EE 382V: Parallel Algorithms **Fall 2017**

Lecture 6: August 19

Lecturer: Vijay Garg *Scribe: Kelsey Sandlin*

6.1 Introduction

This document presents the material from Chapter 8 that was covered in lecture 6. The topic is mutual exclusion in distributed systems, and Lamport's Algorithm and Ricart and Agrawala's algorithm are given as possible solutions.

6.2 Mutual Exclusion in Distribued Systems

There often exisits some functionality that will only consistently produce correct output if limited to one process at a time. We can use a chat program (assume it looks like a live document and there is no conflict resolution) as an example: Two users should not be able to type at the same time or conflicts will occur. Typing should be considered the "Critical Section".
Critical Section Properties

1. **Safety** – two processes should not be in the critical section at the same time
2. **Liveness/Progress** – every request to enter the critical section should eventually be granted
3. **Fairness** – requests are granted in the order they are made. Fairness definitions may vary.

It is trivial to implement safety without the liveness or fairness constraints – don't let any process into the critical section.

6.3 How do we solve the mutual exclusion problem?

6.3.1 Assumptions

- N total processes
- completely connected topology & FIFO
- no faulures (solution does not need to be fault tolerant)

6.3.2 Brainstorming

We can't use a single shared lock because there is no shared memory. We need a queue structure to hold the requests to the critical section. Let's designate one server to manage the critical section and queue of requests.

6.4 Centralized Solution

The dedicated server maintains a queue of requests by adding each request received to the end of the queue. We assume that when a process is done with the critical section, it always alerts the dedicated server and the dedicated server then allows the next process in the queue to enter.
This solution satisfies **safety** and **liveness**, but not necessarily **fairness**. This is because processes are added to the queue in the order they are *received*, which is not necessarily the order the requests are sent.

We can make this solution fair by sending vector clocks along with the request. The dedicated server would insert requests into the queue in order by their process time.

But this solution really isn't distributed.

6.5 Lamport's Algorithm

In order to extend the centralized solution to be distributed, all processes need to maintain a queue of requests.

$$\textbf{queue structure:}((p1LCV, 1), (p9LCV, 9), (p4LCV, 4))$$

Where LCV = Logical Clock Value (Lamport's Clock Value) and the second value in each tuple is pid
A process is considered to be first in the queue if it's LCV is less than all other processes LCVs in the queue. It is possible that two or more processes send their request at the same LCV. In this case, you use resolve by selecting the lower pid value.

How can we guarantee that all processes have identical queues?

Lamport's Algorithm Psuedocode
Pi can enter it's critical section if:

1. Pi's request is the smallest in the queue
2. Pi has received acks from all other processes

request:

1. Enter your request (LCV,pid) into your local queue
2. Send request message to all other processes including LCV and pid

receive request:

1. Add Pi's request to local queue
2. Send ack to Pi

release:

1. Remove request from the queue

2. Send "release" message to all other processes (include pid)

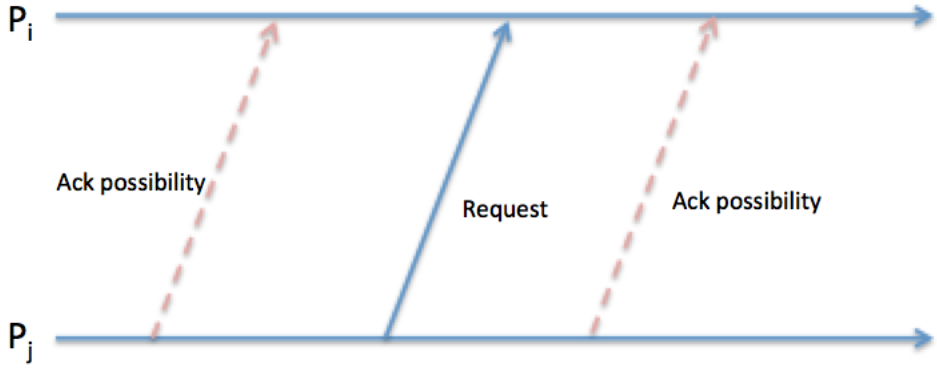
receive release:

1. Remove Pi's request from the queue

Does Lamport's algorithm satisfy the mutual exclusion properties?

- **Safety**

Let's assume Pi and Pj are in the CS at the same time and LVCj is less than LVCi. In order for Pj to have entered the CS, Pi must have seen an ack from Pj. Pj could have sent the ack to Pi either before or after sending it's request message.



Pj could not have sent the ack before the request because that would mean mean Pi's request must have already happened (which would have forced Pj's LVC to update), but that contradicts our assumption that LVCj is less than LVCi.
If the ack to Pi is sent after the request, then (by FIFO property) Pi must have seen Pj's request prior to Pj's ack and would have Pj's request in the queue before it's own request and therefore not entered the CS

- **Liveness and Fairness**

In Lamport's Algorithm, all requests are granted in the order of logical clock value. Assuming no message failures, liveness and fairness are guaranteed.

Analysis Requires 3(N-1) messages for N processes

- N-1 messages to send a request
- N-1 messages to send ack
- N-1 messages to send release

How can we reduce the number of messages sent? In the next algorithm we will combine "ack" and "release" messages into one message: "okay".