Distributed Algorithms

Mutual exclusion

Ludovic Henrio
CNRS - projet SCALE
ludovic.henrio@cnrs.fr
1 – Introduction
2 – Solutions Using Message Passing
3 – Token Passing Algorithms
Atomicity exists only up to a certain level:

- Atomic instructions define the granularity of the computation
  - Types of possible interleaving
  - Assembly Language Instruction?
  - Remote Procedure Call?
  - Weak memory model?

Some applications are:

- **Resource sharing**
- Avoiding concurrent update on *shared data*
- Controlling the grain of atomicity
- Medium Access Control in Ethernet
1- Introduction

Why Do We Need Distributed Mutual Exclusion (DME)?

Example: Bank Account Operations

shared n : integer

Process P

Account receives amount \( n_P \)

Computation: \( n = n + n_P \):

P1. Load Reg_P, n
P2. Add Reg_P, nP
P3. Store Reg_P, n

Process Q

Account receives amount \( n_Q \)

Computation: \( n = n + n_Q \):

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 $n = n + n_P + n_Q$
  - P1, P2, P3, Q1, Q2, Q3
  - Q1, Q2, Q3, P1, P2, P3

- 5 give erroneous result $n = n + n_Q$
  - 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 $n = n + n_P$
  - 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
Each process, before entering the CS acquires the authorizatino to do so.

Critical section should eventually terminate

p1

p2

p3
Mutual exclusion in the shared memory model – a solution for 2 processes

Knuth’s protocol

[Knuth-66]

Process $P_j$

```
loop
  non critical section ;
  loop
    $A[j] := 1$ ;
    await $B == j$ or $A[k] == 0$ ;
    $A[j] := 2$ ;
    if $A[k] != 2$ then break ;
  end loop ;
  $B := j$ ;
  critical section ;
  $B := k$ ;
  $A[j] := 0$
end loop
```

$j \in \{ 0, 1 \}$
other process: $k = 1 - j$

Warm up exercise (not easy): check Knuth’s algo

entry section

exit section

three shared variables $A[0], A[1], B$
Beware weak memory models

Reorder load before store

Proc0
st A=1
if (load B==0) {
   ...critical section
}

Proc1
st B=1
if (load A==0) {
   ...critical section
}

How are memory references from different processors interleaved? If this is not well-specified, synchronization becomes difficult or even impossible. Generally, additional synchronisation barriers / volatile declarations / ... necessary.
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
1 – Introduction

2 – Solutions using Message Passing

3 – Token Passing Algorithms
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
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?
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
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 $\Rightarrow$ Qj.ts < Qk.ts
- k in CS $\Rightarrow$ Qk.ts < Qj.ts

Impossible.
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,

WHY? ALWAYS?

so new requests do not pass the old ones

What is causal ordering?
Proof of FIFO fairness.

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

\[ \Rightarrow \] j enters its CS before k 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 requests + N-1 ack + N-1 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)

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 requests + N-1 acks - no release message)

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

- Req(k)
- Ack(j)
- Req(j)
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 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
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.
Features:

- Safety & liveness are guaranteed, but ordering is not.
- Bandwidth: 1 message per exit
- \((N-1)\) -fairness
- Delay between one process’s exit from the CS and the next process’s entry is between 1 and N-1 message transmissions.
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.
3- Tokens passing algorithms

Suzuki-Kasami Algorithm

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

Each process maintains
- an array $\text{req}$: $\text{req}[j]$ denotes the sequence nb 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

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

$\text{last}$: array $[0..n-1]$ of integer
3- Tokens passing algorithms

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 $\text{last}[i] :=$ its own $\text{num}$
-- 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 last and the tail of $Q$

In fact, $\text{token} \equiv (Q, \text{last})$

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

Last: Array [0..n-1] of integer
{Program of process j}
Initially, \( \forall i: \text{req}[i] = \text{last}[i] = 0 \)

* Entry protocol *
  \( \text{req}[j] := \text{req}[j] + 1 \)
  Send \((j, \text{req}[j])\) to all
  Wait until token \((Q, \text{last})\) arrives

* Exit protocol *
  \( \text{last}[j] := \text{req}[j] \)
  \( \forall k \neq j: k \notin Q \land \text{req}[k] = \text{last}[k] + 1 \rightarrow \text{append } k \text{ to } Q; \)
  if \( Q \) is not empty \( \rightarrow \) send \((\text{tail-of-Q}, \text{last})\) to head-of-Q fi

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

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

req=[1,0,0,0,0]
last=[0,0,0,0,0]
3- Tokens passing algorithms

Example of Suzuki-Kasami Algorithm Execution

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

- req=[1,1,1,0,0]
- last=[1,0,0,0,0]
- Q=(1,2)

0 prepares to exit CS
3- Tokens passing algorithms

**Example of Suzuki-Kasami Algorithm Execution**

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

last=[1,0,0,0,0]

Q=(2)

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

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

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

1 sends token to 2
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 bound
- Note 2: A similar algorithm was published by Ricart and Agrawala at the same period

WHY? -> Homework

WHY?
Prove that Suzuki-Kasami algorithm verifies the three properties of mutual exclusion

Note: have a look at the similar proofs in the course
Objective of these courses:
• design distributed algorithms
• write a few classical ones
• analyse and reason upon an algorithm
  More or less formal approaches (diagrams vs formal reasoning)

A word on more systematic formal approaches
• Model checking
• Framework for reasoning on algorithms, e.g. TLA+