Mutual exclusion

Click to add text

- Mutual exclusion is a property that ensure that a process cannot access a resource while it is already being used by another process.

- Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time.

- Distributed mutual exclusion algorithms can be classified into different categories.

  - ❖ Token based approach
  - ❖ Non token based approach
  - ❖ Permission based approach
  - ❖ Quorum based approach

Token based approach

- In token-based solutions mutual exclusion is achieved by passing a special message between the processes, known as a **token**.

- There is only one token available and who ever has that token is **allowed to access the shared resource**.

- When finished, the token is passed on to a next process.

- If a process having the token is not interested in accessing the resource, it simply passes it on.

Properties

1. They avoid **starvation**
2. Free from **deadlock**

Drawback

- when the token is lost (process holding it crashed), then new token is created.
- It will take time to create new token.

1. A Token Ring Algorithm

- Here we have a bus network, with no inherent ordering of the processes.
- In software, a logical ring is constructed in which each process is assigned a position in the ring.
- The ring positions may be allocated in numerical order of network addresses or some other means.
- It does not matter what the ordering is.
- All that matters is that each process knows who is next in line after itself.

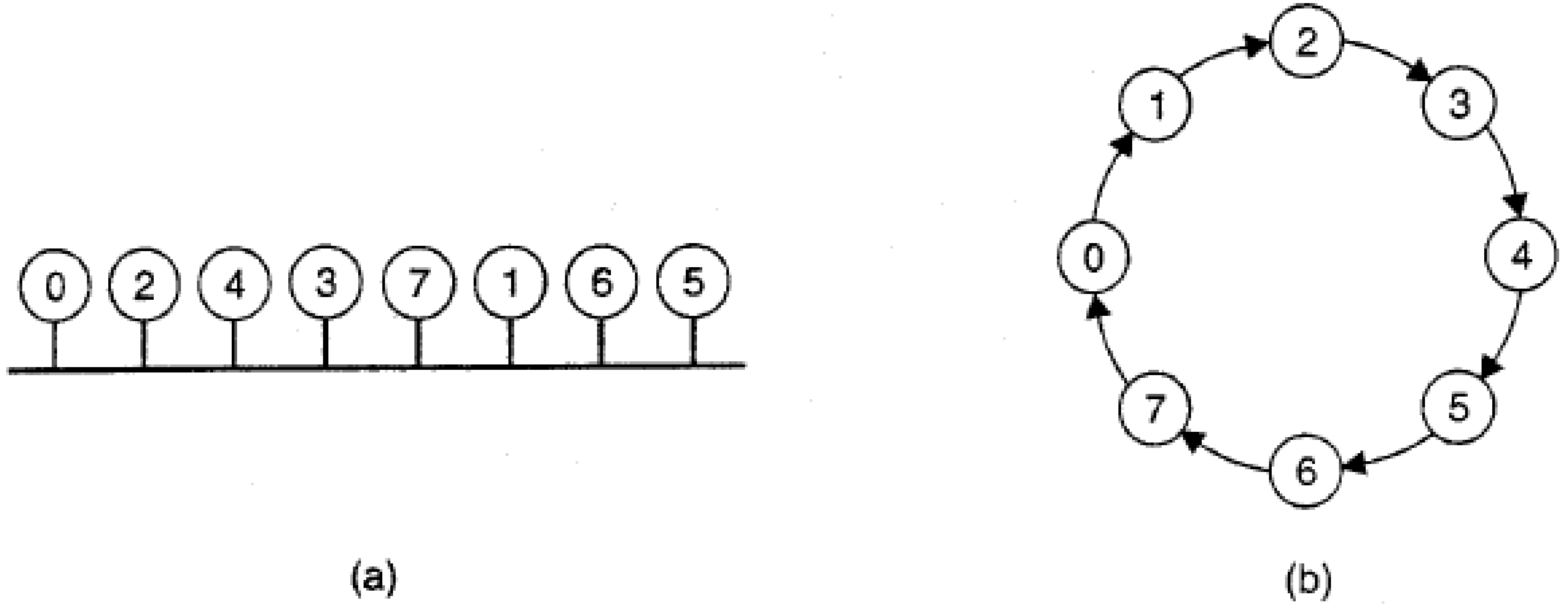


Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

- When the ring is initialized, process 0 is given a token.
- The token circulates around the ring.
- It is passed from process *k* to process *k* +1 (modulo the ring size) in point-to-point messages.
- When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource.
- If so, the process goes ahead, does all the work it needs to, and releases the resources.
- After it has finished, it passes the token along the ring.

- If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along.
- As a consequence, when no processes need the resource, the token just circulates at high speed around the ring.
- **starvation** cannot occur.

Click to add text

- Once a process decides it wants to have access to the resource, at worst it will have to wait for every other process to use the resource.
- As usual, this algorithm has problems too. If the **token is ever lost, it must be regenerated.**

Permission based approach

- Process wanting to access the resource first requires the **permission of other process**

    1. Centralized Algorithm
    2. Decentralized algorithm

1. Centralized Algorithm

- One process is elected as the coordinator.
- Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission.
- If no other process is currently accessing that resource, the coordinator sends back a reply granting permission.
- When the reply arrives, the requesting process can go ahead.

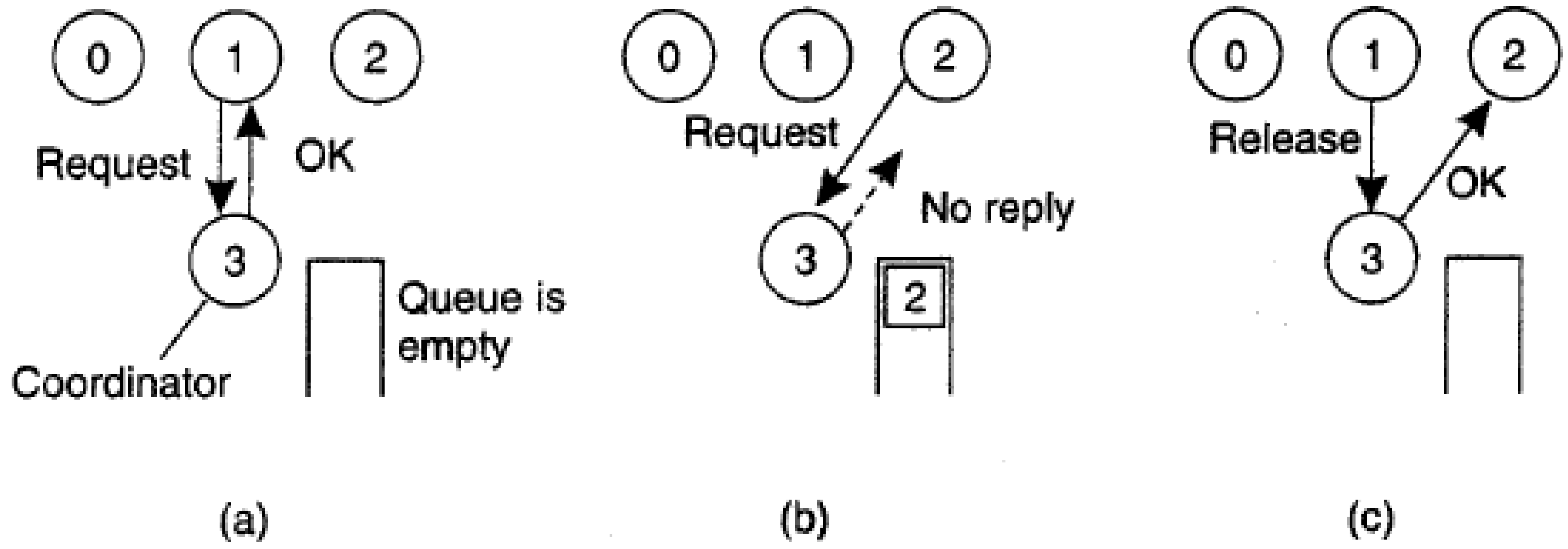


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

- It is easy to see that the algorithm guarantees mutual exclusion: the **coordinator only lets one process at a time to the resource.**

- It is also fair, since requests are granted in the order in which they are received.

- No process ever waits forever (no starvation).

- The scheme is easy to implement, too, and requires only three messages per use of resource (**request, grant, release**).

- The centralized approach also has shortcomings.
- The **coordinator is a single point of failure, so if it crashes, the entire system may go down**.
-
- If processes normally block after making a request, they **cannot distinguish a dead coordinator from "permission denied"** since in both cases no message comes back.

2. Decentralized algorithm

- Based on voting
- If a process wants some resources ,it ask all other processes that whether they want that resource.
- They will reply need or don’t need messages.
- If $m > n/2$ then requested process can access the resources.

Non Token based approach

- Here two or more successive rounds of messages are exchanged between process to determine which process is to enter critical section next.

- It uses timestamp concept

  1. Lamport’s algorithm
  2. Ricart Agarwala algorithm

1. Lamport’s algorithm

- **Permission based** algorithm.

- **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 .**

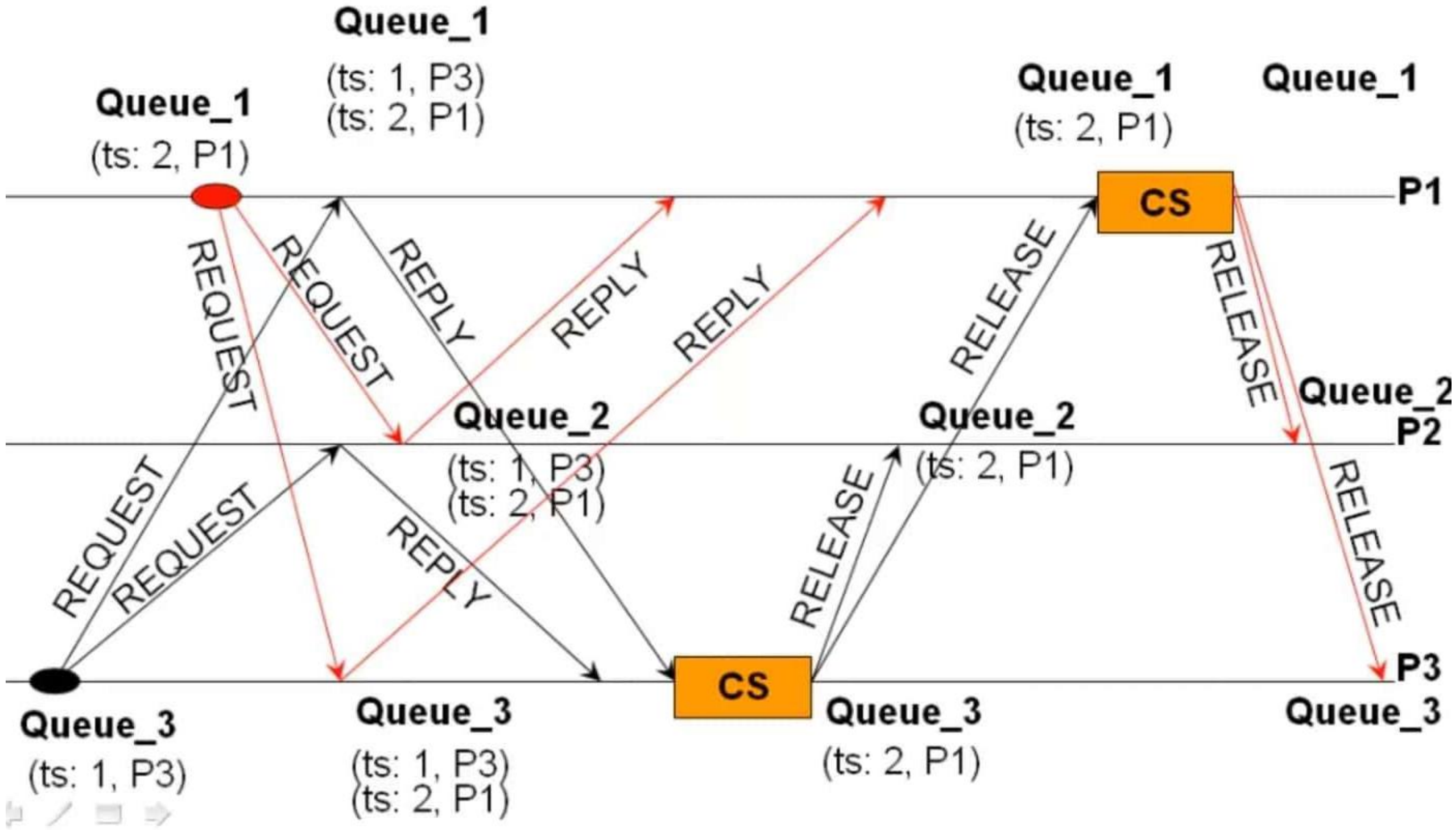
- Request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.

In this algorithm:

- Three type of messages (**REQUEST**, **REPLY** and **RELEASE**) are used.
  - A process send a **REQUEST** message to all other process to get their permission to enter critical section.
  - A process send a **REPLY** message to requesting process to give its permission to enter the critical section.
  - A process send a **RELEASE** message to all other process upon exiting the critical section.

- Every site S_i , keeps a **queue to store critical section requests** ordered by their timestamps.
- **request_queue$_i$** denotes the queue of site S_i
- 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.

Example for Lamport's Mutual Exclusion Algorithm



Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on *request_queue*$_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on *request_queue*$_j$ and returns a timestamped REPLY message to S_i.

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i's request is at the top of *request_queue*$_i$.

Releasing the critical section

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

Algorithm 9.1 Lamport's algorithm.

2.Ricart–Agrawala Algorithm

- 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.

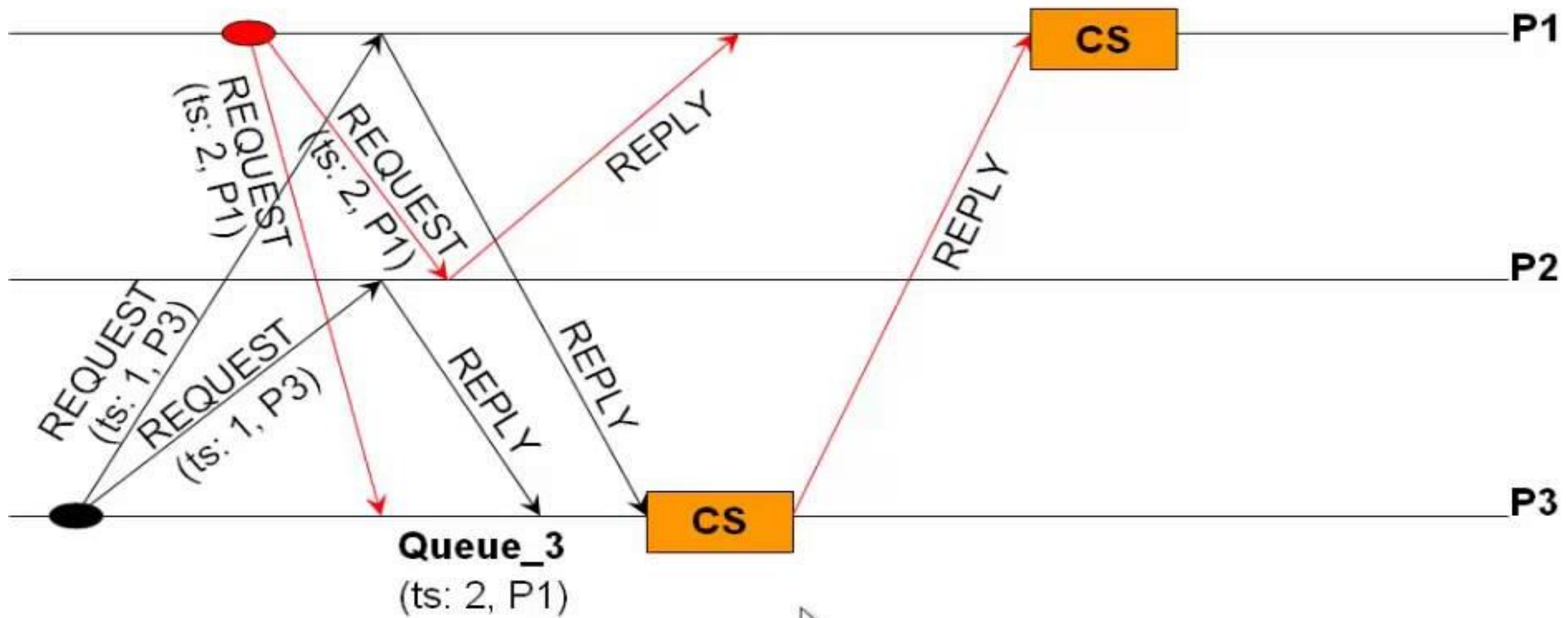
- The algorithm uses two types of messages: **REQUEST and REPLY.**

- A process sends a REQUEST message to all other processes to request their permission to enter the critical section.

- A process sends a REPLY message to a process to give its permission to that process.

- A **timestamp** is given to each critical section request using Lamport’s logical clock.
- Smaller timestamp gets high priority over larger timestamp.
-
- The execution of critical section request is always in the order of their timestamp.
- Each process pi maintains the **request-deferred array, RDi**, the size of which is the same as the number of processes in the system.
- Initially, ∀i ∀j: Rdi[j] = 0.
- Whenever pi defers the request sent by pj , it sets Rdi[j] = 1
- After it has sent a REPLY message to pj , it sets Rdi[j] = 0.

Example for Ricart-Agrawala Mutual Exclusion Algorithm



Requesting the critical section

(a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS, or if the site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. Otherwise, the reply is deferred and S_j sets $RD_j[i] := 1$.

Executing the critical section

(c) Site S_i enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

Releasing the critical section

(d) When site S_i exits the CS, it sends all the deferred REPLY messages: $\forall j$ if $RD_i[j] = 1$, then sends a REPLY message to S_j and sets $RD_i[j] := 0$.

Algorithm 9.2 The Ricart–Agrawala algorithm.

1. Maekawa's Algorithm

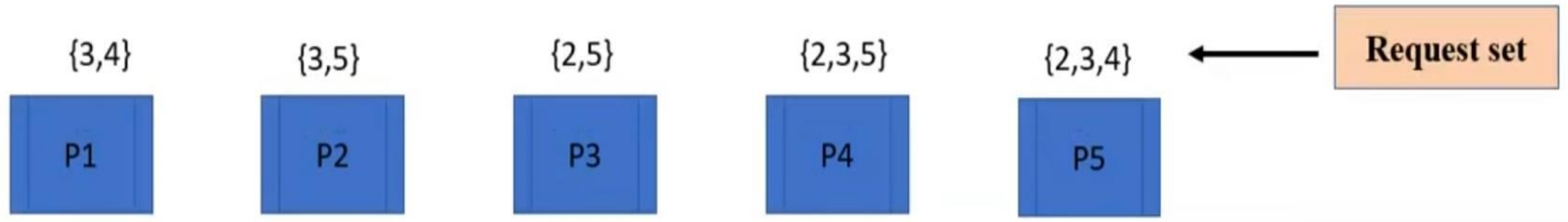
Three types of messages

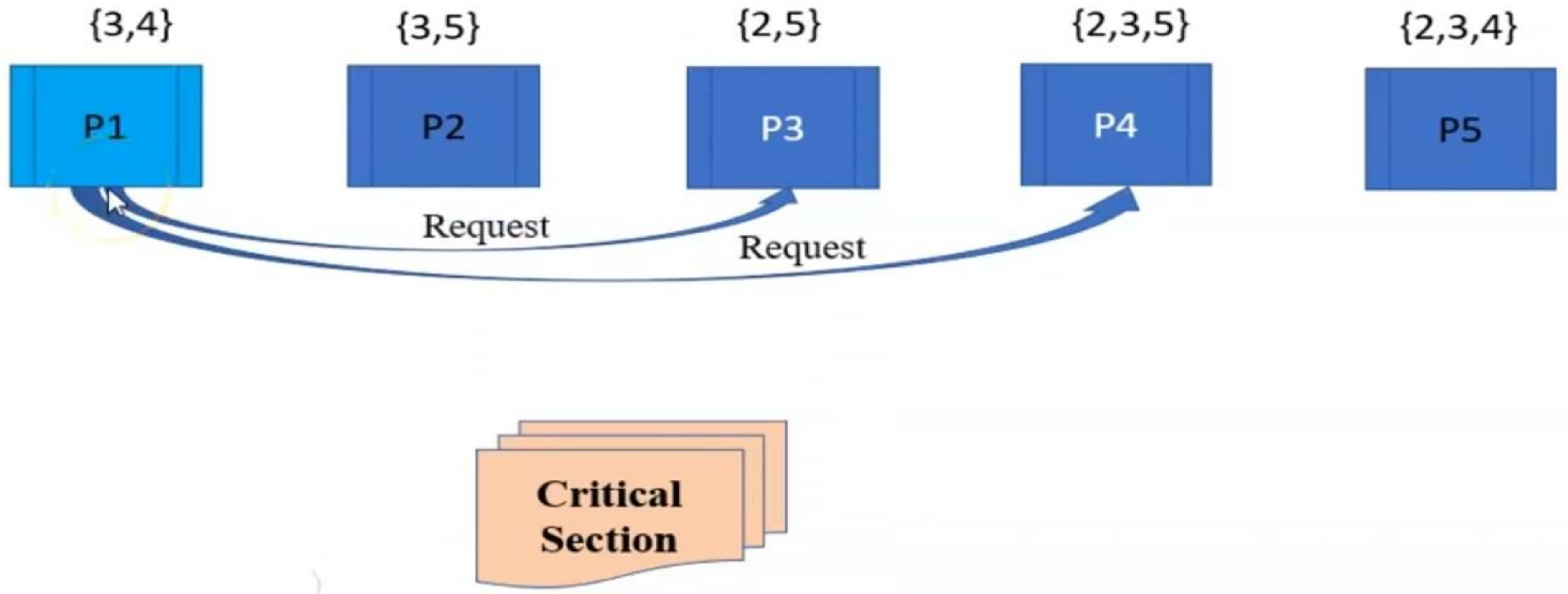
1. **Request**: This message is sent to all other sites in the quorum or request set to get their permission to enter into critical section.

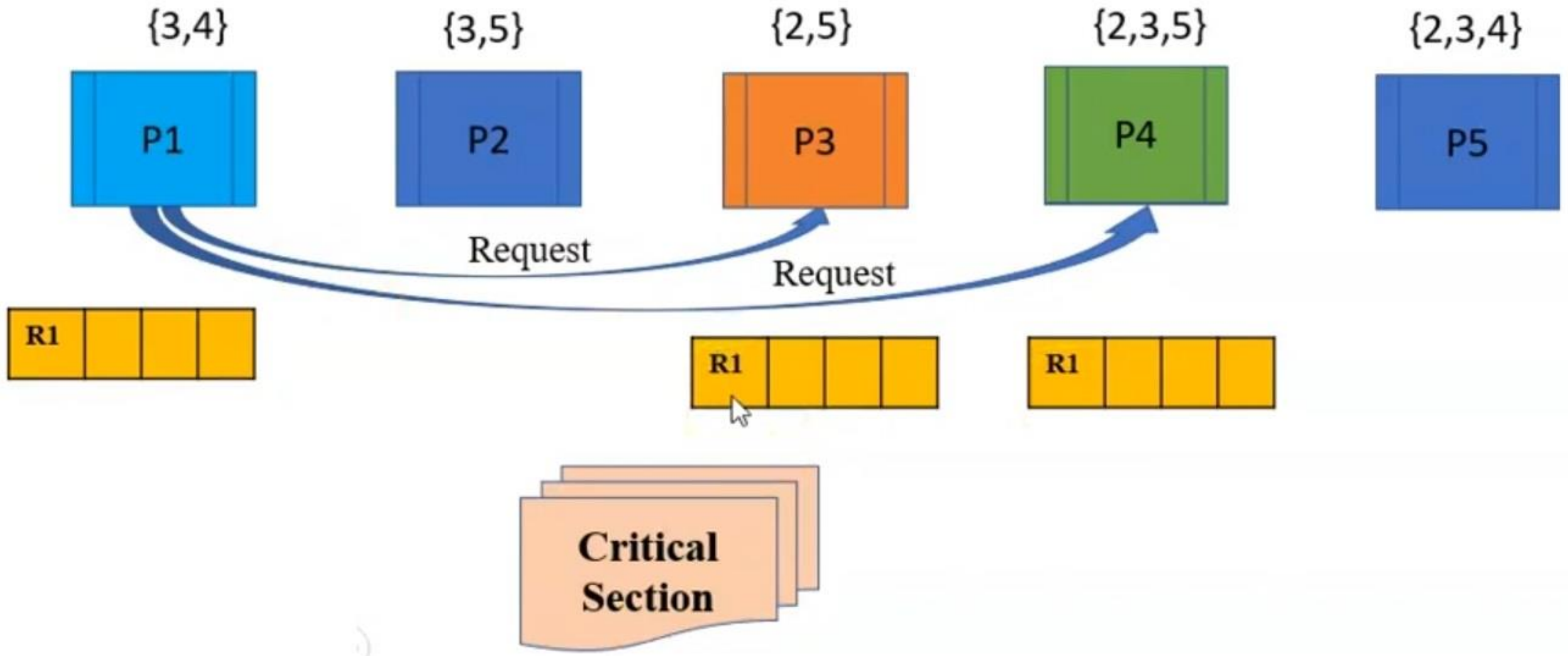
2. **Reply**: Message sent from the sites in the quorum to the requesting site

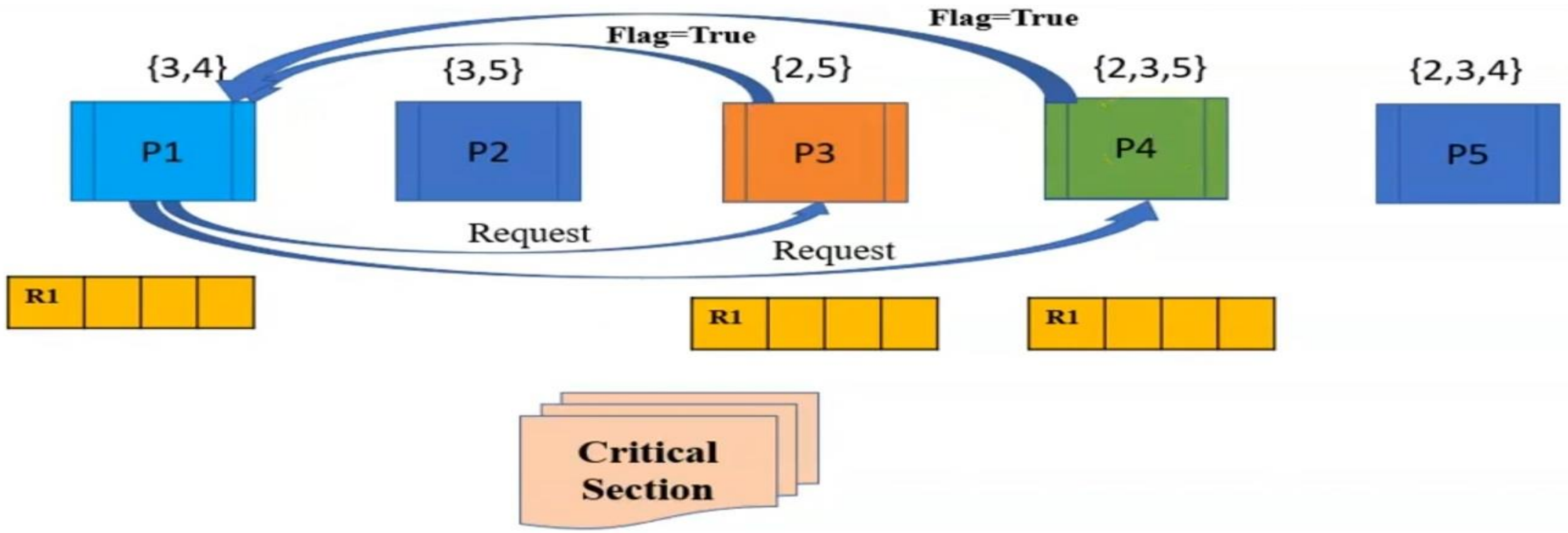
3. **Release**: After exiting the critical section this message is sent to the sites in the quorum

- Every process should belong to some group
- If any process wants to access critical section then it will send request to its group members and receive reply from group members.

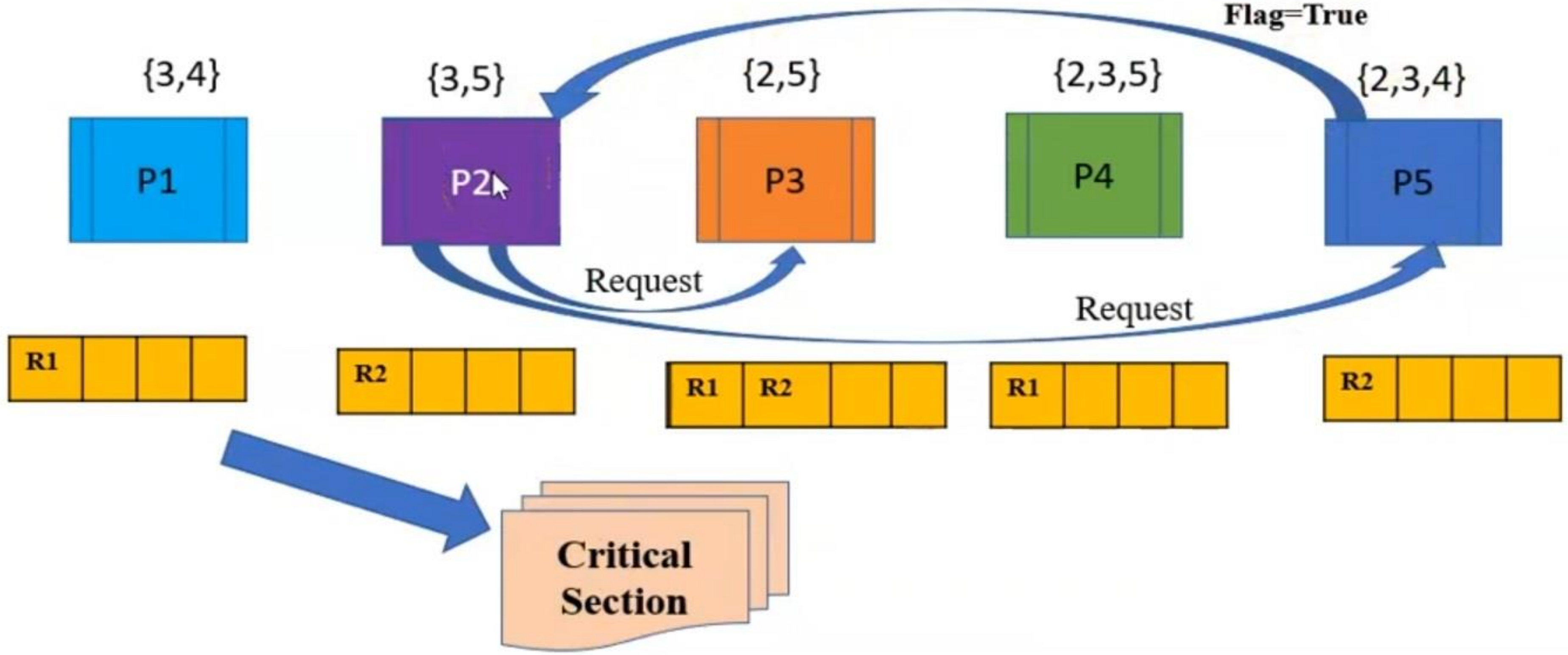

{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
Request set
P1
P2
P3
P4
P5

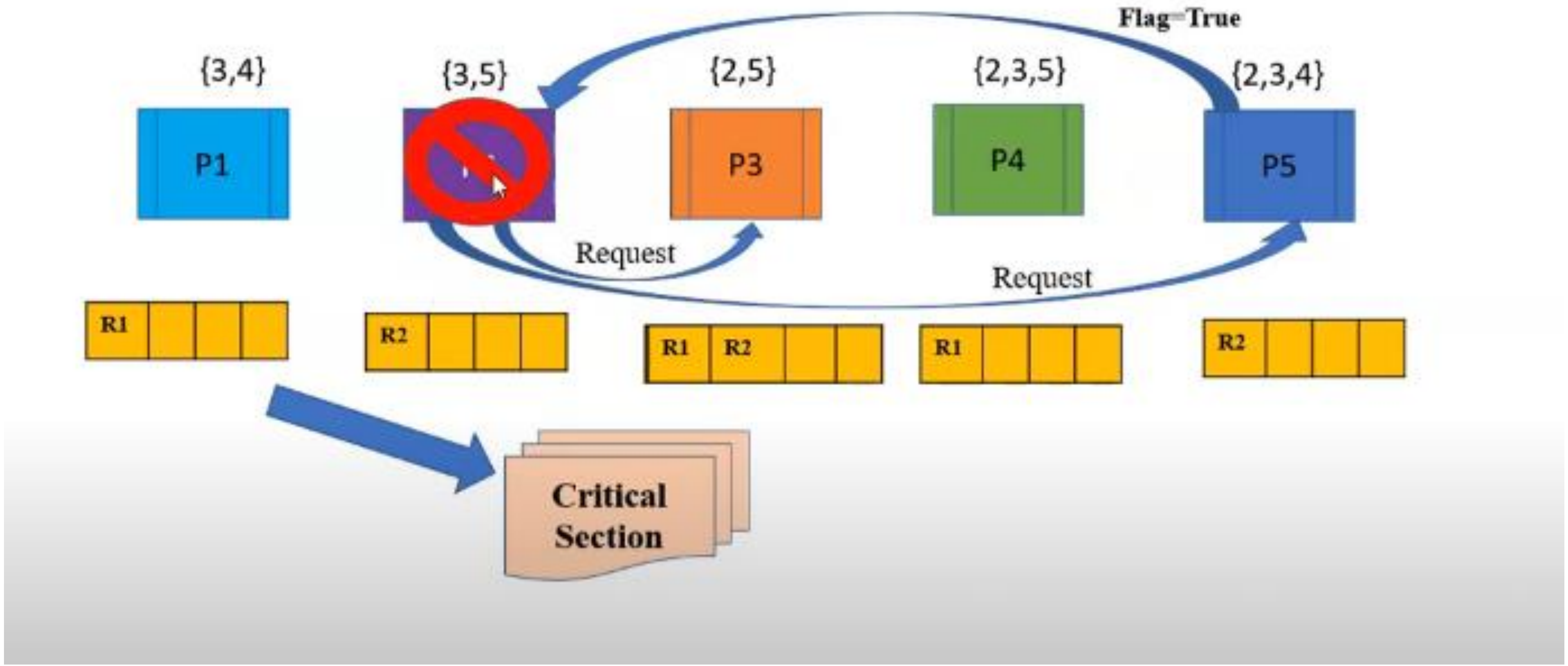

{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
P1
P2
P3
P4
P5
Request
Request
Critical Section

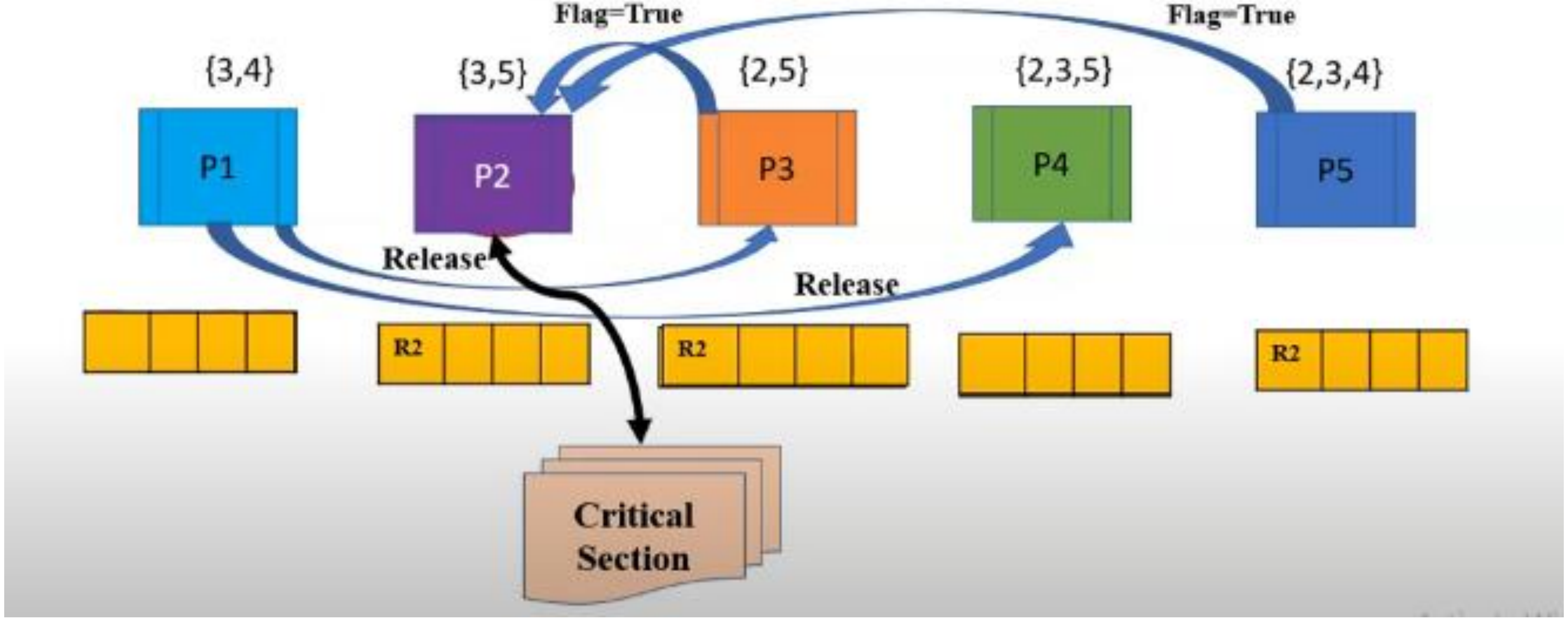

{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
P1
P2
P3
P4
P5
Request
Request
R1
R1
R1
Critical Section


Flag=True
Flag=True
{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
P1
P2
P3
P4
P5
Request
Request
R1
R1
R1
Critical Section

How Maekawa's algorithm guarantees mutual exclusion?




Flag=True
{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
P1
P3
P4
P5
Request
Request
R1
R2
R1
R2
R1
R2
Critical Section


Flag=True
Flag=True
{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
P1
P2
P3
P4
P5
Release
Release
R2
R2
R2
Critical Section

Requesting the critical section:

(a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i.

(b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.

Executing the critical section:

(c) Site S_i executes the CS only after it has received a REPLY message from every site in R_i.

Releasing the critical section:

(d) After the execution of the CS is over, site S_i sends a RELEASE(i) message to every site in R_i.

(e) When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

Algorithm 9.5 Maekawa's algorithm.