Mansoura University
Faculty of Computers and Information Sciences
Department of Computer Science
First Semester- 2020-2021

[CS412P] Distributed Systems

Grade : Fourth grade

By : Zeinab Awad

LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM

Lamport's Distributed Mutual Exclusion Algorithm is a permission-based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems.

- In permission-based timestamp is used to order critical section requests and to resolve any conflict between requests.

-In Lamport's Algorithm critical section requests are executed in the increasing order of timestamps i.e a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.

LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM

Algorithm:

To enter Critical section:

When a site S_i wants to enter the critical section, it sends a request message **Request(ts_i, i)** to all other sites and places the request on **request_queue$_i$**. Here, Ts_i denotes the timestamp of Site S_i

When a site S_j receives the request message **REQUEST(ts_i, i)** from site S_i, it returns a timestamped REPLY message to site S_i and places the request of site S_i on **request queue$_j$** .

To execute the critical section:

A site S_i can enter the critical section if it has received the message with timestamp larger than **(ts_i, i)** from all other sites and its own request is at the top of **request queue$_i$** .

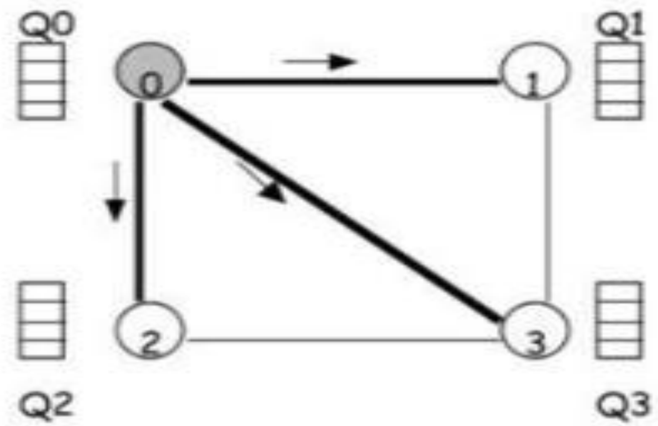
To release the critical section:

When a site S_i exits the critical section, it removes its own request from the top of its request queue and sends a timestamped **RELEASE** message to all other sites. When a site S_j receives the timestamped **RELEASE** message from site S_i, it removes the request of S_i from its request queue

LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM

Lamport's algorithm

1. Broadcast a timestamped ***request*** *to all.*
2. Request received → enqueue it in **local Q**. Not in CS → send ***ack***, else postpone sending ***ack*** until exit from CS.
3. Enter CS, when
   (i) You are at the "head" of your Q
   (ii) You have received ***ack*** from all
4. To exit from the CS,
   (i) Delete the ***request*** from your Q, and
   (ii) Broadcast a timestamped ***release***
5. When a process receives a ***release*** message, it removes the sender from its **Q**.



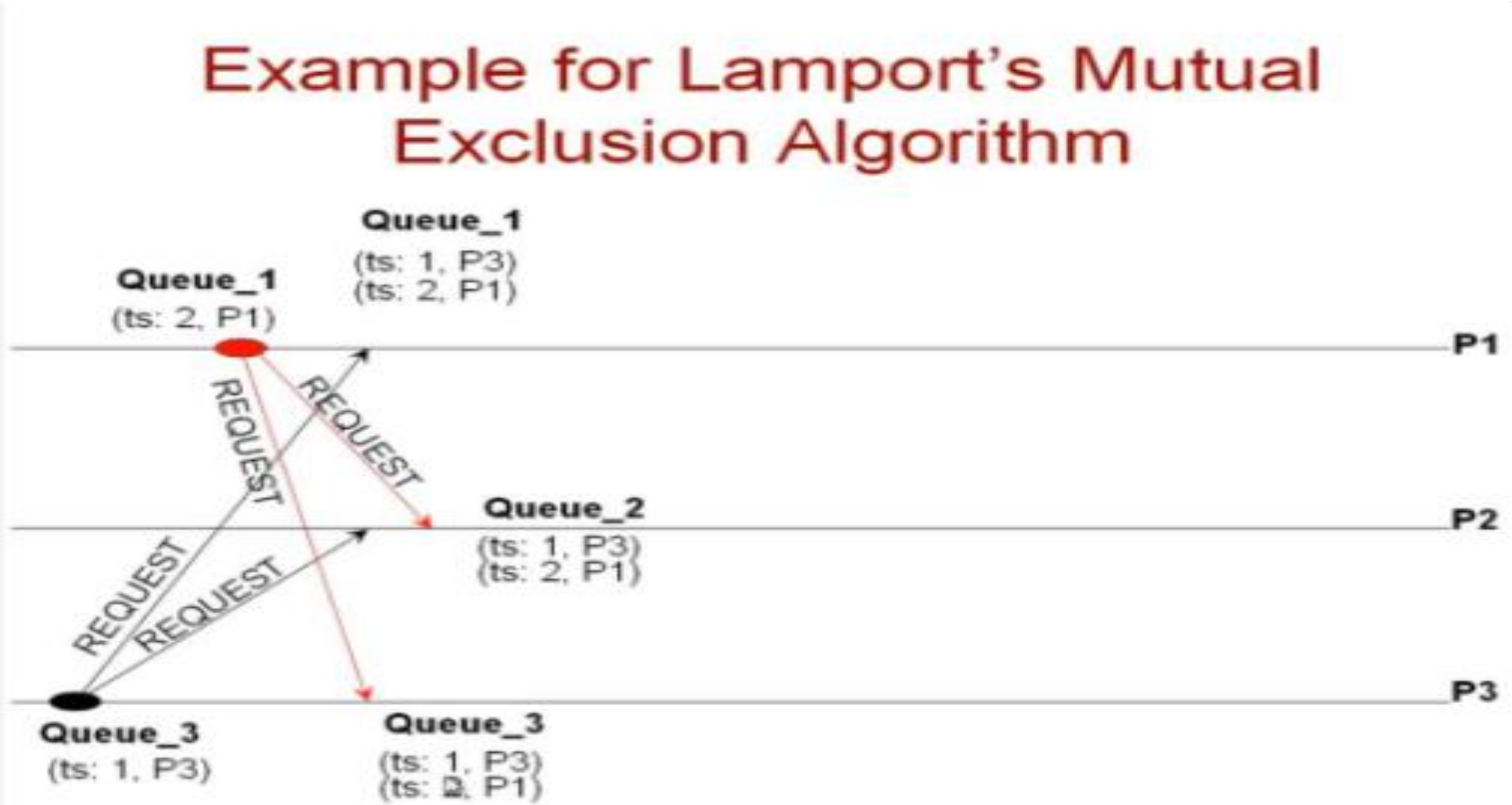
Completely connected topology

LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM

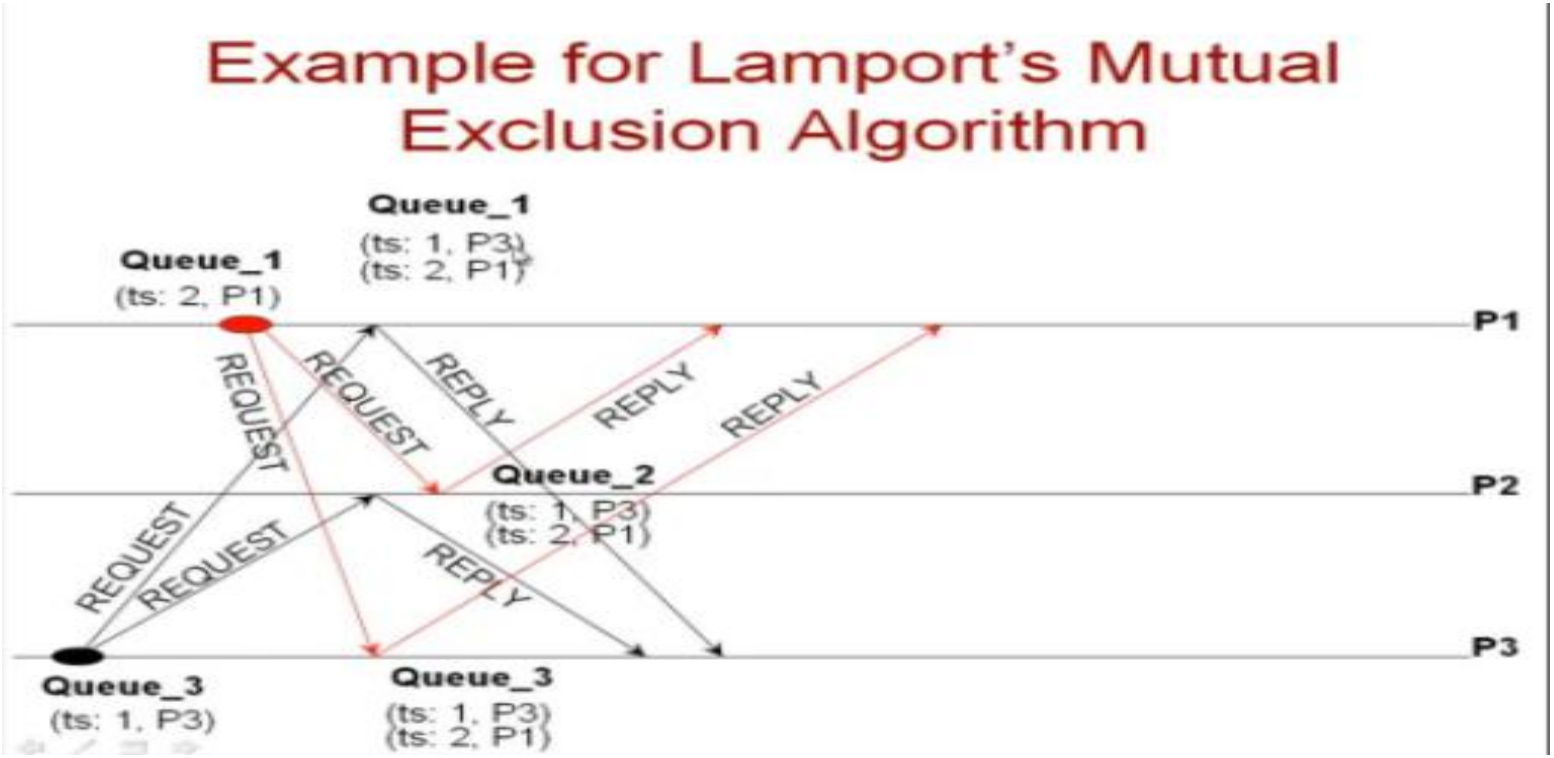
Lamport's Algorithm

- **Assumptions:**
  - FIFO and reliable communication channels
  - Priority Queues for storing CS Requests at each process (in the increasing value of time stamps; ties are broken by preferring lower process ids)
- **A process Pi wishing to execute a CS:**
  - Broadcasts a locally time stamped REQUEST message (tsi, i) to all the processes.
  - Inserts the REQUEST message in its proper position in queue_i
- **Receiving process Pj:**
  - Inserts the REQUEST message (tsi, i) in its proper position in queue_ j
  - Sends a REPLY to process Pi
- **A process Pi can execute the CS if:**
  - REPLY messages are received from all the other processes
  - The REQUEST of Pi is in the front of queue_i
- **After finishing the execution of the CS, a process Pi:**
  - Removes the Request message from its own queue
  - Broadcasts a RELEASE message (time stamped with that of the corresponding REQUEST message) to all the processes so that the latter can remove the REQUEST of Pi from their queue.

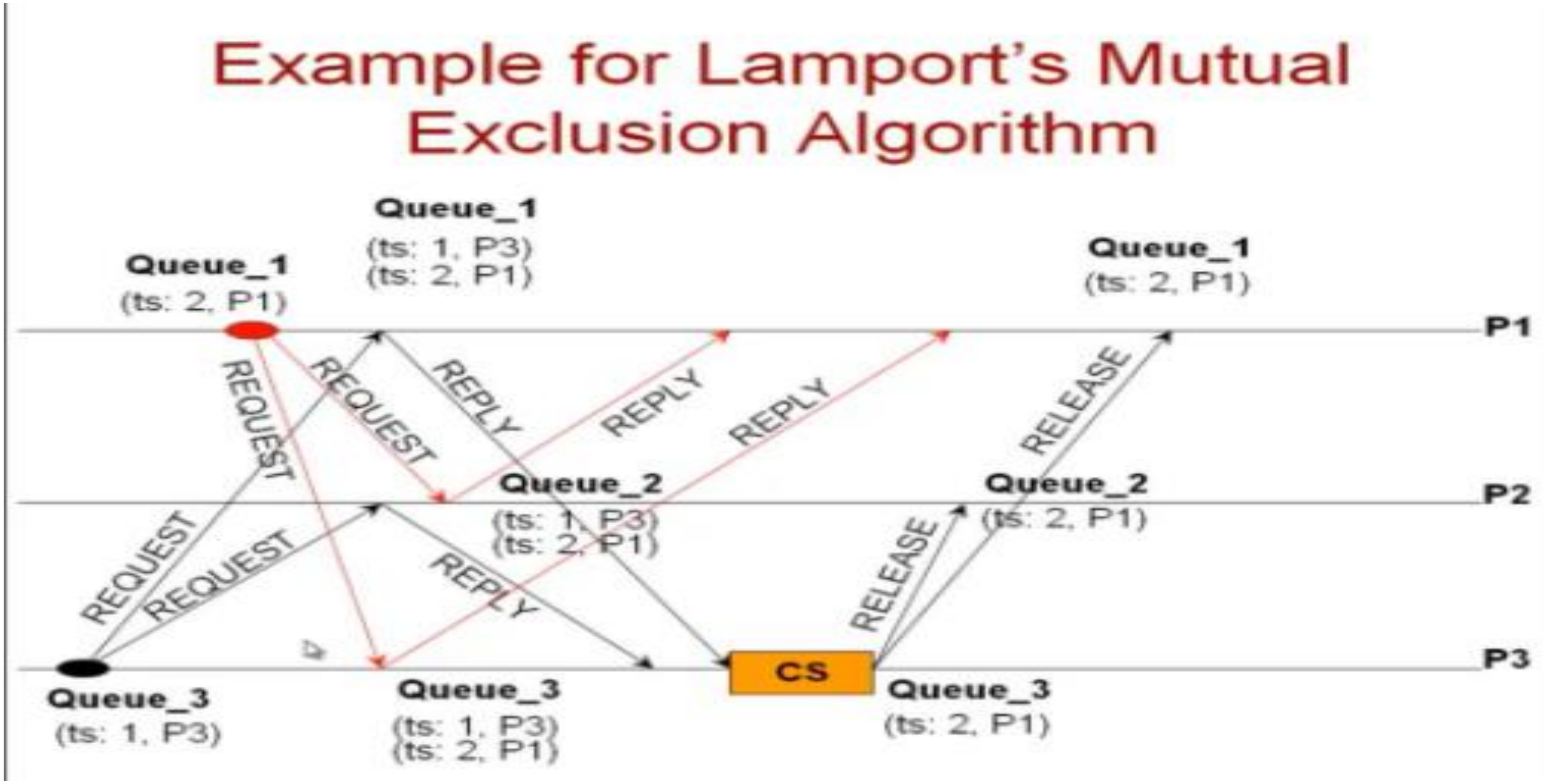
LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM



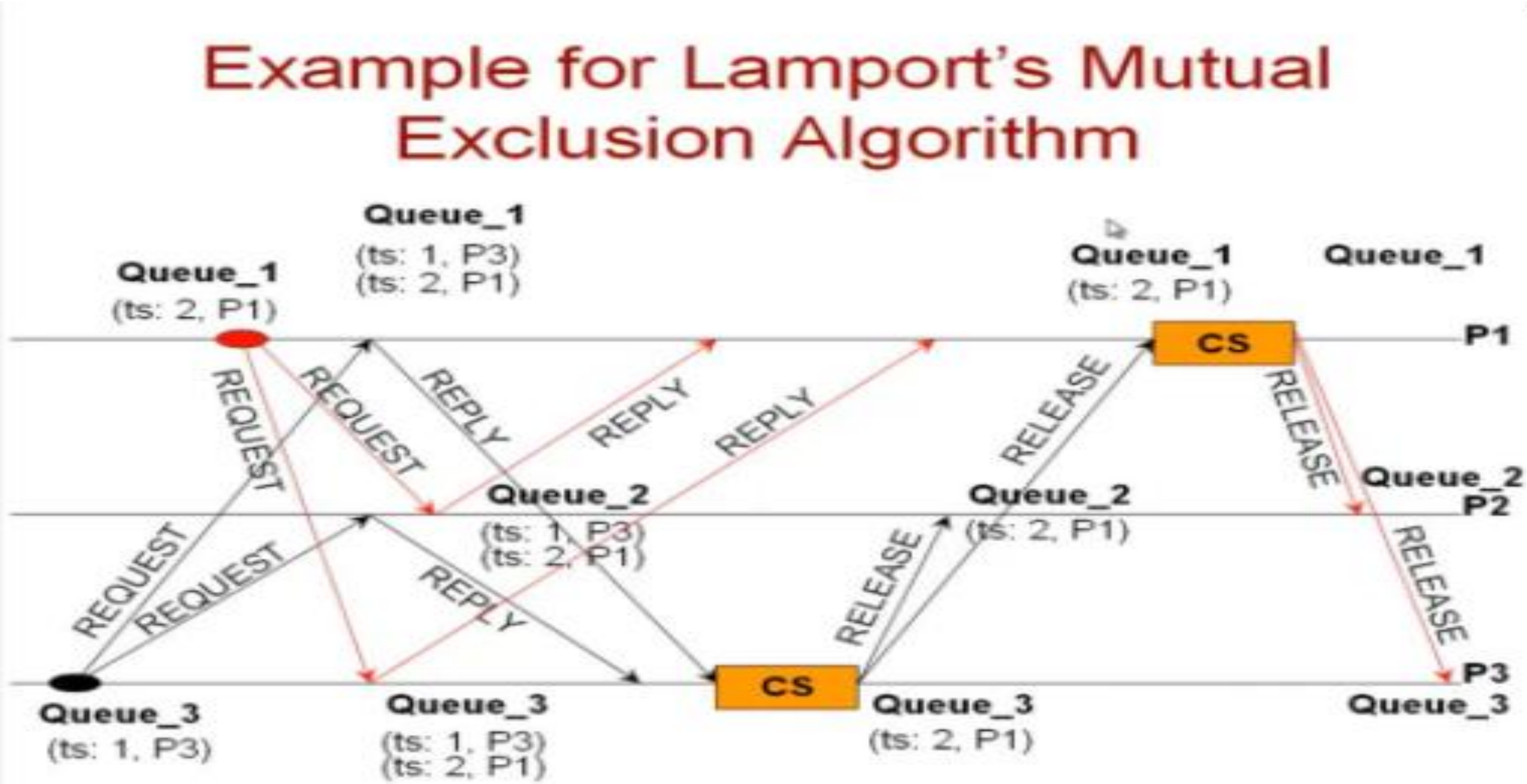
LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM



LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM



LAMPORT'S DISTRIBUTED MUTUAL EXCLUSION ALGORITHM



RICART– AGRAWALA ALGORITHIM

Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala. This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm. Like Lamport's Algorithm, it also follows permission-based approach to ensure mutual exclusion.

In this algorithm:

-Two type of messages (**REQUEST** and **REPLY**) are used and communication channels are assumed to follow FIFO order.

-A site sends a **REQUEST** message to all other site to get their permission to enter critical section.

- A site sends a **REPLY** message to other site to give its permission to enter the critical section.

-A timestamp is given to each critical section request using Lamport's logical clock.

Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

RICART-AGRWALA ALGORITHM

Algorithm:

To enter Critical section:

-When a site S_i wants to enter the critical section, it sends a timestamped **REQUEST** message to all other sites.

-When a site S_j receives a **REQUEST** message from site S_i, It sends a **REPLY** message to site S_i if and only if:

-Site S_j is neither requesting nor currently executing the critical section.

In case Site S_j is requesting, the timestamp of Site S_i's request is smaller than its own request.

- Other wise the request is deferred by site S_j.

To execute the critical section:

-Site S_i enters the critical section if it has received the **REPLY** message from all other sites.

To release the critical section:

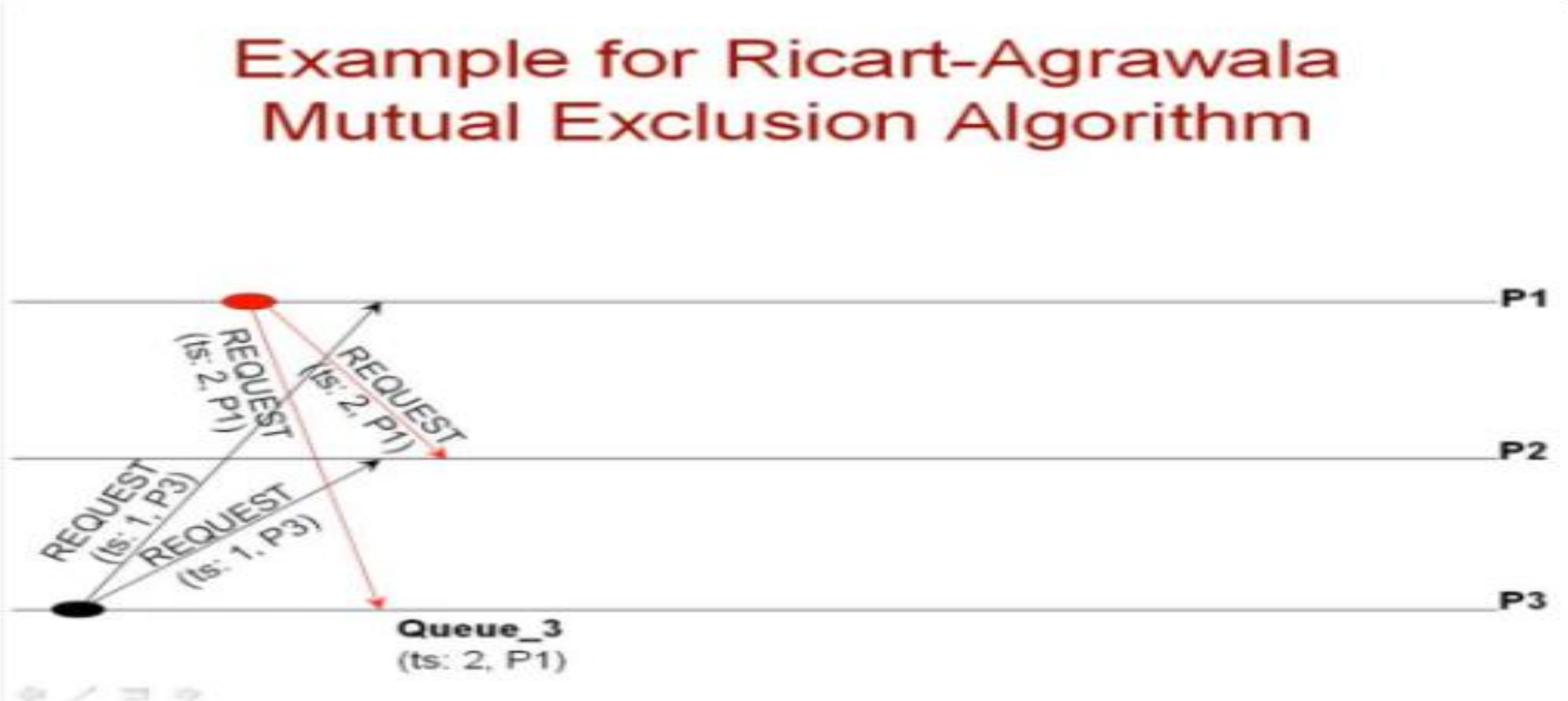
-Upon exiting site S_i sends **REPLY** message to all the deferred requests.

RICART-AGRWALA ALGORITHM

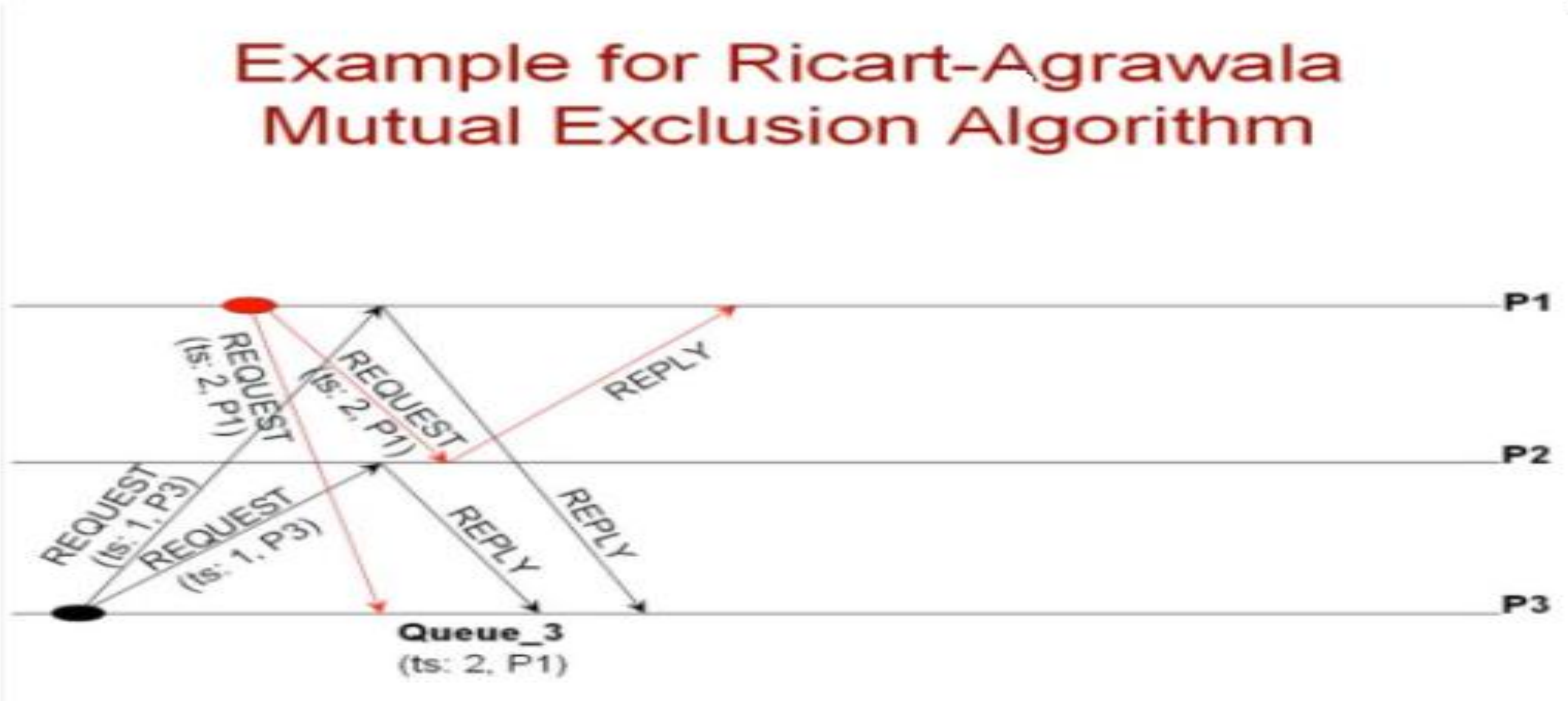
Ricart-Agrwala Algorithm

- The idea is to combine the REPLY and RELEASE messages of the Lamport's algorithm.
- A process Pi broadcasts its CS REQUEST message to all other processes.
- A process Pj replies for a CS REQUEST message from Pi only if:
  - It is neither requesting access to the CS nor executing the CS (or)
  - It has requested for the CS; but, its timestamp is larger than the REQUEST from Pi
  - Otherwise, the reply is deferred.
- A process Pi enters the CS only after getting the REPLY messages from all other processes.
- Upon exiting from the CS, a process Pi sends out the deferred REPLY messages.
- Unless you are deferring your REPLY for a CS REQUEST, there is no need to store the REQUEST in your queue.

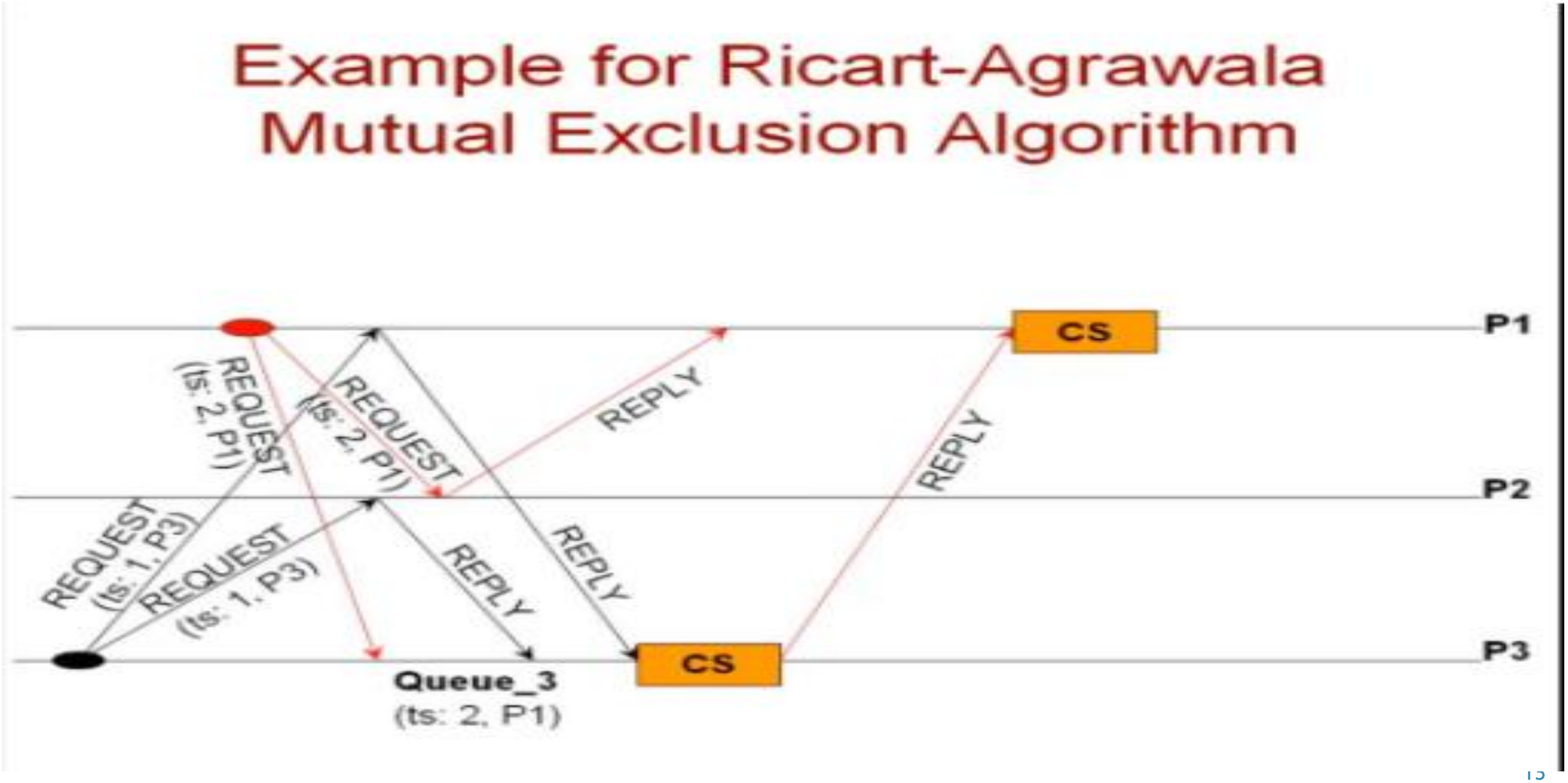
RICART-AGRWALA ALGORITHM



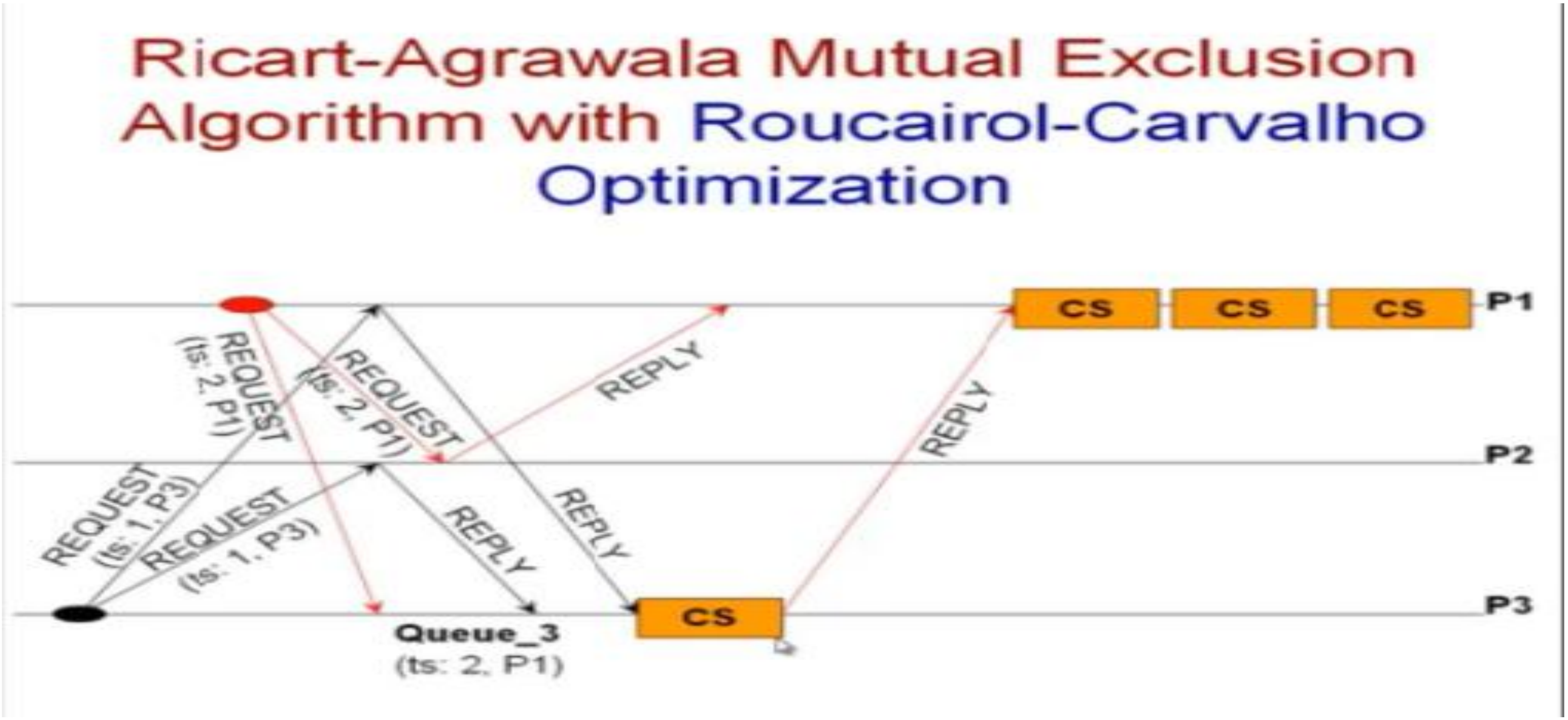
RICART-AGRWALA ALGORITHM



RICART-AGRWALA ALGORITHM



RICART-AGRWALA ALGORITHM



MAEKAWA'S ALGORITHM

Maekawa's Algorithm is quorum-based approach to ensure mutual exclusion in distributed systems. As we know, In permission-based algorithms like Lamport's Algorithm, Ricart-Agrawala Algorithm etc. a site request permission from every other site but in quorum-based approach, A site does not request permission from every other site but from a subset of sites which is called **quorum**.

In this algorithm:

Three type of messages (**REQUEST**, **REPLY** and **RELEASE**) are used.

A site sends a **REQUEST** message to all other site in its request set or quorum to get their permission to enter critical section.

A site sends a **REPLY** message to requesting site to give its permission to enter the critical section.

A site sends a **RELEASE** message to all other site in its request set or quorum upon exiting the critical section.

MAEKAWA'S ALGORITHM

Maekawa's algorithm

Maekawa associated a *voting set* V_i with each process p_i $(i = 1, 2, \ldots, N)$, where $V_i \subseteq \{p_1, p_1, \ldots, p_N\}$. The sets V_i are chosen so that, for all $i, j = 1, 2, \ldots, N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \varnothing$ – there is at least one common member of any two voting sets
- $|V_i| = K$ – to be fair, each process has a voting set of the same size
- Each process p_j is contained in M of the voting sets V_i.

MAEKAWA'S ALGORITHM

Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;

For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i ;
  Wait until (number of replies received = K);
  state := HELD;

On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i ;
    voted := TRUE;
  end if

MAEKAWA'S ALGORITHM

Maekawa's algorithm

For p_i *to exit the critical section*
    state := RELEASED;
    Multicast *release* to all processes in V_i;

On receipt of a release from p_i *at* p_j
    if (queue of requests is non-empty)
    then
            remove head of queue – from p_k, say;
            send *reply* to p_k;
            voted := TRUE;
    else
            voted := FALSE;
    end if

MAEKAWA'S ALGORITHM

Example. Let there be seven processes 0, 1, 2, 3, 4, 5, 6

S0 = {0, 1, 2}

S1 = {1, 3, 5}

S2 = {2, 4, 5}

S3 = {0, 3, 4}

S4 = {1, 4, 6}

S5 = {0, 5, 6}

S6 = {2, 3, 6}

MAEKAWA'S ALGORITHM

Version 1 {Life of process I}

1. Send timestamped ***request*** to each process in $\mathbf{S_i}$.
2. Request received → 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 → **unlock** yourself. Then send ack to the next process with the lowest timestamp.

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$

MAEKAWA'S ALGORITHM

ME1. ***At most one process can enter its critical section at any time.***

Let **i** and **j** attempt to enter their Critical Sections

$\mathbf{S_i} \cap \mathbf{S_j} \neq \varphi$ there is a process $\mathbf{k} \in \mathbf{S_i} \cap \mathbf{S_j}$

Process **k** will **never** send ack to both.

So it will act as the arbitrator and establishes ME1

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$

MAEKAWA'S ALGORITHM

ME2. No deadlock. Unfortunately deadlock is possible! Assume 0, 1, 2 want to enter their critical sections.

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}

From $\mathbf{S_0}$**= {0,1,2}**, **0,2** send *ack* to **0**, but **1** sends *ack* to **1**;

From $\mathbf{S_1}$**= {1,3,5}**, **1,3** send *ack* to **1**, but **5** sends *ack* to **2**;

From $\mathbf{S_2}$**= {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)**, , and 2 waits for 0 **(to send a release)**, . So deadlock **is** possible!

CONTENTION BASED ALGORITHMS

Thanks