = n + nP + nQ$
- P1, P2, P3, Q1, Q2, Q3
- Q1, Q2, Q3, P1, P2, P3

5 give erroneous result $n = n + nQ$
- P1, Q1, P2, Q2, P3, Q3
- P1, P2, Q1, Q2, P3, Q3
- P1, Q1, Q2, P2, P3, Q3
- Q1, P1, Q2, P2, P3, Q3
- Q1, Q2, P1, P2, P3, Q3

5 give erroneous result $n = n + nP$
- Q1, P1, Q2, P2, Q3, P3
- Q1, Q2, P1, P2, Q3, P3
- Q1, P1, P2, Q2, Q3, P3
- P1, Q1, P2, Q2, Q3, P3
- P1, P2, Q1, Q2, Q3, P3
1- Introduction

Solutions to the Mutual Exclusion Problem

\begin{center}
\begin{tikzpicture}
  \node[minimum width=3cm, minimum height=2cm] (p0) at (0,0) {p0};
  \node[minimum width=3cm, minimum height=2cm] (p1) at (0,-3) {p1};
  \node[minimum width=3cm, minimum height=2cm] (p2) at (0,-6) {p2};
  \node[minimum width=3cm, minimum height=2cm] (p3) at (0,-9) {p3};
  \node[draw, minimum width=3cm, minimum height=2cm] (CS0) at (0,0) {CS};
  \node[draw, minimum width=3cm, minimum height=2cm] (CS1) at (0,-3) {CS};
  \node[draw, minimum width=3cm, minimum height=2cm] (CS2) at (0,-6) {CS};
  \node[draw, minimum width=3cm, minimum height=2cm] (CS3) at (0,-9) {CS};

  \draw[thick, -latex] (p0.center) -- (CS0.center);
  \draw[thick, -latex] (p1.center) -- (CS1.center);
  \draw[thick, -latex] (p2.center) -- (CS2.center);
  \draw[thick, -latex] (p3.center) -- (CS3.center);
\end{tikzpicture}
\end{center}
2 classes of solutions:

- Ad hoc solutions
- Solutions based on non-preemptible resource allocation

Both classes require a special code around the critical section

<table>
<thead>
<tr>
<th>Ad-hoc case</th>
<th>Non-preempt. resource case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enter protocol</strong></td>
<td><strong>Request resource</strong></td>
</tr>
<tr>
<td>&lt;critical section&gt;</td>
<td>&lt;critical section&gt;</td>
</tr>
<tr>
<td><strong>Exit protocol</strong></td>
<td><strong>Release resource</strong></td>
</tr>
</tbody>
</table>
1 – Introduction

2 – Solutions Using Message Passing

3 – Token Passing Algorithms

4 – The Group Mutual Exclusion Problem
Problem formulation

Assumptions

- $n$ processes ($n > 1$), numbered 0 ... $n-1$, noted $P_i$
- form a distributed system
- topology: completely connected graph ($K_n$)
- each $P_i$ periodically wants:
  1. enter the Critical Section (CS)
  2. execute the CS code
  3. eventually exits the CS code

Devise a protocol that satisfies:
- ME1 : Mutual Exclusion
- ME2 : Freedom from deadlock
- ME3 : Progress
Zoom on Conditions...

- **ME1 : Mutual Exclusion**
  - At most one process can remain in CS at any time
  - Safety property

- **ME2 : Freedom from deadlock**
  - At least one process is eligible to enter CS
  - Safety property

- **ME3 : Progress**
  - Every process trying to enter must eventually succeed
  - Liveness property
  - Violation called *livelock or starvation*

- **A measure of fairness: bounded waiting**
  - Specifies an upper bound on the number of times a process waits for its turn to enter SC
Centralized Solutions?

- Use a coordinator process
  - External process
  - One of the Pi-s

- Problems:
  - Single point of failure
  - Unable to achieve FIFO fairness

Example:

How to anticipate this late arrival?
2- Solutions Using Message Passing

**Lamport's Solution**

**Assumptions:**
- Each communication channel is FIFO
- Each process maintains a request queue Q

**Algorithm described by 5 rules**

LA1. To request entry, send a time-stamped message to **every** other process and **enqueue to local Q**

LA2. Upon reception place request in Q and send time-stamped ACK but **once out of CS**
   (possibly immediately if already out)

LA3. Enter CS when:
   1. request first in Q (chronological order)
   2. all ACK received from others

LA4. To exit CS, a process must:
   1. delete request from Q
   2. send time-stamped release message to others

LA5. When receiving a release msg, remove request from Q
Can you show that it satisfies all the properties (i.e. ME1, ME2, ME3) of a correct solution?

**Observation.** Processes taking a decision to enter CS must have identical views of their local queues, when all ACKs have been received.

**Proof of ME1.** At most one process can be in its CS at any time.

Suppose not, and both j,k enter their CS. This implies
- j in CS ⇒ Qj.ts.j < Qk.ts.k
- k in CS ⇒ Qk.ts.k < Qj.ts.j

Impossible.

![Diagram showing the order of processes and their queues](attachment:image.png)
Proof of ME2. (*No deadlock*)

The waiting chain is acyclic.

i waits for j

⇒ i is behind j in all queues

(or j is in its CS)

⇒ j does not wait for i

Proof of ME3. (*progress*)

New requests join the end of the
queues, so new requests do not pass
the old ones
Proof of FIFO fairness.

\[ \text{timestamp (j)} < \text{timestamp (k)} \]

\[ \Rightarrow \ j \text{ enters its CS before } k \text{ does so} \]

Suppose not. So, \( k \) enters its CS before \( j \). So \( k \) did not receive \( j \)'s request. But \( k \) received the ack from \( j \) for its own req.

This is impossible if the channels are FIFO.

Message complexity = \( 3(N-1) \) (per trip to CS)

\( (N-1 \text{ requests} + N-1 \text{ ack} + N-1 \text{ release}) \)
What is new?

1. Broadcast a timestamped request to all.
2. Upon receiving a request, send ack if
   - You do not want to enter your CS, or
   - You are trying to enter your CS, but your timestamp is higher than that of the sender.
   (If you are already in CS, then buffer the request)
3. Enter CS, when you receive ack from all.
4. Upon exit from CS, send ack to each pending request before making a new request.
   (No release message is necessary)
**ME1.** Prove that at most one process can be in CS.

**ME2.** Prove that deadlock is not possible.

**ME3.** Prove that FIFO fairness holds even if channels are not FIFO

Message complexity = 2(N-1)
(N-1 requests + N-1 acks - no release message)
Timestamps grow in an unbounded manner. This makes real implementation impossible. Can we somehow bound timestamps?

Think about it.
First solution with a sublinear $O(\sqrt{N})$ message complexity.

“Close to” Ricart-Agrawala’s solution, but each process is required to obtain permission from only a subset of peers.
Maekawa’s Algorithm

• With each process $i$, associate a subset $S_i$. Divide the set of processes into subsets that satisfy the following two conditions:

$$i \in S_i$$
$$\forall i, j : 0 \leq i, j \leq n-1 :: S_i \cap S_j \neq \emptyset$$

• **Main idea.** Each process $i$ is required to receive permission from $S_i$ *only*. Correctness requires that multiple processes will never receive permission from all members of their respective subsets.
Example. Let there be seven processes 0, 1, 2, 3, 4, 5, 6

\[
\begin{align*}
S_0 &= \{0, 1, 2\} \\
S_1 &= \{1, 3, 5\} \\
S_2 &= \{2, 4, 5\} \\
S_3 &= \{0, 3, 4\} \\
S_4 &= \{1, 4, 6\} \\
S_5 &= \{0, 5, 6\} \\
S_6 &= \{2, 3, 6\}
\end{align*}
\]
Maekawa’s Algorithm (example cont'd)

Version 1 {Life of process I}

1. Send timestamped request to each process in $S_i$.
2. Request received $\rightarrow$ send ack to process with the lowest timestamp. Thereafter, "lock" (i.e. commit) yourself to that process, and keep others waiting.
3. Enter CS if you receive an ack from each member in $S_i$.
4. To exit CS, send release to every process in $S_i$.
5. Release received $\rightarrow$ unlock yourself. Then send ack to the next process with the lowest timestamp.

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$
2- Solutions Using Message Passing

Analysis of Maekawa's Algorithm (version 1)

**ME1.** At most one process can enter its critical section at any time.

Let $i$ and $j$ attempt to enter their Critical Sections

\[ S_i \cap S_j \neq \emptyset \]

there is a process $k \in S_i \cap S_j$

Process $k$ will **never** send ack to both.

So it will act as the arbitrator and establishes ME1

\[
S_0 = \{0, 1, 2\} \\
S_1 = \{1, 3, 5\} \\
S_2 = \{2, 4, 5\} \\
S_3 = \{0, 3, 4\} \\
S_4 = \{1, 4, 6\} \\
S_5 = \{0, 5, 6\} \\
S_6 = \{2, 3, 6\}
\]
Analysis of Maekawa’s Algorithm (version 1)

ME2. No deadlock. Unfortunately deadlock is possible! Assume 0, 1, 2 want to enter their critical sections.

From $S_0 = \{0, 1, 2\}$, 0, 2 send ack to 0, but 1 sends ack to 1;

From $S_1 = \{1, 3, 5\}$, 1, 3 send ack to 1, but 5 sends ack to 2;

From $S_2 = \{2, 4, 5\}$, 4, 5 send ack to 2, but 2 sends ack to 0;

Now, 0 waits for 1 (to send a release), 1 waits for 2 (to send a release), and 2 waits for 0 (to send a release). So deadlock is possible!

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$
Avoiding deadlock
If processes always receive messages in increasing order of timestamp, then deadlock “could be” avoided. But this is too strong an assumption.

Version 2 uses three additional messages:
- failed
- inquire
- relinquish

\[
S_0 = \{0, 1, 2\} \\
S_1 = \{1, 3, 5\} \\
S_2 = \{2, 4, 5\} \\
S_3 = \{0, 3, 4\} \\
S_4 = \{1, 4, 6\} \\
S_5 = \{0, 5, 6\} \\
S_6 = \{2, 3, 6\}
\]
Maekawa’s Algorithm (version 2)

New features in version 2

- Send **ack** and set **lock** as usual.
- If **lock is set** and a request with a larger timestamp arrives, send **failed** *(you have no chance)*. If the incoming request has a lower timestamp, then send **inquire** *(are you in CS?)* to the locked process.
- Receive **inquire** and at least one **failed** message → send **relinquish**. The recipient resets the lock.

\[
S_0 = \{0, 1, 2\} \\
S_1 = \{1, 3, 5\} \\
S_2 = \{2, 4, 5\} \\
S_3 = \{0, 3, 4\} \\
S_4 = \{1, 4, 6\} \\
S_5 = \{0, 5, 6\} \\
S_6 = \{2, 3, 6\}
\]
Let \( K = |S_i| \). Let each process be a member of \( D \) subsets. When \( N = 7 \), \( K = D = 3 \). When \( K = D \), \( N = K(K-1)+1 \). So \( K = O(\sqrt{N}) \) (from theory of finite projective planes).

- The message complexity of Version 1 is \( 3\sqrt{N} \). Maekawa’s analysis of Version 2 reveals a complexity of \( 7\sqrt{N} \)

- Sanders identified a bug in version 2 …
In Ricart and Agrawala's distributed mutual exclusion algorithm, show that:

a) Processes enter their critical sections in the order of their request timestamps

b) Correctness is guaranteed even if the channels are not FIFO

A Generalized version of the mutual exclusion problem in which up to L processes (L ≥ 1) are allowed to be in their critical sections simultaneously is known as the L-exclusion problem. Precisely, if fewer than L processes are in the CS at any time and one more process wants to enter it, it must be allowed to do so. Modify R.-A. algorithm to solve the L-exclusion problem.
1 – Introduction

2 – Solutions Using Message Passing

3 – Token Passing Algorithms

4 – The Group Mutual Exclusion Problem
3- Tokens passing algorithms

**Suzuki-Kasami Solution**

**Completely connected** network of processes

There is **one token** in the network. The holder of the token has the permission to enter CS.

Any other process trying to enter CS must acquire that token. Thus the token will move from one process to another based on demand.
3- Tokens passing algorithms

Suzuki-Kasami Algorithm

Process i broadcasts \((i, \text{num})\)

Each process maintains

- an array `req`: `req[j]` denotes the sequence number of the latest request from process \(j\)
  
  *(Some requests will be stale soon)*

Additionally, the holder of the token maintains

- an array `last`: `last[j]` denotes the sequence number of the latest visit to CS for process \(j\).
- a queue \(Q\) of waiting processes

\(\text{req}: \text{array}[0..n-1] \text{ of integer} \)

\(\text{last}: \text{array}[0..n-1] \text{ of integer} \)
When a process $i$ receives a request $(k, \text{num})$ from process $k$, it sets \text{req}[k]$ to $\max(\text{req}[k], \text{num})$.

**The holder of the token**
- Completes its CS
- Sets $\text{last}[i] := \text{its own num}$
- Updates $Q$ by retaining each process $k$ only if $1 + \text{last}[k] = \text{req}[k]$  
  (*This guarantees the freshness of the request*)
- Sends the token to the head of $Q$, along with the array $\text{last}$ and the tail of $Q$

In fact, $\text{token} \equiv (Q, \text{last})$
{Program of process j}
Initially, ∀i: req[i] = last[i] = 0

* Entry protocol *
req[j] := req[j] + 1
Send (j, req[j]) to all
Wait until token (Q, last) arrives
Critical Section

* Exit protocol *
last[j] := req[j]
∀k ≠ j: k ∉ Q \( \Rightarrow \) req[k] = last[k] + 1 \( \Rightarrow \) append k to Q;
if Q is not empty \( \Rightarrow \) send (tail-of-Q, last) to head-of-Q fi

* Upon receiving a request (k, num) *
req[k] := max(req[k], num)
initial state: process 0 has sent a request to all, and grabbed the token
3- Tokens passing algorithms

Example of Suzuki-Kasami Algorithm Execution

1 & 2 send requests to enter CS
Example of Suzuki-Kasami Algorithm Execution

0 prepares to exit CS
Example of Suzuki-Kasami Algorithm Execution

3- Tokens passing algorithms

0 passes token (Q and last) to 1
3- Tokens passing algorithms

Example of Suzuki-Kasami Algorithm Execution

req=[2,1,1,1,0]

req=[2,1,1,1,0]

req=[2,1,1,1,0]

req=[2,1,1,1,0]

req=[2,1,1,1,0] last=[1,0,0,0,0]

Q=(2,0,3)

0 and 3 send requests
Example of Suzuki-Kasami Algorithm Execution

3- Tokens passing algorithms

1 sends token to 2
3- Tokens passing algorithms

Raymond's Solution

- Improved version of token-based solution
  - Uses a tree-topology

- Idea:
  - At any time, one node holds the token
    - The holder is the root of the tree
  - Every edge is assigned a direction
    - Route requests towards the root
    - If edge from Pi to Pj, Pj called holder of Pi
  - When the token moves, some edges change direction
Raymond's Algorithm

Outline

Each node has a **holder** variable and a local **Q**. Only first request forwarded to holder.

R1. A node **enters CS when it has token**. Otherwise (no token), registers request in local Q

R2. A node \( P_j \) with non empty \( Q \) sends 1\(^{st} \) request to its holder, **unless already sent** and awaiting for token.

R3. When root receives request, **sends to** neighbor at the **head of its local Q** after exiting CS. And changes **holder** to that node.

R4. When receiving a token, node \( P_j \) does:

- forward to neighbor at head of its local Q
- delete request from Q
- set **holder** to that neighbor
- if there are pending requests in \( Q \), send another request to **holder**
Example of Raymond's Algorithm Execution

1, 4, 7 want to enter their CS
Example of Raymond's Algorithm Execution

Example of Raymond's Algorithm Execution

3- Tokens passing algorithms

3 sends the token to 6
These two directed edges will reverse their direction

6 forwards the token to 1

The message complexity is $O(\text{diameter})$ of the tree. Extensive empirical measurements show that the average diameter of randomly chosen trees of size $n$ is $O(\log n)$. Therefore, the authors claim that the average message complexity is $O(\log n)$
In Suzuki-Kasami algorithm, prove the liveness property that any process requesting a token eventually receives the token. Also compute an upper bound on the number of messages exchanged in the system before the token is received.

Repeat previous exercise with Raymond's algorithm.
1 – Introduction
2 – Solutions Using Message Passing
3 – Token Passing Algorithms
4 – The Group Mutual Exclusion Problem
Problem proposed and solved by Young in 1999

- N processes, each belongs to one of M forums
- Four conditions must hold:
  1. Mutual exclusion. At most one forum in session at a time.
  2. Freedom from deadlock. At any time, at least one process should be able to make a move.
  3. Bounded waiting. Every forum chosen by a process must be in session in bounded time.
  4. Concurrent entry. Once a forum is in session, concurrent entry in session is guaranteed for all willing processes.
Simplistic Centralized Solution

- Assume only 2 forums F and F'.
- Each process has a flag with values in \{F, F', \bot\}
- Coordinator reads flags of each process in ascending order from 0 to N-1
  - Guarantees that first active Pi always served
  - followed by others requesting same forum

- Satisfies all requirement except bounded waiting
  - Possible starvation for one forum if processes keep entering always the same
  - Solved by electing a leader
    - first to enter forum
    - no more process allowed to join when leader leaves
Each process cycles through 4 phases
- request, in-cs, in-forum, passive
- Each process has flag={state,op}
  - state=phase, and op={F,F',⊥}

First version (for Pi, forum F):
- turn: F or F'
  - while ∃ Pj s.t. flag[j]=(in-cs,F')
    - do
      - flag[i] = (request,F)
        - while (turn ≠ F' and not all-passive(F')) do nop done
      - flag[i] = (in-cs, F)
    - done
  - attend forum F
  - turn = F'
  - flag[i] = (passive, ⊥)
Fair with respect to forums
- turn variable
- note that a process has to wait for all other candidate to F’ to be out of in-cs

Not fair for processes
- If several processes request F, at least one will succeed
- A process sleeping in NOP may not notice a forum change from F’ to F and then F’ again

Young's solution:
- Introduce a leader for each session (as in centralized)
- Each Pi has a variable successor[i] in (F, F’, ⊥)
  - denote which is next forum
- Only one leader can capture successors
- A Pk with successor[k] = F enters in session F if leader of F in session