EE 382N-11: Distributed Systems **Fall 2017**

Lecture 4: September 12

Lecturer: Vijay Garg *Scribe: Colin Maxfield, Chin Wei Yeap*

4.1 Outline

- Mutual Exclusion
- Lamport's Algorithm
- Ricart and Agrawala's Algorithm
- Quorum-based Algorithm

4.2 Mutual Exclusion

4.2.1 Definitions

- **Critical Section (CS):** Parts of a program that share resources [1].
- **Mutual Exclusion:** Two processes should not be in critical section at any time (concurrently) [2].

4.2.2 Properties

1. **Safety:** System never gets into a "bad" state (two processes should not be in CS concurrently).
2. **Liveness:** Something "good" will eventually happen (every request is eventually granted).
3. **Fairness:** Different requests are granted in the order they are made [2].

4.3 Lamport's Algorithm

4.3.1 Resource Allocation

Mutual exclusion deals with resource allocation. To decide order and break ties/conflicts we can use logical clock timestamp and process identifiers.

- **Centralized Resource Allocation**
  One process becomes the coordinator/server and the rest are the clients. The coordinator stores a queue of all received requests to be able to grant the requests in order.
  1. Client → Server: Request resource

2. Server → Client: Grant resource
3. Client → Server: Release resource when done

Due to message delay the fairness property is not guaranteed. Requests are granted in the order they are received at the server, not necessarily in the order the requests were made.

Exercise: Modify the centralized algorithm such that it satisfies the fairness property from above. *Hint: use vector clock.*

- **Distributed Resource Allocation**
  Each process P_i stores a queue Q which contains all outstanding requests ordered by request timestamp.

  On request, do:

  1. Create request timestamp $ts :=$ logical clock()
  2. Insert request into Q_i
  3. Send ("request", (ts, i)) to all processes

  On receiving request from P_j, do:

  1. Insert the request into Q_i
  2. Send acknowledge *ack* to P_j

  P_i can enter CS if:

  1. Its request is at the top of Q_i
  2. Received *ack* from all processes

  To release CS, do:

  1. Remove my request from Q_i
  2. Send "release" message to all processes

  On receiving "release" from P_j, do:

  1. Remove corresponding request from Q_i

Assumption: By preserving FIFO (First In First Out) property, the queue stores all the outstanding requests ordered by request timestamps.

4.3.2 Proving Lamport's

Theorem: Lamport's Algorithm satisfies the safety property.

Proof: We have two requests, one from P_i and one from P_j. Since we are using logical clock, one is earlier than the other.

Assume: Request from P_j is earlier than request from P_i. Now if we assume that both P_i and P_j have entered the critical section then P_j must have sent *ack* following one of two cases as seen in Figure 4.1.

1. Case 1: P_j sends *ack* before sending its own request. If this case occurs it means that P_j would have received a request before its own and that P_j's request would be later in Q_j and would not have entered CS.
2. Case 2: P_j sends its request and then later sends *ack*. P_i would have already received request from P_j by the time it got *ack* and would not have entered CS since in Q_i, P_i is later than P_j.

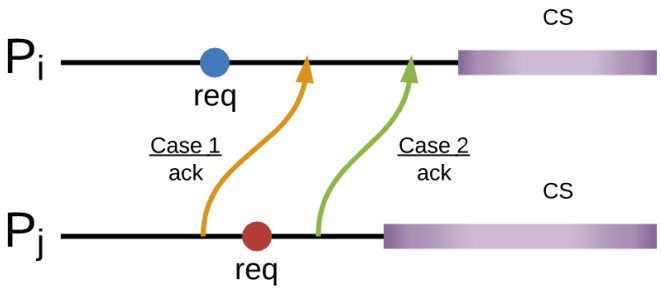

Figure 4.1: P_j can either send an *ack* before or after its request

4.3.3 Complexity

Direct Dependency Clock: $\forall s, t : s \neq t : cs(s) \wedge cs(t) \wedge s \| t \Rightarrow false$

Messenger/CS: $3(n-1)$ messages

- Request
- Ack
- Release

4.4 Ricart & Agrawala's Algorithm

4.4.1 Complexity

Ricart & Agrawala's algorithm decreases complexity to $2(n-1)$ with a few key advantages:

1. delays sending *ack* and combines with release into an "okay" message
2. does not assume FIFO message reception
3. keeps "pending" queue of requests which have not been responded to

4.4.2 Algorithm

PendingQ: queue of identifiers, initialized to empty queue.

On request, do:

1. Create request timestamp $ts :=$ logical clock()
2. Send ("request", (ts, i)) to all processes

On receiving request from P_j and both want to use CS, do:

1. if ($hists < myts$) reply "okay"
2. else insert his request into PendingQ

P_i can enter CS if:

1. receives "okay" from $(n-1)$ processes

To release CS, do:

1. send "okay" to all processes in PendingQ
2. clear PendingQ

On receiving "release" from P_j, do:

1. Remove corresponding request from Q_i

4.5 Quorum Based Algorithms

4.5.1 Maekawa's Algorithm

R_i: request set for P_i

Non-Empty Intersection Property: $\forall i, j : R_i \cap R_j \neq \{\}$

Use lattice of $\sqrt{N}$ rows by $\sqrt{N}$ columns, where N is the number of processes. Choose all nodes along row of P_i and column of P_i as seen in Figure 4.2. There will always be two intersection points between the request sets of any two processes.

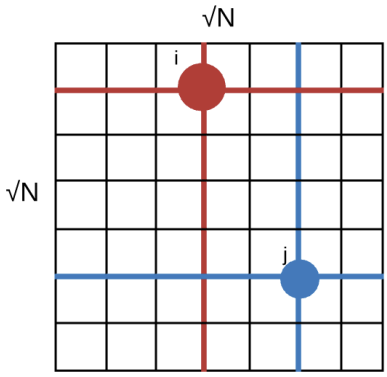

Figure 4.2: Intersection of P_i and P_j is found by both request sets made up of the column and row for each process

Complexity: $O(\sqrt{N})$

This decreases complexity by quite a bit but processes need to be able to take back votes if a request with earlier timestamp comes to prevent deadlocks.

4.5.2 Crumbling Wall

To generalize Maekawa's Algorithm we can look at ways of not taking a full row and a full column. Try generalizing in these ways:

- $\underline{R_i}$: my full row and only those in my column below me
- $\underline{R_i}$: my full row and one representative from each row below me
- $\underline{R_i}$: any full row and one representative from each row below the chosen row

A quorum in a crumbling wall is the union of one full row and a representative from every row below the full rows (original definition from textbook section 7.4.2). As long as 1 quorum (row/column) is alive, the crumbling wall can continue. This is seen in Figure 4.3.

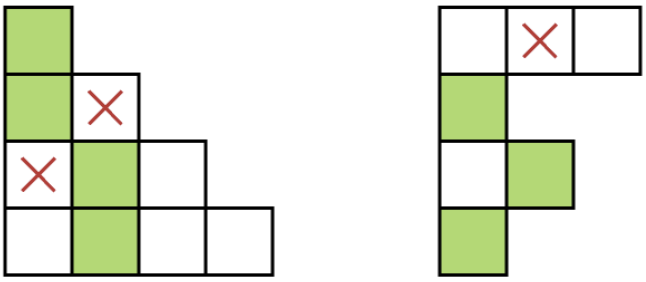

Figure 4.3: A couple different possible organizations for the crumbling wall generalization

References

[1] Wikipedia, "Critical Section", https://en.wikipedia.org/wiki/Critical_section

[2] V. J. Garg, "Chapter 6: Mutual Exclusion: Using Timestamps", and "Chapter 7: Mutual Exclusion: Tokens and Quorums", *Elements of Distributed Computing* **5** 1st ed. New York: John Wiley & Sons, Inc., 2002, pp. 65-91.