Distributed Algorithms

Mutual exclusion

Ludovic Henrio
CNRS - projet OASIS
ludovic.henrio@inria.fr

Distributed Mutual Exclusion

Mostly from Sukumar Ghosh's book and handsout:
1 – Introduction
2 – Solutions Using Message Passing
3 – Token Passing Algorithms

Also in Ghosh’s book (not covered by this lecture):
- Group based mutual exclusion
- Mutual exclusion using special instruction
- Solution using Test-and-Set
- Solution using DEC LL and SC instruction

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?
  - Weak memory model?
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

Example: Bank Account Operations

```
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, Q2, P1, 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
  - P1, P1, P2, Q2, Q3, P3
  - P1, Q1, P2, Q2, Q3, P3
  - P1, P2, Q1, Q2, Q3, P3

Solutions to the Mutual Exclusion Problem
1. Introduction

Principle of the Mutual Exclusion Problem (2)

Each process, before entering the CS acquires the authorization to do so.

<table>
<thead>
<tr>
<th>Acquire authorization</th>
<th>Acquire authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter CS</td>
<td>Enter CS</td>
</tr>
<tr>
<td>&lt;critical section&gt;</td>
<td>&lt;critical section&gt;</td>
</tr>
<tr>
<td>Exit CS</td>
<td>Exit CS</td>
</tr>
</tbody>
</table>

Critical section should eventually terminate.

2. Solutions using Message Passing

Correctness 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
  - Liveness property

- **ME3**: Fairness
  - Every process trying to enter must eventually succeed
  - Absence of 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 -> n-fairness (n is the MAXIMUM number of rounds)
- FIFO fairness when n=0
How to Measure Performance

- Number of msg’s per CR invocation.
- Fairness measure (previous slide).
- Synchronization Delay (SD).
- Response Time (RT).
- System Throughput (ST):
  \[ ST = \frac{1}{SD + E} \]
  where E is the average CR execution time.

Last words before getting to the algorithms ...

Fault tolerance: in the case of failure, the algorithm can reorganize itself so that it continues to function without any disruption. -> not much true for the algorithms presented here.

Two kinds of algorithms: logical clock based and token based.

Distributed Mutual Exclusion

1 – Introduction
2 – Solutions using Message Passing
3 – Token Passing Algorithms

Problem formulation

- Assumptions
  - n processes (n>1), numbered 0 ... n-1, noted Pi communicating by sending / receiving messages
  - topology: completely connected graph
  - each Pi periodically wants:
    1. enter the Critical Section (CS)
    2. execute the CS code
    3. eventually exit the CS code
- Devise a protocol that satisfies:
  ME1 : Mutual Exclusion
  ME2 : Freedom from deadlock
  ME3 : Progress (of each process) → Fairness
Distributed Algorithms – Mutual Exclusion

2- Solutions using Message Passing

Centralized solution

- Use a coordinator process
- External process
- One of the Pi-s
- Queue requests and authorize one by one

Problems:
- Major: Single point of failure, contention
- Minor: Unable to achieve FIFO fairness (except if CO)

Example:

What if a timestamp is given when sending the CS request?

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 (of sender)
- 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. AND 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

Run an example with 3 processes and different interleavings

2- Solutions Using Message Passing

Analysis of Lamport’s Solution

Can you show that it satisfies all the properties (i.e. ME1, ME2, ME3) of a correct solution?

Observation. When all ACKs have been received any request on the way has a greater ts.
- ⇒ “coherent” view of the queue

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 < Qk.ts
- k in CS ⇒ Qk.ts < Qj.ts
- Impossible.

Analysis of Lamport’s Solution (2)

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.

What is causal ordering?
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 \( \text{ack} \) from \( j \) for its own req. This is impossible if the channels are FIFO.

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

\( (N-1 \text{ requests} + N-1 \text{ acks} + N-1 \text{ release}) \)

Exercise:

There is no global time => we are never sure \( j \) asked really first!

However, as a lot of messages are exchanged, this gives the opportunity for different clocks to synchronise – considering the time processes asked for the CS, the system is it still n-fair?

For which \( n \)? why?

---

Ricart & Agrawala’s Solution

What is new?
1. Broadcast a timestamped request to all.
2. Upon receiving a request, send \( \text{ack} \) 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 \( \text{ack} \) from all.
4. Upon exit from CS, send \( \text{ack} \) to each pending request before making a new request. (No release message is necessary)

Run an example with 3 processes and different interleavings

Exercise:

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 (note: this is the same fairness as in Lamport’s solution)

Message complexity = 2(N-1)

\( (N-1 \text{ requests} + N-1 \text{ acks} - \text{no release message}) \)
Exercises

A Generalized version of the mutual exclusion problem in which up to \( L \) processes (\( L \geq 1 \)) are allowed to be in their critical sections simultaneously is known as the \textbf{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 \textit{R.-A.} algorithm to solve the \textit{L-exclusion} problem.

Distributed Mutual Exclusion

1 – Introduction
2 – Solutions Using Message Passing
3 – Token Passing Algorithms

Token Ring Approach

Processes are organized in a logical ring: \( p_i \) has a communication channel to \( p_{(i+1) \mod (n)} \).

Operations:
- Only the process holding the token can enter the CS.
- To enter the critical section, wait passively for the token. When in CS, hold on to the token.
- To exit the CS, the process sends the token onto its neighbor.
- If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.

The basic ring approach

\( \begin{itemize} 
\item Safety & liveness are guaranteed, but \textit{ordering} is not.
\item Bandwidth: 1 message per exit
\item \((N-1)\) -fairness
\item Delay between one process's exit from the CS and the next process's entry is between 1 and N-1 message transmissions.
\end{itemize} \)
Suzuki-Kasami Solution

A mix of the Lamport queue and the token approach

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.

Suzuki-Kasami Algorithm

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

Each process maintains

- An array \( \text{req} \): \( \text{req}[j] \) 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 \( \text{last} \): \( \text{last}[j] \) denotes the sequence number of the latest visit to CS for process \( j \).
- A queue \( Q \) of waiting processes

Sequence number of the request

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

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

Suzuki-Kasami Algorithm (2)

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 \( \text{last}[i] := \text{num} \)
- 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, token \( \equiv (Q, \text{last}) \)

Req: array\[0..n-1\] of integer

Last: Array \[0..n-1\] of integer

Suzuki-Kasami Algorithm (3)

\[(\text{Program of process } j, \text{Initially, } \forall i: \text{req}[i] = \text{last}[i] = 0)\]

* Entry protocol *

- \( \text{req}[i] := \text{req}[i] + 1 \)
- Send \( (i, \text{req}[i]) \) to all
- Wait until token \((Q, \text{last})\) arrives
- Critical Section

* Exit protocol *

- \( \text{last}[i] := \text{req}[i] \)
- \( \forall k \in Q: \text{req}[k] = \text{last}[i] + 1 \Rightarrow \text{append } k \text{ to } Q; \)
- If \( Q \) is not empty \Rightarrow send \((\text{tail-of-} Q, \text{last})\) to head-of- \( Q \)

* Upon receiving a request \((k, \text{num})\) *

- \( \text{req}[k] := \max(\text{req}[k], \text{num}) \)
Example of Suzuki-Kasami Algorithm Execution

3- Tokens passing algorithms

Initial state: process 0 has sent a request to all, and grabbed the token

1 & 2 send requests to enter CS

0 prepares to exit CS

0 passes token (Q and last) to 1
Example of Suzuki-Kasami Algorithm

Execution

0 and 3 send requests

1 sends token to 2

Summary and advantages

Token-based + queue:
- Satisfies ME1 to ME3
- Less messages: N by CS
- Question: is this algorithm fair? All messages received during the CS are enqueued at the same position, cannot we do better?
- Note: index can be bounded
- Note 2: A similar algorithm was published by Ricart and Agrawala at the same period

Homework

Back to message passing: Maekawa’s Solution

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.

In this course: a quick overview of the basic 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

- $S_0 = \{0, 1, 2\}$
- $S_1 = \{1, 3, 5\}$
- $S_2 = \{2, 4, 5\}$
- $S_3 = \{0, 3, 4\}$
- $S_4 = \{1, 4, 6\}$
- $S_5 = \{0, 5, 6\}$
- $S_6 = \{2, 3, 6\}$

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.

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

\[ \begin{align*}
S_0 &= \{0, 1, 2\} \\
S_1 &= \{1, 3, 5\} \\
S_2 &= \{2, 4, 5\} \\
S_3 &= \{0, 3, 4\} \\
S_4 &= \{1, 4, 6\} \\
S_5 &= \{0, 5, 6\} \\
S_6 &= \{2, 3, 6\}
\end{align*} \]

Problem: Potential of Deadlock

Without the loss of generality, assume that three sites \( S_i, S_j \) and \( S_k \) simultaneously invoke Mutual Exclusion, and suppose:

\[ R_i \cap R_j = \{S_u\} \]
\[ R_j \cap R_k = \{S_v\} \]
\[ R_k \cap R_i = \{S_w\} \]

Solution: extra msg's to detect deadlock, maximum number of msg's = 5(\sqrt{n} + 1). NOT PRESENTED HERE

Dekker's algorithm

\[
\begin{align*}
&\text{flag}[0] := \text{false} \\
&\text{flag}[1] := \text{false} \\
&\text{turn} := 0 \quad /\ or 1 \\
&\quad \text{flag}[0] := \text{true} \\
&\quad \text{while} \:\ \text{flag}[1] = \text{true} \{ \\
&\quad\quad \text{if} \: \text{turn} = 0 \{ \\
&\quad\quad\quad \text{flag}[0] := \text{false} \\
&\quad\quad\quad \text{while} \: \text{turn} = 0 \{ \\
&\quad\quad\quad\quad \text{flag}[1] := \text{true} \\
&\quad\quad\quad\} \\
&\quad\quad\}\text{flag}[0] := \text{true} \\
&\quad \quad \text{// critical section} \\
&\quad \quad \} \\
&\quad \text{turn} := 1 \\
&\quad \text{flag}[1] := \text{true} \\
&\quad \text{// remainder section} \\
&\text{flag}[1] := \text{false} \\
&\text{// remainder section}
\end{align*}
\]