

Mutual exclusion in distributed systems

- All the solutions to the mutual exclusion problem studied assume presence of shared memory
 - Ex. Semaphores, monitors, etc. all rely on shared variables
- The mutual exclusion problem is complicated in distributed system by
 - lack of shared memory
 - lack of a common physical clock
 - unpredictable communication delays
- Several algorithms have been proposed to solve this problem with different performance trade-offs

Simple algorithm

- A trivial solution to the distributed mutual exclusion problem:
 - a single *control site* in charge of granting permissions to access the resource
- This solution has several drawbacks:
 - existence of a single point of failure
 - control site is a bottleneck
 - time to grant a new permission is $2T$ (T = average message delay)

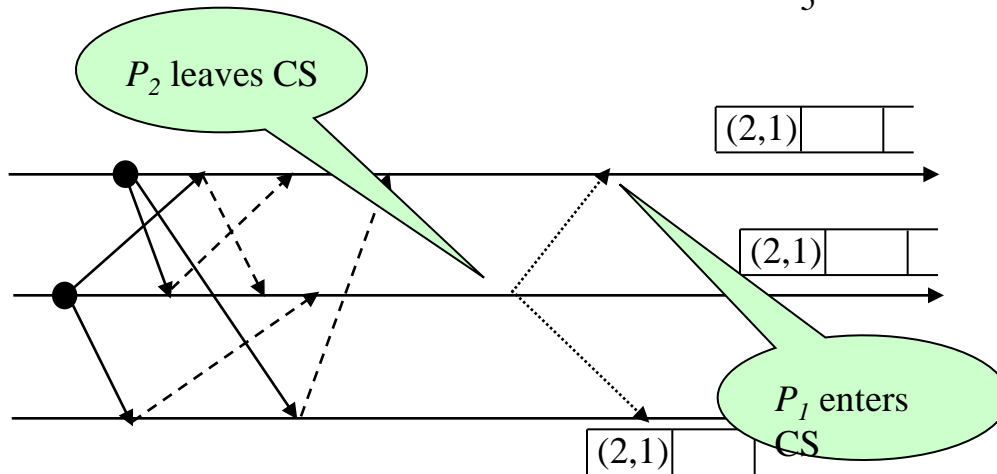
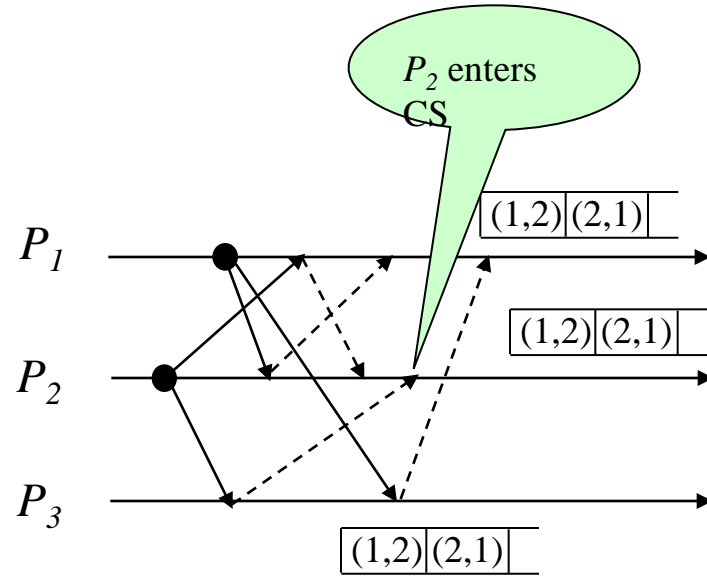
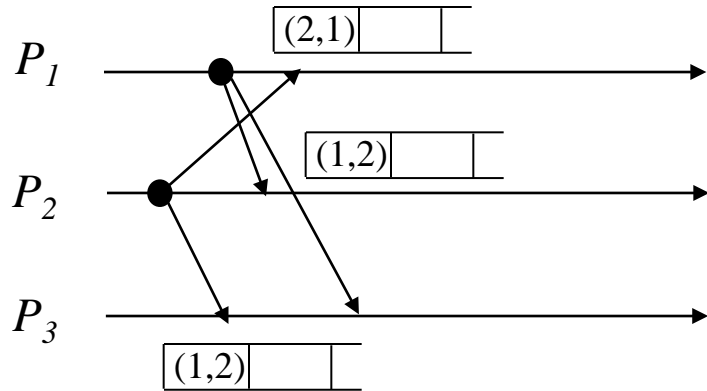
Lamport's Algorithm

- Assumption: message delivered in FIFO order
- Requesting the CS
 - P_i sends message **REQUEST**(t_i, i) to other processes, then enqueues the request in its own *request_queue_i*
 - when P_j receives a request from P_i , it returns a timestamped **REPLY** to P_i and places the request in *request_queue_j*
- A process P_i executes the CS only when:
 - P_i has received a message with timestamp larger than t_i from all other processes
 - its own request is the first of the *request_queue_i*

Lamport's Algorithm (2)

- Releasing the critical section:
 - when done, a process remove its request from the queue and sends a timestamped **RELEASE** message to all
 - upon receiving a **RELEASE** message from P_i , a process removes P_i 's request from the request queue

Lamport's Algorithm Example



Lamport's: proof of correctness

- Proof by contradiction:
 - assume P_i and P_j are executing the CS at the same time
 - (assume request timestamp of P_i is smaller than that of P_j)
 - this means both P_i and P_j have their request at the top of the queue
 - FIFO channels + first condition + P_j executing \Rightarrow request from P_i must be in *request_queue_j*
 - contradiction: P_i request in *request_queue_j* and not at the top of the queue, however we said $\text{timestamp}(P_i) < \text{timestamp}(P_j) \dots$
- Therefore it cannot be that P_i and P_j are executing the CS at the same time!

Ricart-Agrawala Algorithm

- Optimization of Lamport's algorithm:

Lamport's Algorithm

Requesting the CS

- P_i sends message **REQUEST**(t_i, i) + enqueues the request in *request_queue_i*
- when P_i receives a request from P_j , it enqueues it and returns a **REPLY** to P_i

P_i executes the CS only when:

- has received a msg with timestamp $> t_i$ from everybody
- its own request is the first in the *request_queue_i*

Releasing the CS:

- when done, a process remove its request from the queue + sends a timestamped **RELEASE** msg. to everybody else
- upon receiving a **RELEASE** message from P_j , a process removes P_i 's request from its request queue

Ricart-Agarwala Algorithm

Requesting the CS

- P_i sends message **REQUEST**(t_i, i)
- when P_i receives a request from P_j , it returns a **REPLY** to P_i if it is not requesting or executing the CS, or if it made a request but with a larger timestamp. Otherwise, the request is **deferred**.

P_i executes the CS only when:

- has received a **REPLY** from everybody

Releasing the CS:

- when done, a process sends a **REPLY** to all deferred requests

Ricart-Agrawala: proof of correctness

- Assumption: Lamport's clock is used
- Proof by contradiction:
 - assume P_i and P_j are executing the CS at the same time
 - assume request timestamp of P_i is smaller than that of P_j
 - this means P_i issued its own request first and then received P_j 's request, otherwise P_j request timestamp would be smaller
 - for P_i and P_j to execute the CS concurrently means P_i sent a REPLY to P_j before exiting the CS
 - Contradiction: a process is not allowed to send a REPLY if the timestamp of its request is smaller than the incoming one
- Therefore it cannot be that P_i and P_j are executing the CS at the same time!

Algorithm comparisons

- Ricart-Agrawala's can be seen as an optimization of Lamport's:
 - RELEASE messages are merged with REPLYs
- Basic differences:
 - Lamport's idea is to maintain (partially) coherent copies of a replicated data structure - the *request_queue*
 - Ricart-Agrawala does away with the data structure and just propagates state changes
 - messages needed for CS execution in the two schemes:
 - $3(N-1)$ vs. $2(N-1)$