Chapter 3

Synchronization

Content….

3.1 Clock Synchronization, Logical Clocks, Election Algorithms, Mutual Exclusion, Distributed Mutual Exclusion-Classification of mutual Exclusion Algorithm, Requirements of Mutual Exclusion Algorithms, Performance measure.

3.2 Non Token based Algorithms: Lamport Algorithm, Ricart–Agrawala‘s Algorithm, Maekawa‘s Algorithm

3.3 Token Based Algorithms: Suzuki-Kasami‘s Broardcast Algorithms, Raymond‘s Tree based Algorithm, Comparative Performance Analysis.

Synchronization

- Synchronization: coordination of actions between processes.
- Processes are usually asynchronous, (operate independent of events in other processes)
- Sometimes need to cooperate/synchronize
  - For mutual exclusion
  - For event ordering (was message x from process P sent before or after message y from process Q?)

Synchronization

Need of Synchronization

- To *control access* to a single, shared resource.
- To agree on the *ordering of events*.
- Synchronization in Distributed Systems is much more difficult than in uniprocessor systems

Clock Synchronization

- Synchronization in centralized systems is primarily accomplished through shared memory
  - Event ordering is clear because all events are timed by the same clock

- Synchronization in distributed systems is harder
  - No shared memory
  - No common clock
  - Each computer has it’s own clock
  - Global time not known

What is Clock

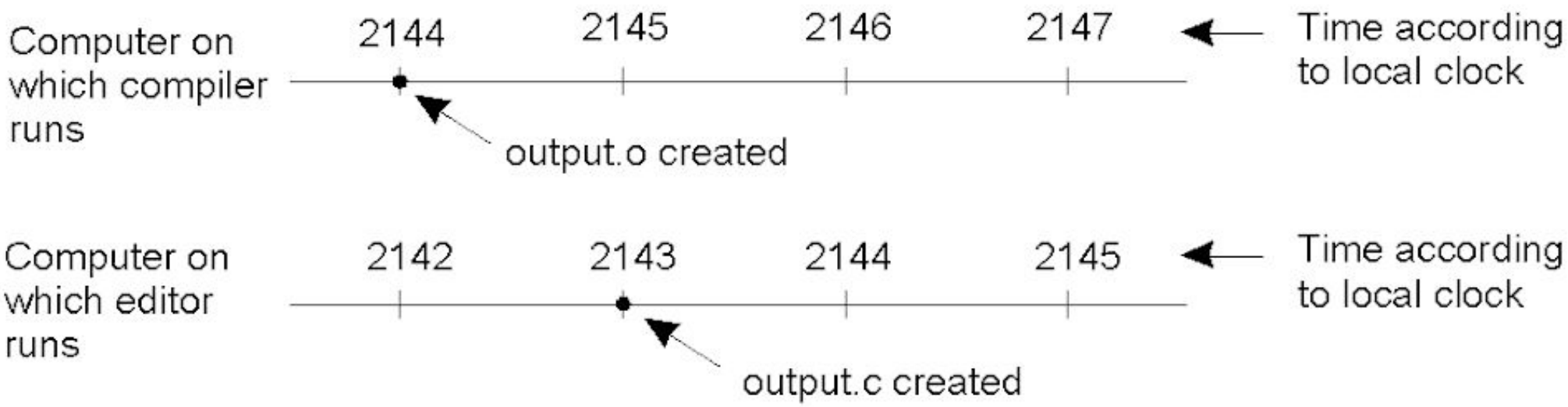
- In general, the clock refers to a microchip that regulates the timing and speed of all computer functions.
- In the chip In the chip is a crystal that vibrates at a specific frequency when electricity is applied. The shortest time any computer is capable of performing is one clock, or one vibration of the clock chip.
- The speed of a computer processor is measured in clock speed, for example, 1 MHz is one million cycles, or vibrations, a second. 2 GHz is two billion cycles, or vibrations, a second.
- In short Clock can be used to timestamp of an event on that computer.

Physical Clocks

- Physical clock example: counter + holding register + oscillating quartz crystal
  - The counter is decremented at each oscillation
  - Counter interrupts when it reaches zero
  - Reloads from the holding register
  - Interrupt = clock tick (often 60 times/second)

- The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system

Physical Clocks



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Logical clocks

- Logical clocks solves synchronization problem.
- Logical clocks shows ordering or organization of events, not based on time.
- Logical Clock is a mechanism for capturing advanced, not in reverse in DS.
- DS may have no physical synchronous global clock.
- So Logical Clock allows global ordering on events from different processes in each system.

Logical clocks

Define a relation □ on the events as follows

On the same process: a □ b *iff* *time (a) < time (b)*

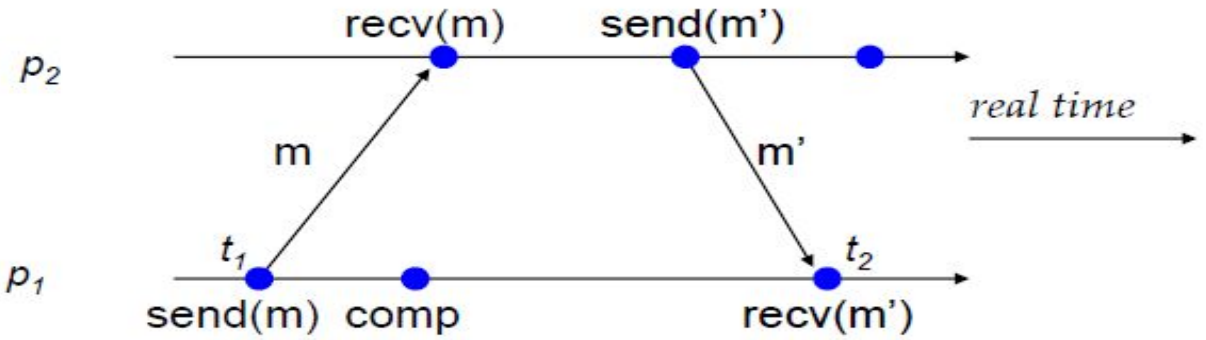
If p_1 sends m to p_2: *send(m)* □ *recv(m)*

Transitivity: *if a* □ *b and b* □ *c then a* □ *c*

□ is called the Happens-Before Relation

Events *a* and *b* are concurrent if

not (a □ b or b□ a)

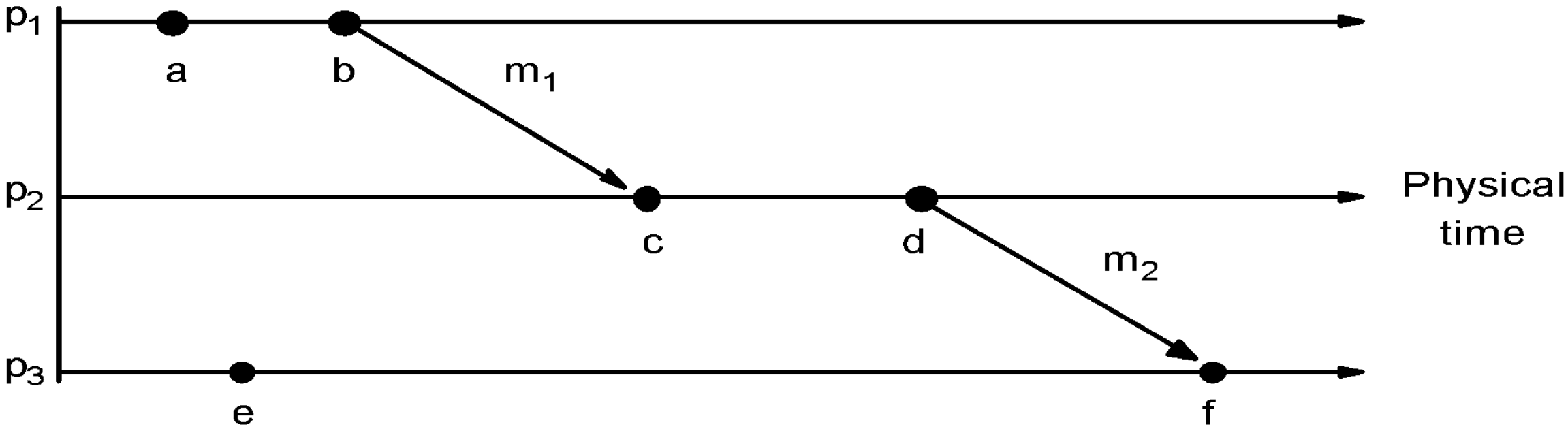


Happens-Before Relation on Events

Logical clocks

Events Occurring at Three Processes

How to construct the happen-before relation in a distributed system?



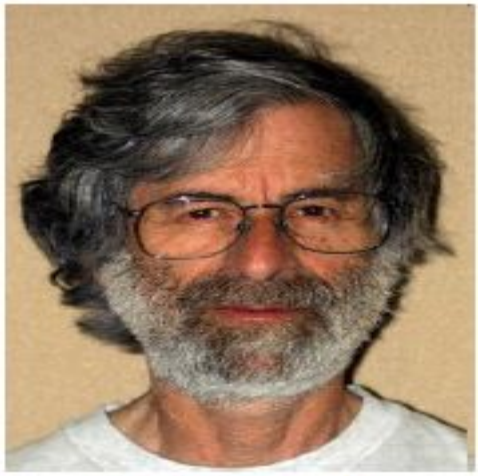
Synchronizes logical clocks

Synchronizes logical clocks

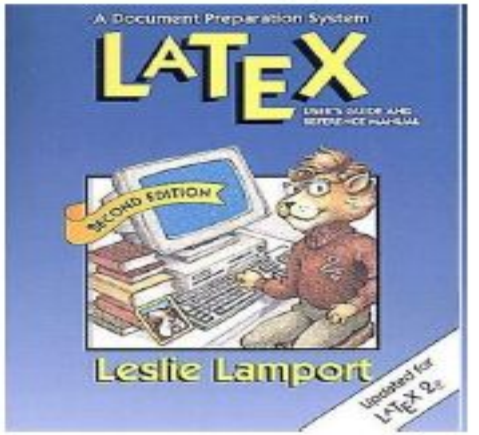
- Lamport's algorithm,
- Vector timestamps.

Lamport's Logical Clock

- First proposed by Leslie Lamport in 70's
- Lamport algorithm assigns logical timestamps to events
- The "happens-before" relation $\rightarrow$ can be observed directly in two situations:
- If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$ is true.
- If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \rightarrow b$

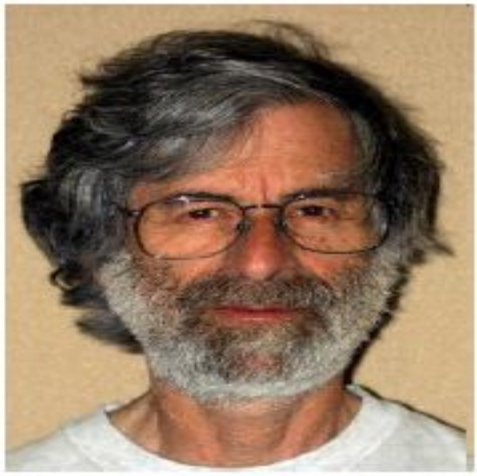


Leslie Lamport

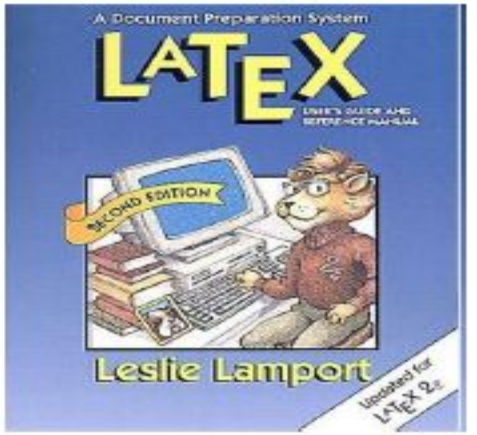


Lamport's Logical Clock

- Each process has a counter (logical clock)
- Initially logical clock is set to 0
- Process increments its counter when a *send* or a computation (*comp*) step is performed
- Counter is assigned to event as its timestamp
- send(message) event carries its timestamp
- On recv(message) event, the counter is updated by
  max(local clock, message timestamp) + 1

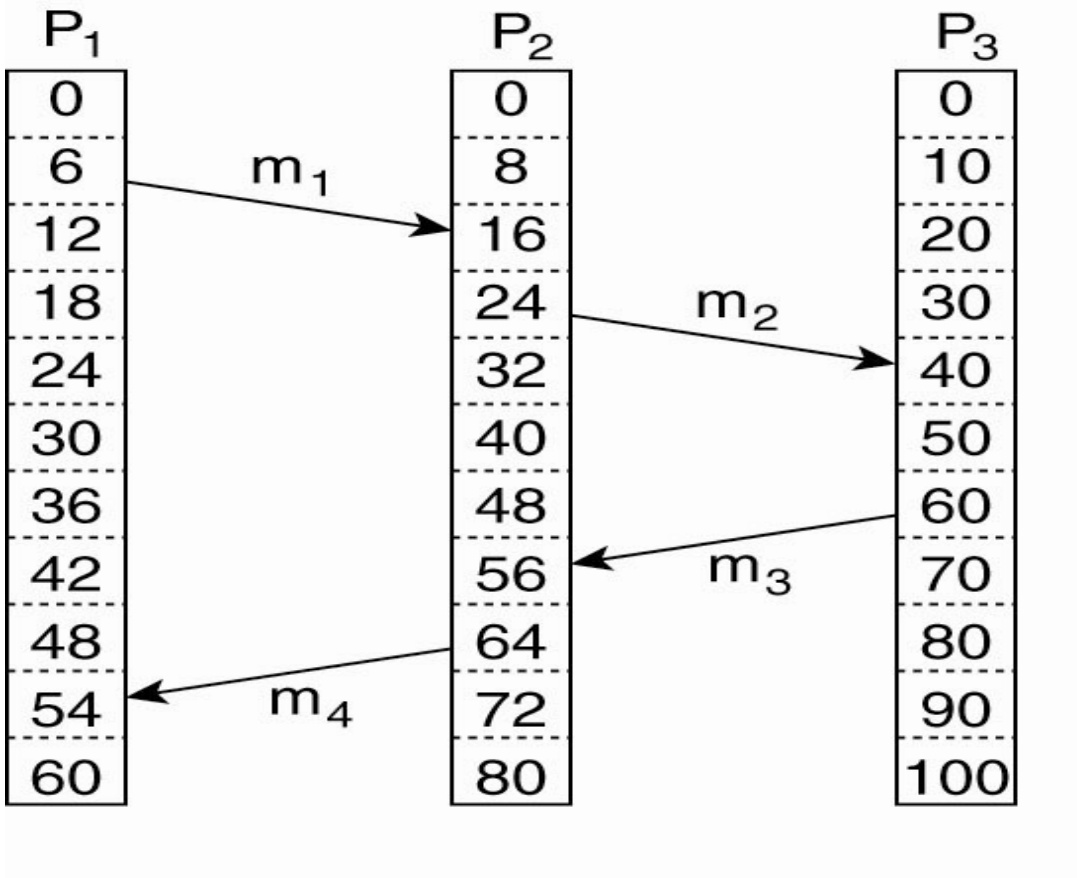


Leslie Lamport

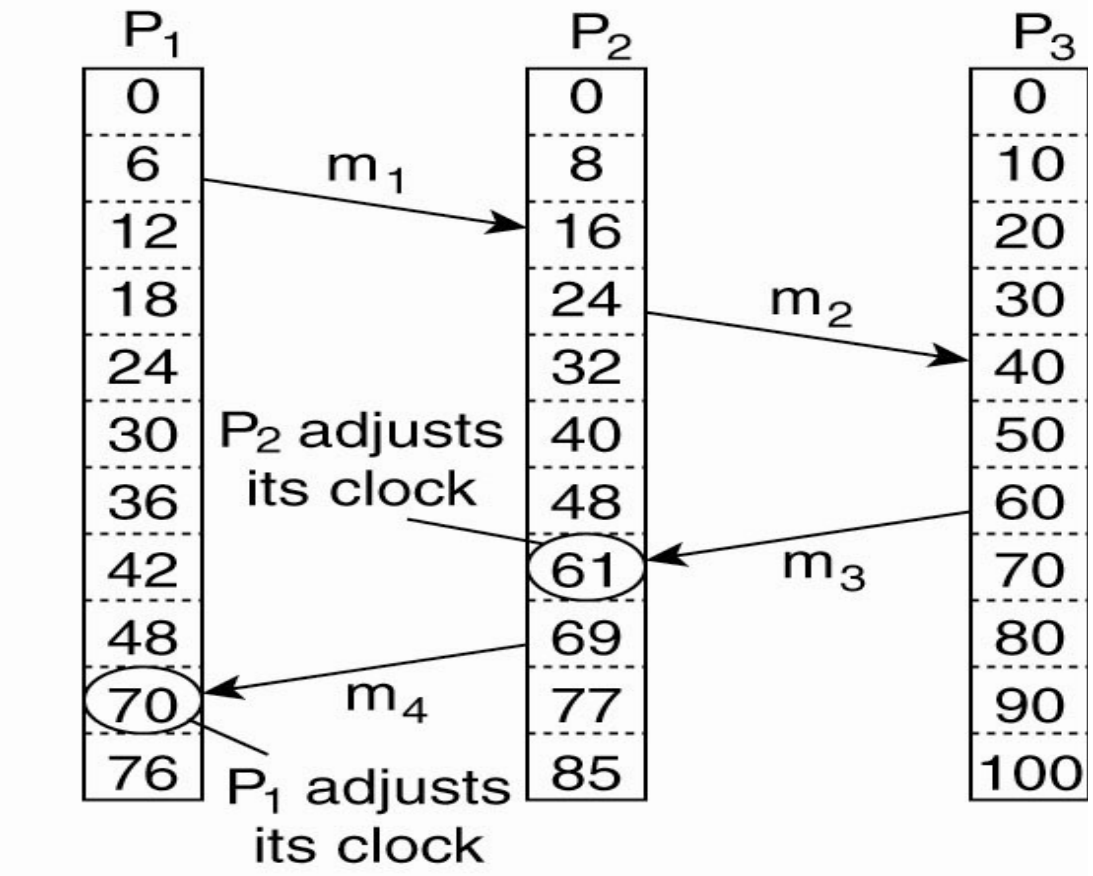


Lamport's Logical Clock

Three processes, each with its own clock. The clocks run at different rates.



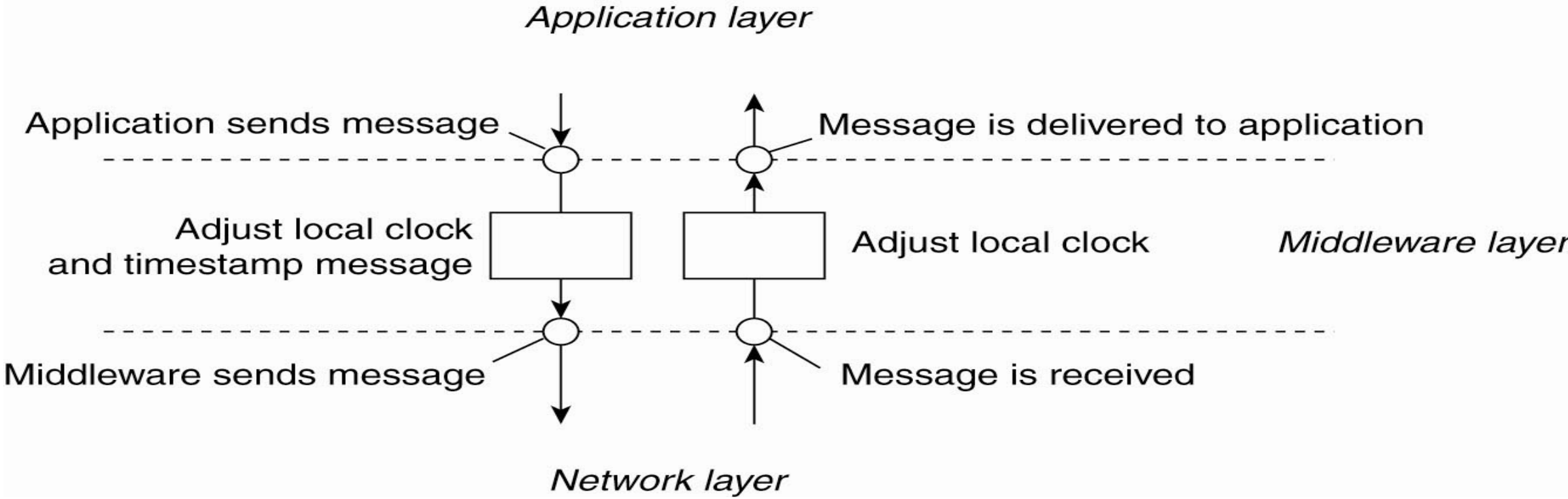
Lamport's Logical Clock



Lamport's algorithm corrects the clocks.

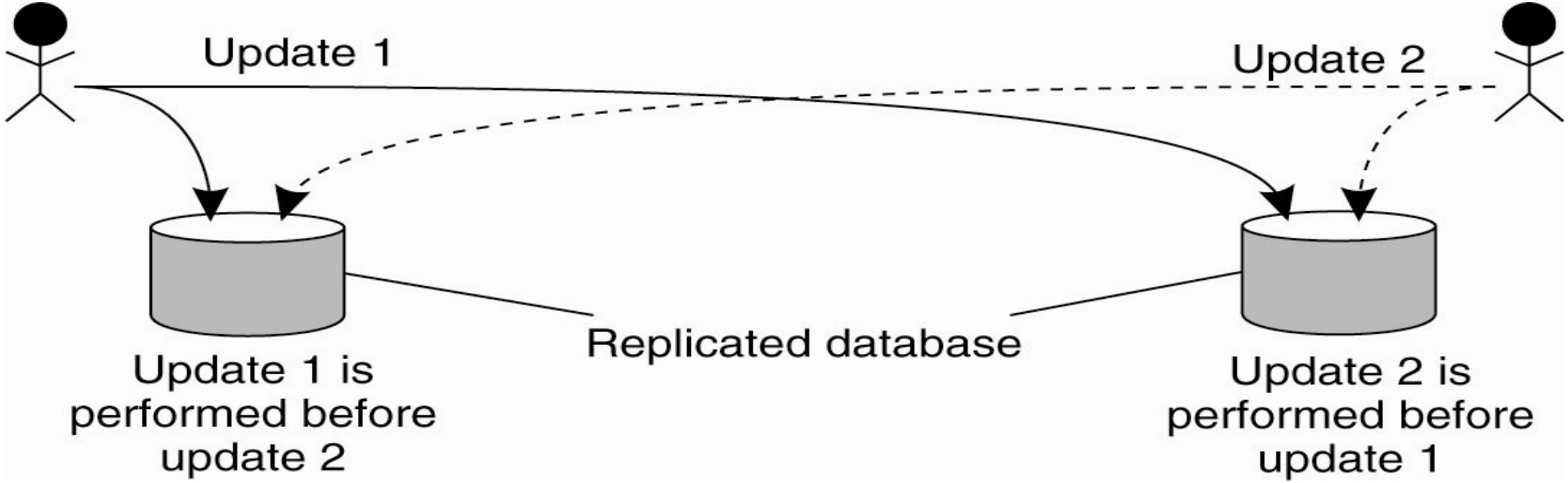
Lamport's Logical Clock

The positioning of Lamport's logical clocks in distributed systems.



Lamport's Logical Clock

Example: **Totally Ordered Multicasting**



Updating a replicated database and leaving it in an inconsistent state.

Problem with Lamport Logical Clock

Let *timestamp(a)* be the Lamport logical clock timestamp

$a \rightarrow b \Rightarrow timestamp(a) < timestamp(b)$

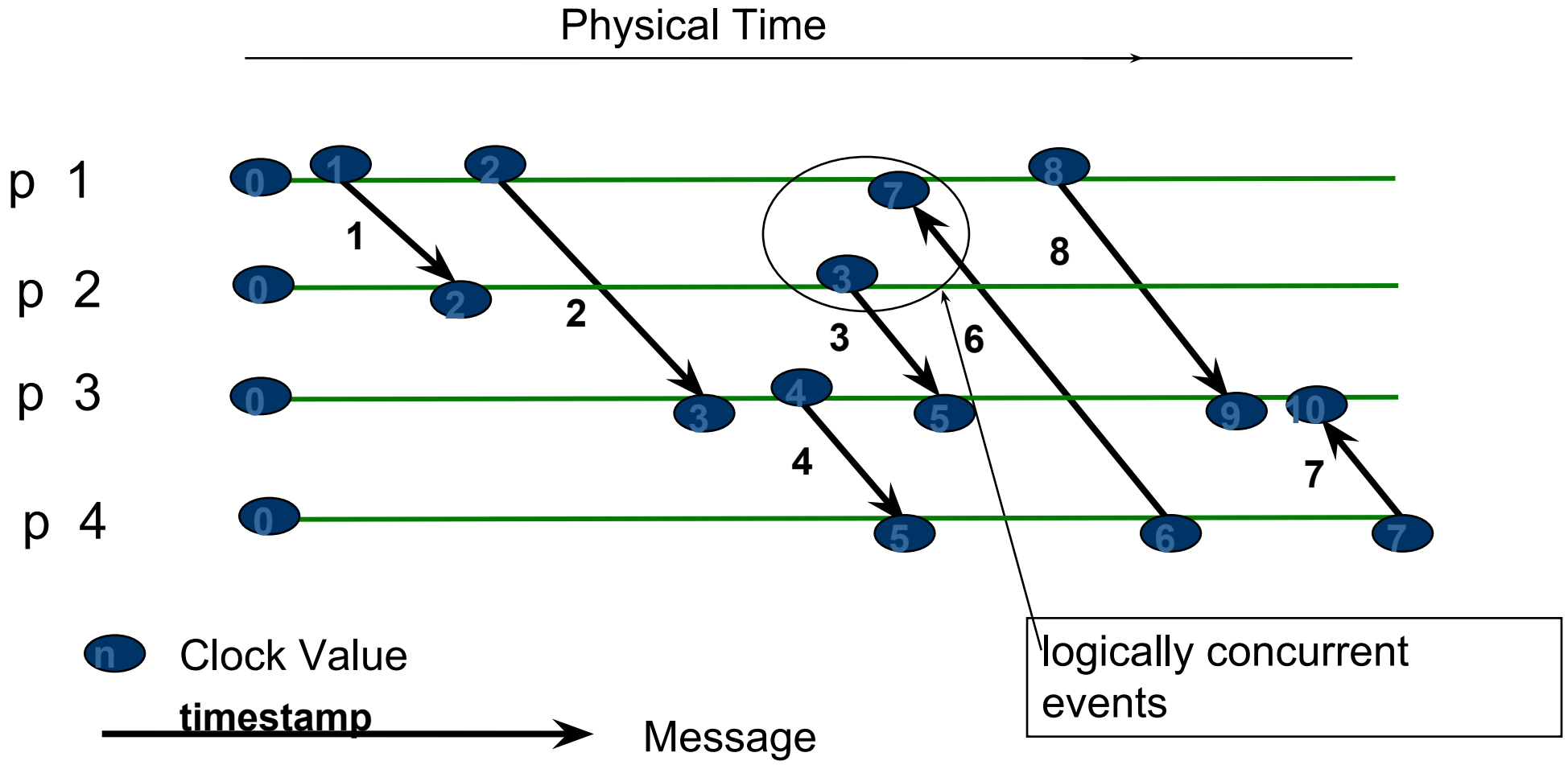
(if *a* happens before *b*, then *Lamport_timestamp(a)* < *Lamport_timestamp(b)*)

$timestamp(a) < timestamp(b) \not\Rightarrow a \rightarrow b$

(If *Lamport_timestamp(a)* < *Lamport_timestamp(b)*, it does NOT imply that *a* happens before *b*

The problem is that Lamport clocks do not capture **causality**.

Example: problem Vector Clocks



Note: Lamport Timestamps: 3 < 7, but event with timestamp 3 is concurrent to event with timestamp 7, i.e., events are not in 'happen-before' relation.

Vector Clocks

Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

1. $VC_i [i]$ is the number of events that have occurred so far at P_i.
   In other words, $VC_i [i]$ is the local logical clock at process P_i .

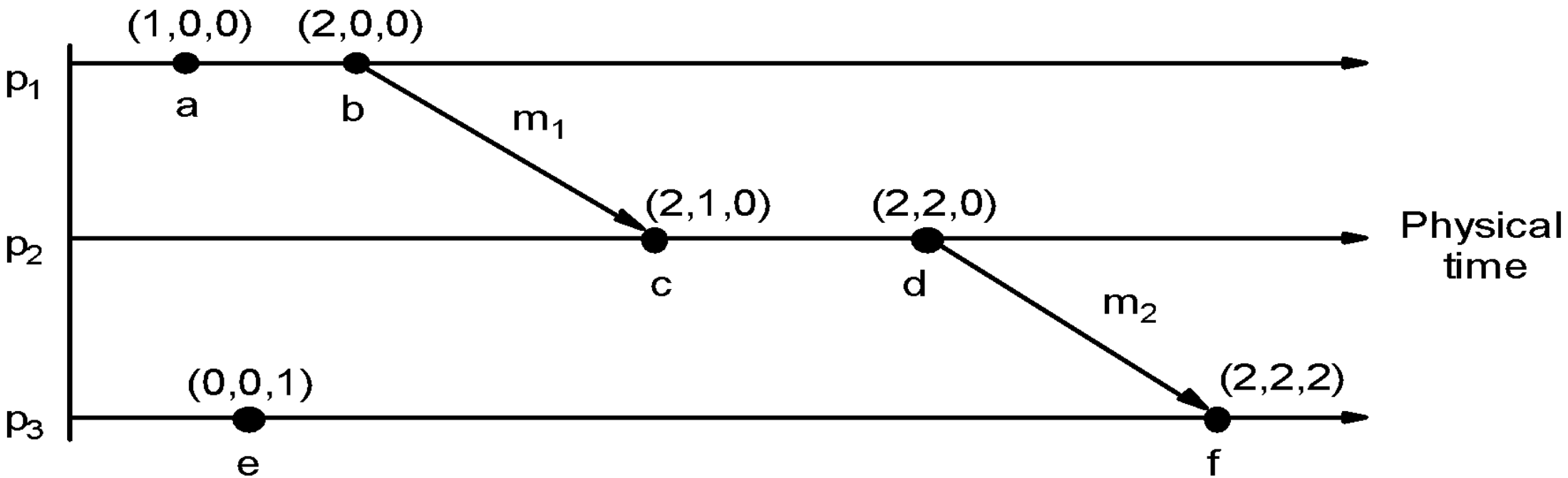
2. If $VC_i [j] = k$ then P_i knows that k events have occurred at P_j.
   It is thus P_i's knowledge of the local time at P_j .

Vector Clocks

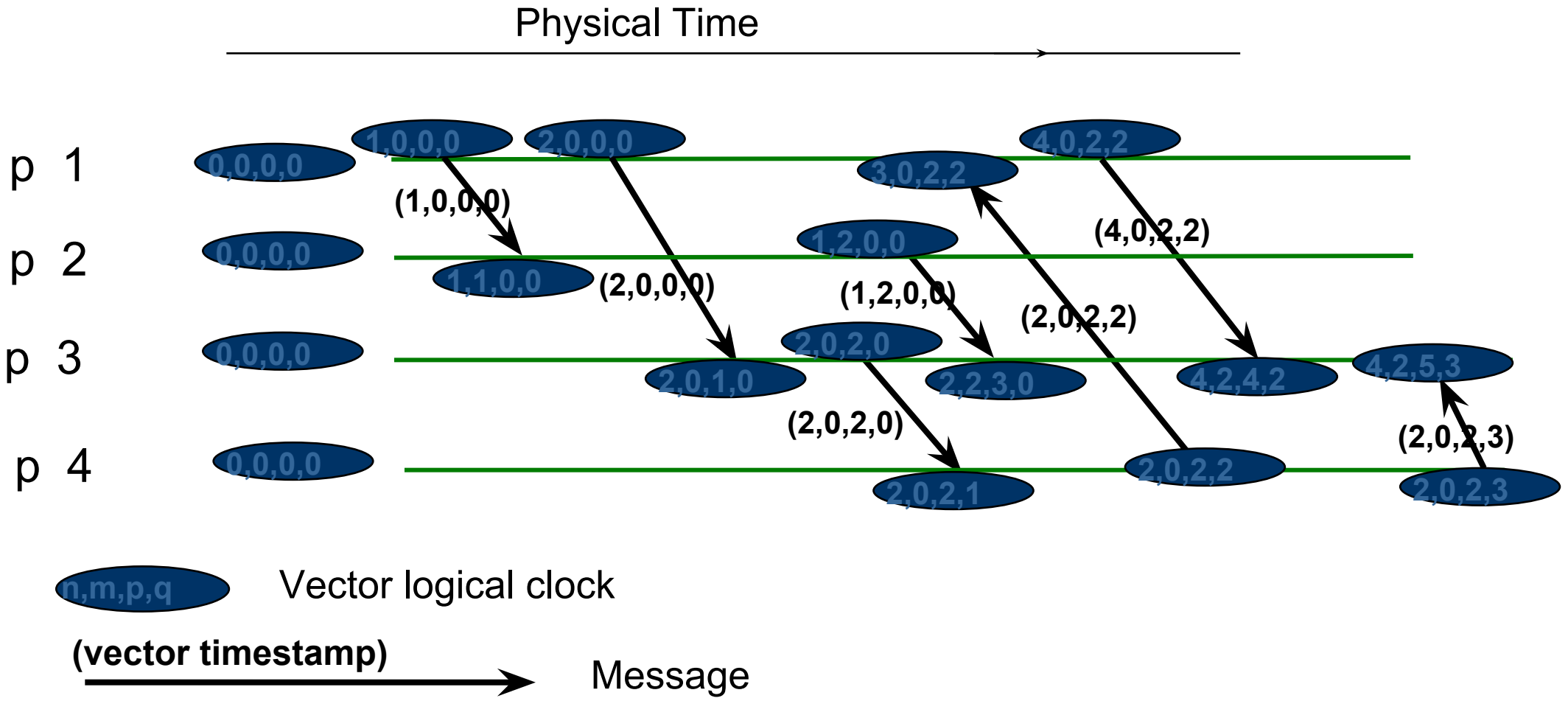
Steps carried out to accomplish property 2 of previous slide:

1. Before executing an event P_i executes
   $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets *m*'s (vector) timestamp *ts (m)* equal to VC_i after having executed the previous step.
3. Upon the receipt of a message m, process P_j adjusts its own vector by setting
   $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each *k*, after which it executes the first step and delivers the message to the application.

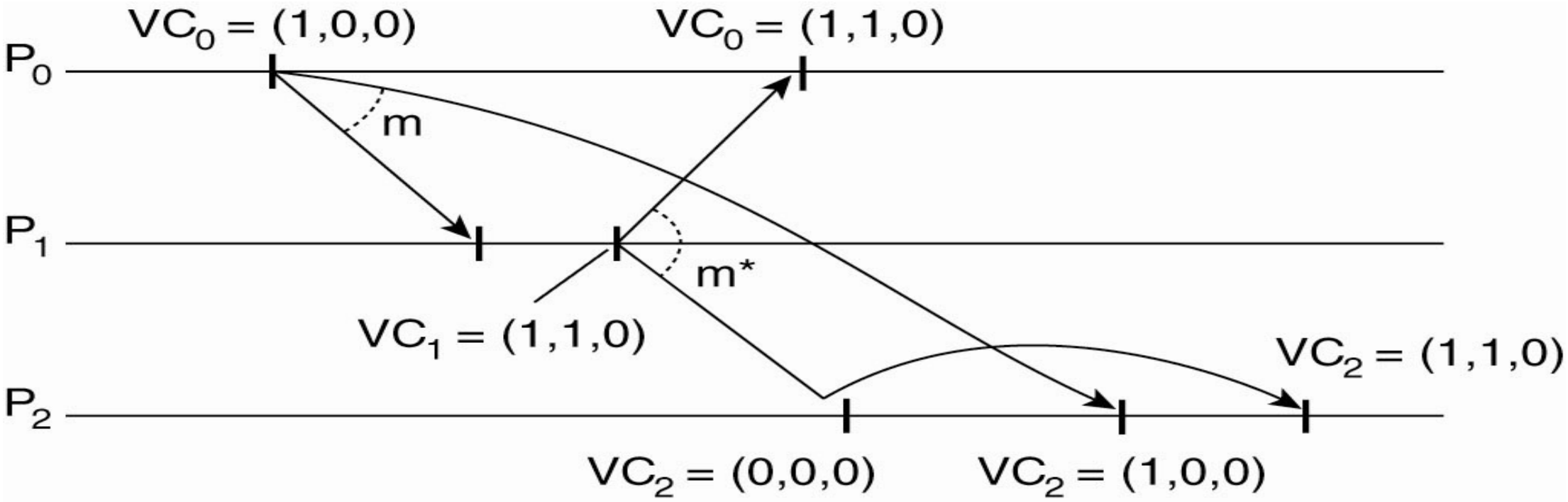
Vector Timestamps



Example: Vector Logical Time



Enforcing Causal Communication

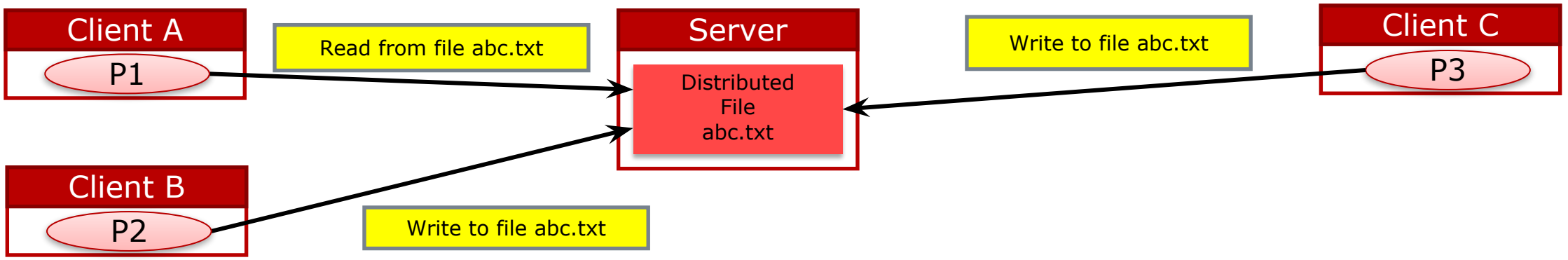


Mutual Exclusion

- Mutual Exclusion(ME) ensures that no two concurrent processes will enter in **CRITICAL SECTION** at the same time. It is basic requirement of concurrency control ,to prevent RACE CONDITION.
- **CRITICAL SECTION** refers to a period when the processes accesses a shared resource ,such as shared memory .
- **RACE CONDITION** is a situation that occurs when a device or system attempts to perform two or more operations at the same time.

Need for Mutual Exclusion

- Distributed processes need to coordinate to access shared resources
- Example: Writing a file in a Distributed File System



In uniprocessor systems, mutual exclusion to a shared resource is provided through shared variables or operating system support.

However, such support is insufficient to enable mutual exclusion of distributed entities

In Distributed System, processes coordinate access to a shared resource by passing messages to enforce *distributed mutual exclusion*

Mutual Exclusion

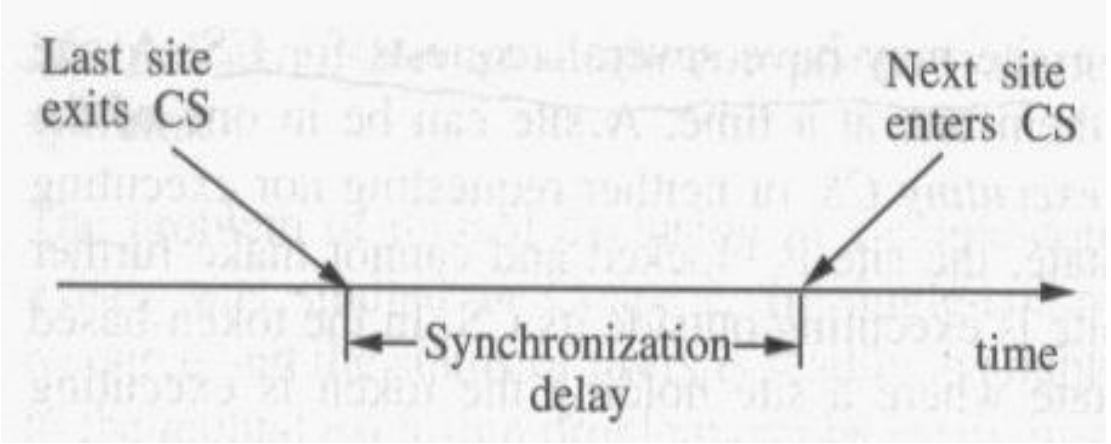
- Fundamental to distributed systems is the concurrency and collaboration among multiple processes.
- This also means that processes will need to simultaneously access the same resources.
- To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes.

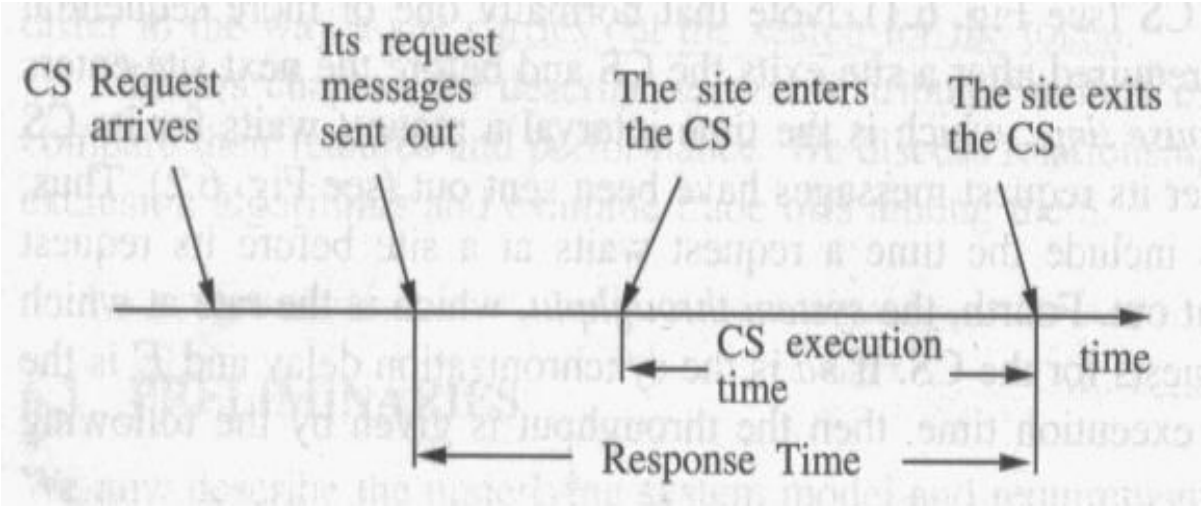
Requirements of Mutual Exclusion Algorithms

1) No deadlocks – no set of processes/sites should be permanently blocked, waiting for messages from other processes/sites in that set.
2) no starvation – no process/site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
3) fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation.)
4) fault tolerance – the algorithm is able to survive a failure at one or more process/sites

Performance Measure for Distributed Mutual Exclusion

- The number of messages per CS invocation
- Synchronization delay
  - The time required after a site leaves the CS and before the next site enters the CS
- System throughput
  - System throughput 1/(sd+E), where sd is the synchronization delay and E the average CS execution time
- Response time
  - The time interval a request waits for its CS execution to be over after its request messages have been sent out



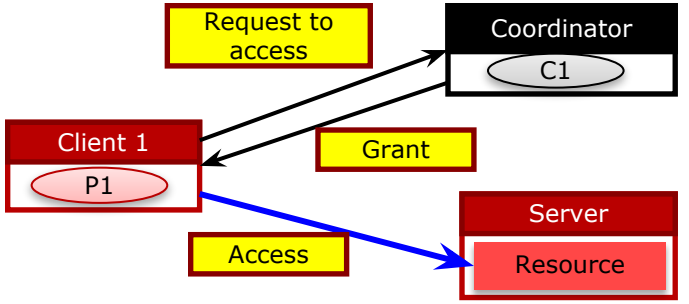


Types of Distributed Mutual Exclusion

Mutual exclusion algorithms are classified into two categories

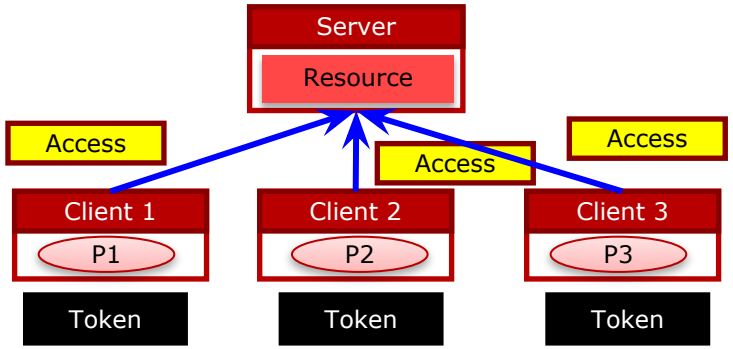
1. Permission-based Approaches

- A process, which wants to access a shared resource, requests the permission from one or more coordinators



2. Token-based Approaches

- Each shared resource has a token
- Token is circulated among all the processes
- A process can access the resource if it has the token



Types of Distributed Mutual Exclusion

- There are two types of permission-based (non- token) mutual exclusion algorithms
  a. Centralized Algorithms
  b. Decentralized Algorithms
     1. Lamport's algorithm
     2. Ricart-Agrawala algorithm
     3. Maekawa's Algorithm

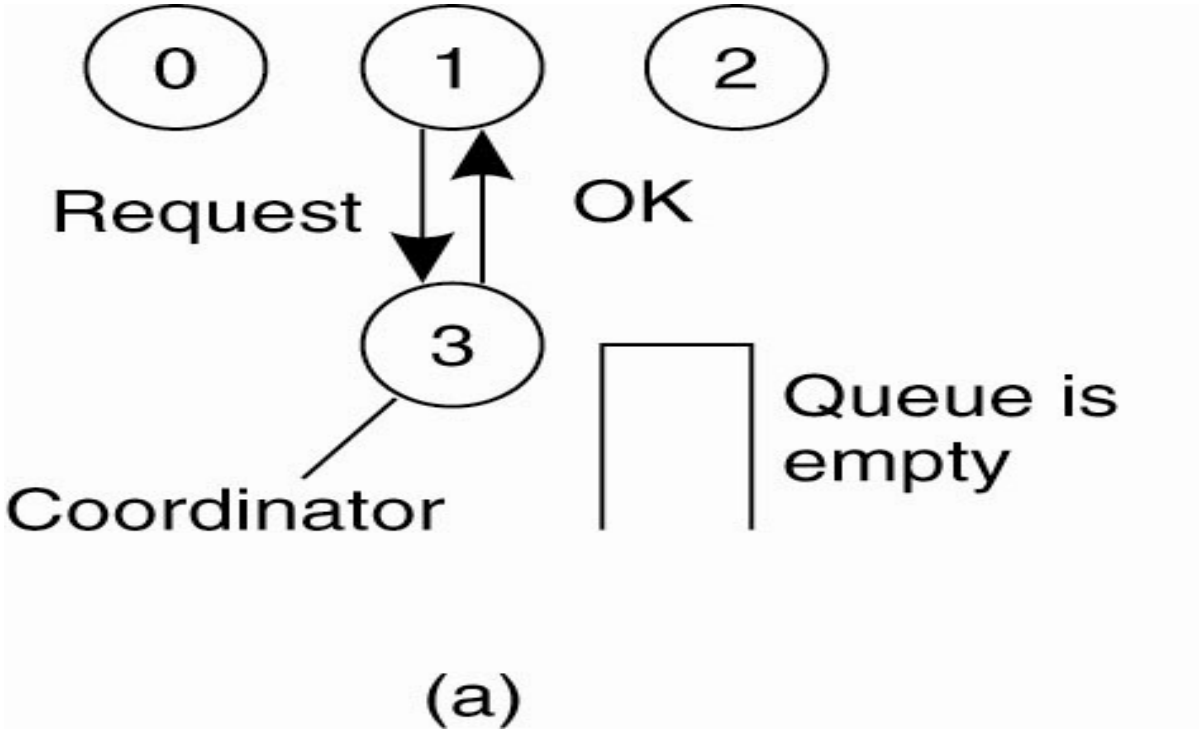
- There are three types of Token-based mutual exclusion algorithm
  1. Suzuki-Kasami's Broardcast Algorithms,
  2. Raymond's Tree based Algorithm

DS Mutual(Joint) Exclusion(Ruling out) Techniques

- **Centralized**: a single coordinator(process) controls whether a process can enter a critical region.

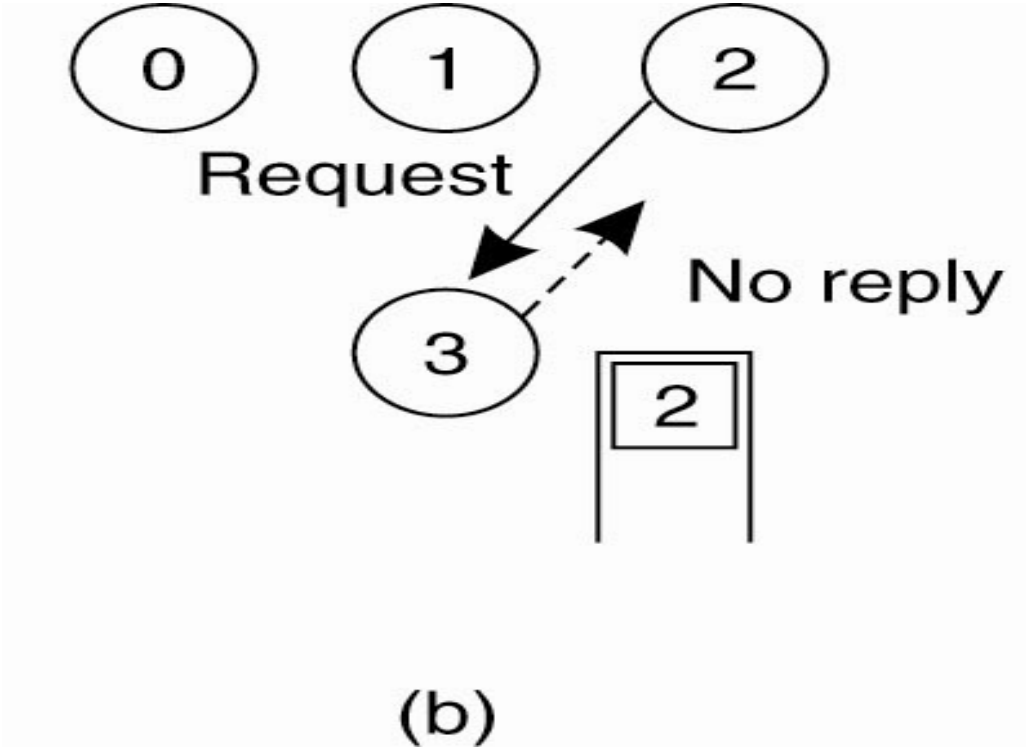
- **Distributed**: the group ,determine whether or not it is safe for a process to enter a critical section.

Mutual Exclusion: A Centralized Algorithm



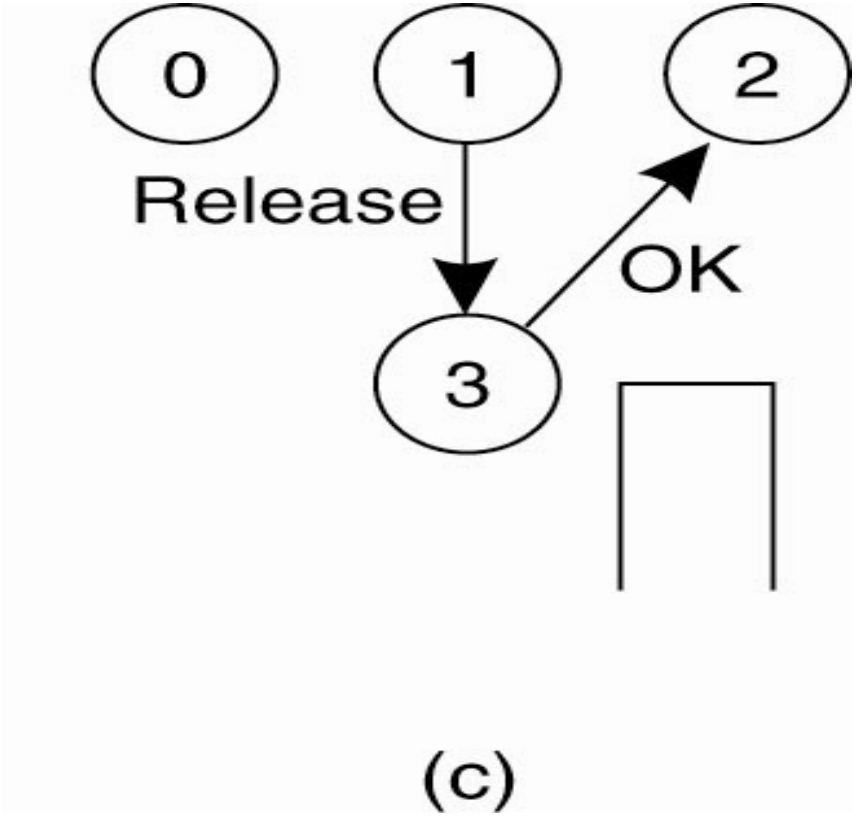
A) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion: A Centralized Algorithm



b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

Mutual Exclusion: A Centralized Algorithm



(c) When process 1 releases the resource, it tells the coordinator, which then replies to 2

Comments: The Centralized Algorithm

- ***Advantages***:
  - It works.
  - It is fair.
  - There's no process starvation.
  - Easy to implement.
- ***Disadvantages***:
  - There's a single point of failure!
  - The coordinator is a bottleneck on busy systems.

Non Token based Distributed Algorithms

- Distributed Algorithms the group ,determine whether or not it is safe for a process to enter a critical section.

- Non Token based Distributed Algorithms for mutual exclusion
  1. Lamport's algorithm
  2. Ricart-Agrawala algorithm
  3. Maekawa's Algorithm

Lamport's Algorithm

- Notations:
  - Si: Site I
  - Ri: Request set, containing the ids of all Sis from which permission must be received before accessing CS.
  - Non-token based approaches use time stamps to order requests for CS.
  - Smaller time stamps get priority over larger ones.
- Lamport's Algorithm
  - Ri = {S1, S2, …, Sn}, i.e., all sites.
  - Request queue: maintained at each Si. Ordered by time stamps.
  - Assumption: message delivered in FIFO.

Lamport's Algorithm

- Requesting CS:
  - Send REQUEST(tsi, i). (tsi,i): Request time stamp. Place REQUEST in *request_queuei*.
  - On receiving the message; sj sends time-stamped REPLY message to si. Si's request placed in *request_queuej*.
- Executing CS:
  - Si has received a message with time stamp larger than (tsi,i) from all other sites.
  - Si's request is the top most one in *request_queuei*.
- Releasing CS:
  - Exiting CS: send a time stamped RELEASE message to all sites in its request set.
  - Receiving RELEASE message: Sj removes Si's request from its queue.

Lamport's Algorithm Steps

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends a REQUEST(ts_i, i) message to all the sites in its request set R_i and places the request on $request\_queue_i$. ((ts_i, i) is the timestamp of the request.)
2. When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it returns a timestamped REPLY message to S_i and places site S_i's request on $request\_queue_j$.

Executing the critical section. Site S_i enters the CS when the two following 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 $request\_queue_i$.

Lamport's Algorithm Stpes

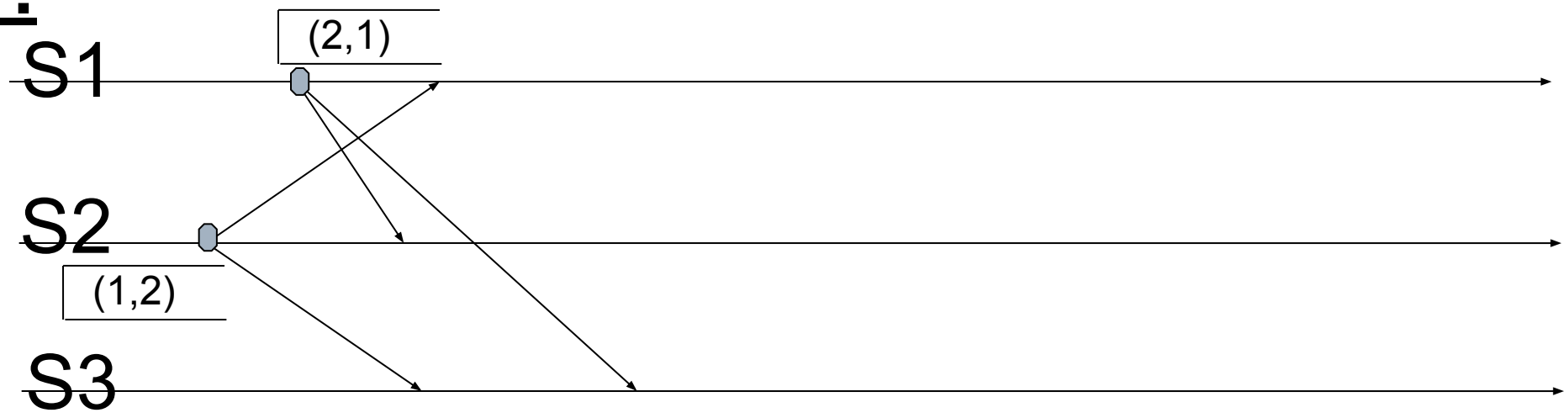
Releasing the critical section.

3. Site S_i, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
4. When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

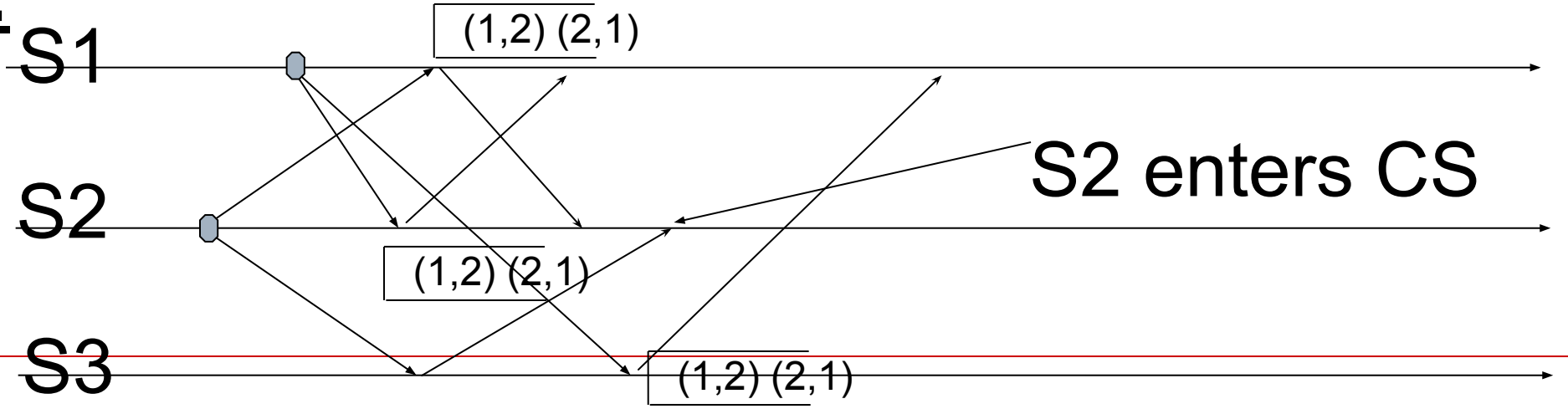
When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. The algorithm executes CS requests in the increasing order of timestamps.

Lamport's Algorithm: Example

Step 1:

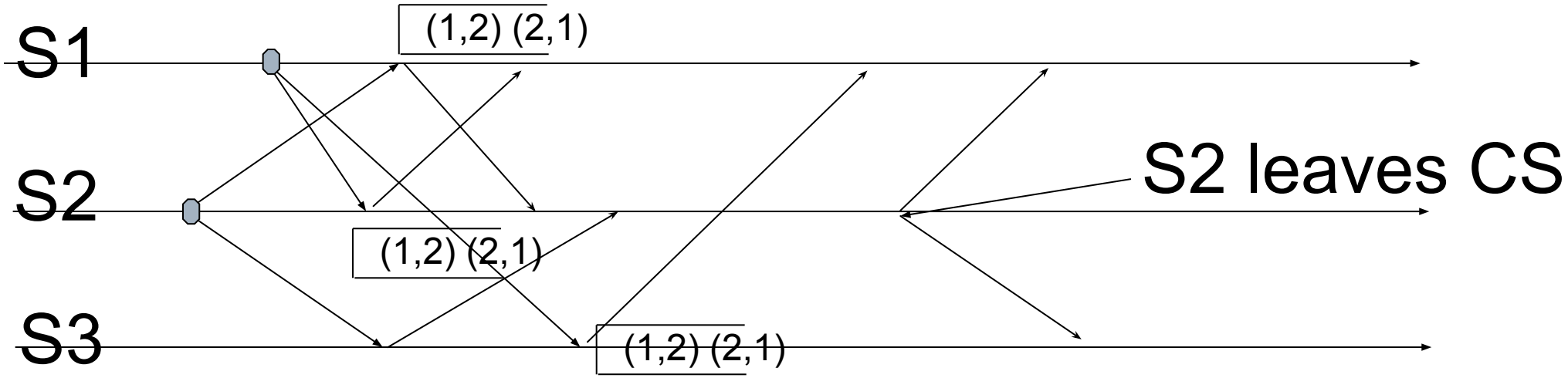


Step 2:

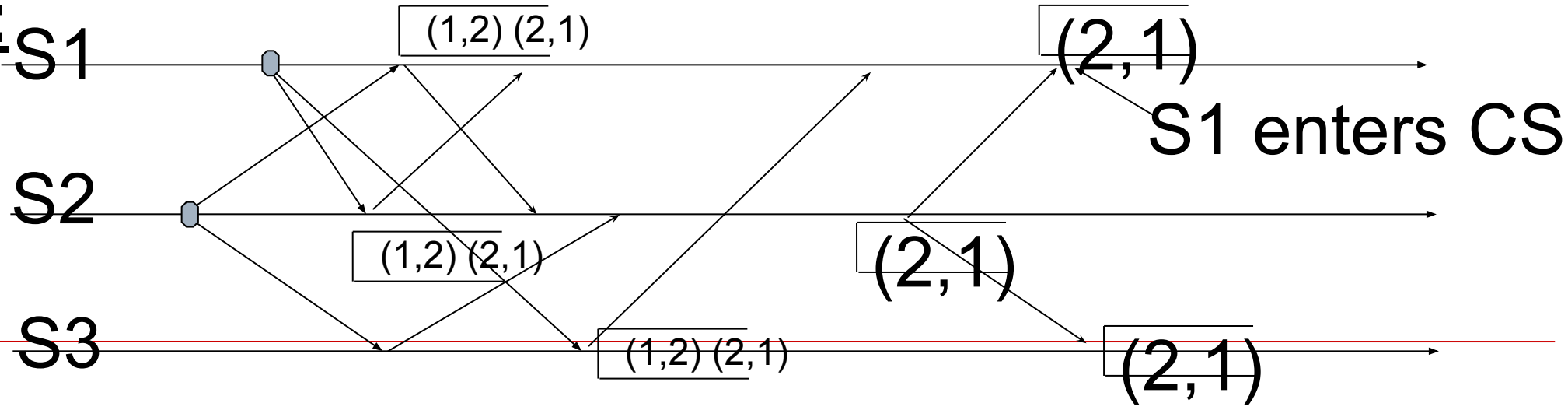


Lamport's Algorithm: Example

Step 3:



Step 4:



Lamport's Algorithm

Performance analysis

- Number of messages per CS
- 3(N-1) messages per CS invocation. (N - 1) REQUEST, (N - 1) REPLY, (N - 1) RELEASE messages.
- Synchronization delay :T
- Throughput
- Response time

Ricart-Agrawala Algorithm

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
2. 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 site S_i is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. The request is deferred otherwise.

Executing the critical section

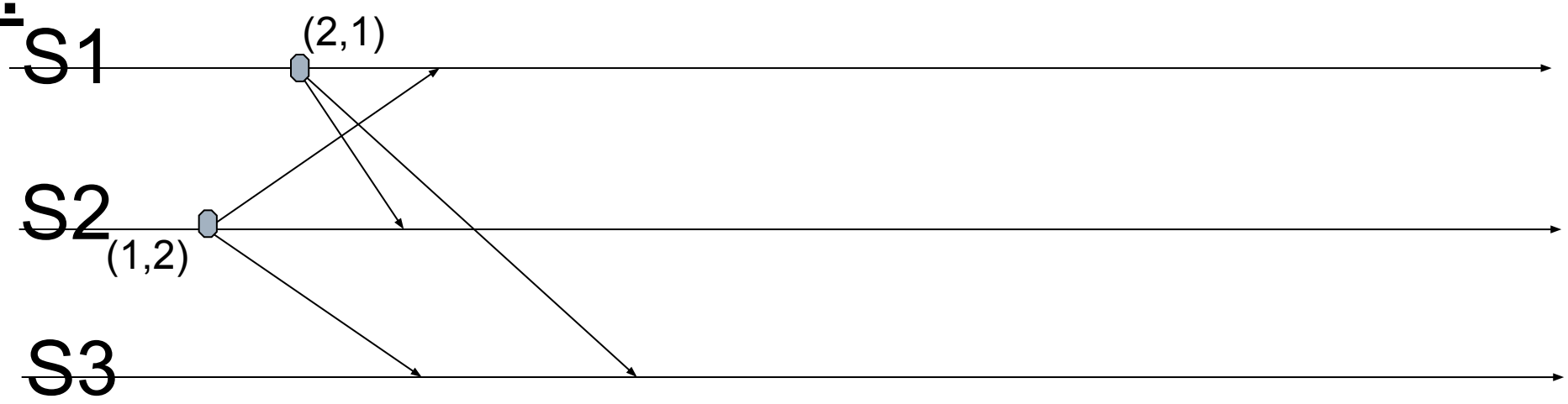
3. Site S_i enters the CS after it has received REPLY messages from all the sites in its request set.

Releasing the critical section

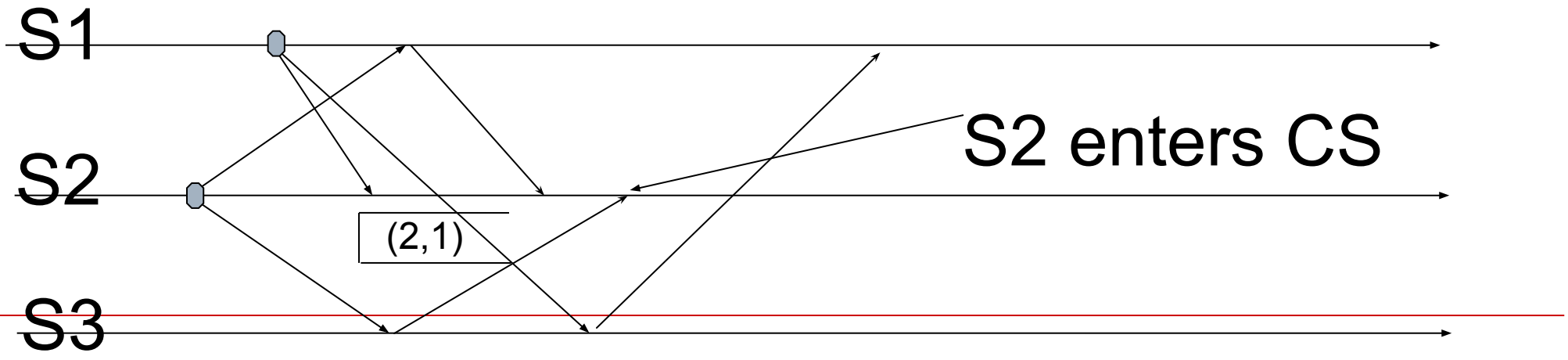
4. When site S_i exits the CS, it sends REPLY messages to all the deferred requests.

Ricart-Agrawala Algorithm : Example

Step 1:

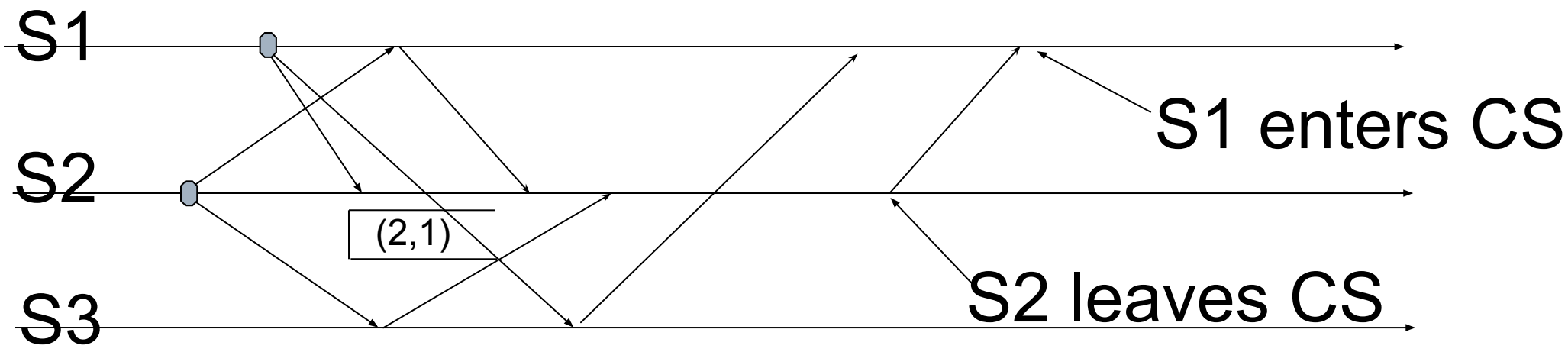


Step 2:



Ricart-Agrawala Algorithm : ExampleRicart-Agrawala Algorithm : Example

Step 3:



Ricart-Agrawala algorithm

Performance analysis

- Number of messages per CS: **2(N-1) messages per CS execution**. (N-1) REQUEST + (N-1) REPLY.
- Synchronization delay :T
- Throughput
- Response time

Comments: Ricart-Agrawala algorithm

- **Advantages**:
  - It works.
  - There is no single point of failure.

- **Disadvantages**:
  - All processes must maintain a list of the current processes in the group (and this can be tricky)
  - One overworked process in the system can become a *bottleneck* to the entire system – so, everyone slows down.

Maekawa's algorithm

- Maekawa's Algorithm is **quorum based approach** to ensure mutual exclusion in distributed systems. 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**.
- Three type of messages (REQUEST, REPLY and RELEASE) are used.
- A site send a REQUEST message to all other site in its request set or quorum to get their permission to enter critical section.
- A site send a REPLY message to requesting site to give its permission to enter the critical section.
- A site send a RELEASE message to all other site in its request set or quorum upon exiting the critical section

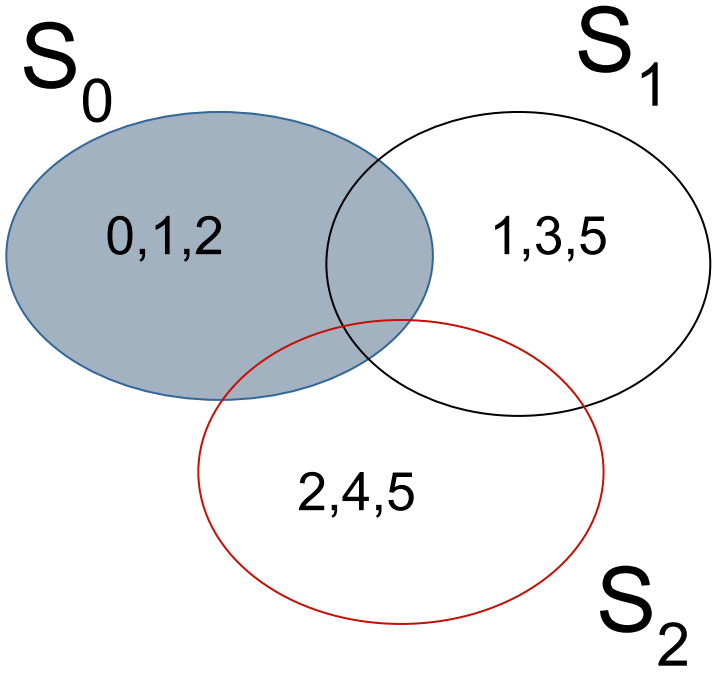
Maekawa's algorithm

- With each process i, associate a subset S_i. Divide the set of processes into subsets that satisfy the following two conditions:

  $\mathbf{i} \in S_i$

  $\forall i,j : 0 \leq i,j \leq n-1 \mid S_i \cap S_j \neq \varnothing$

- **Main idea**. Each process i is required to receive permission from S_i only. Correctness requires that multiple processes will never receive permission from all members of their respective subsets.



Maekawa's algorithm

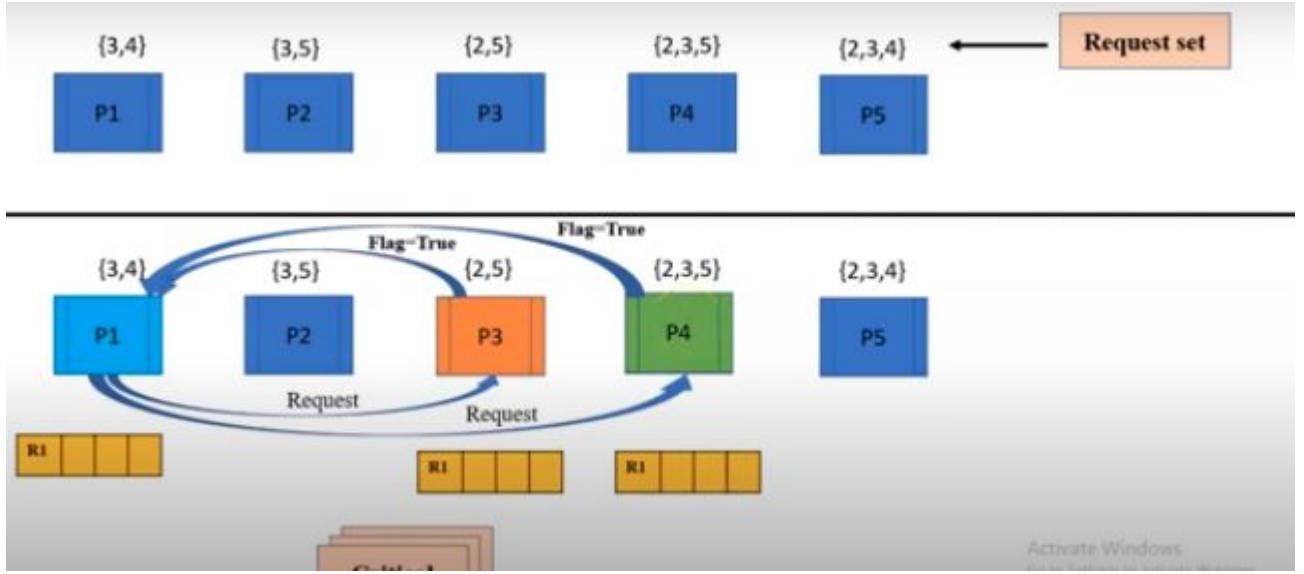
Algorithm:

- **To enter Critical section:**
  - When a site S_i wants to enter the critical section, it sends a request message **REQUEST(i)** to all other sites in the request set $\mathbf{R_i}$.
  - When a site S_j receives the request message **REQUEST(i)** from site S_i, it returns a **REPLY** message to site S_i if it has not sent a **REPLY** message to the site from the time it received the last **RELEASE** message. Otherwise, it queues up the request.
- **To execute the critical section:**
  - A site S_i can enter the critical section if it has received the **REPLY** message from all the site in request set $\mathbf{R_i}$
- **To release the critical section:**
  - When a site S_i exits the critical section, it sends **RELEASE(i)** message to all other sites in request set $\mathbf{R_i}$
  - When a site S_j receives the **RELEASE(i)** message from site S_i, it send **REPLY** message to the next site waiting in the queue and deletes that entry from the queue
  - In case queue is empty, site S_j update its status to show that it has not sent any **REPLY** message since the receipt of the last **RELEASE** message

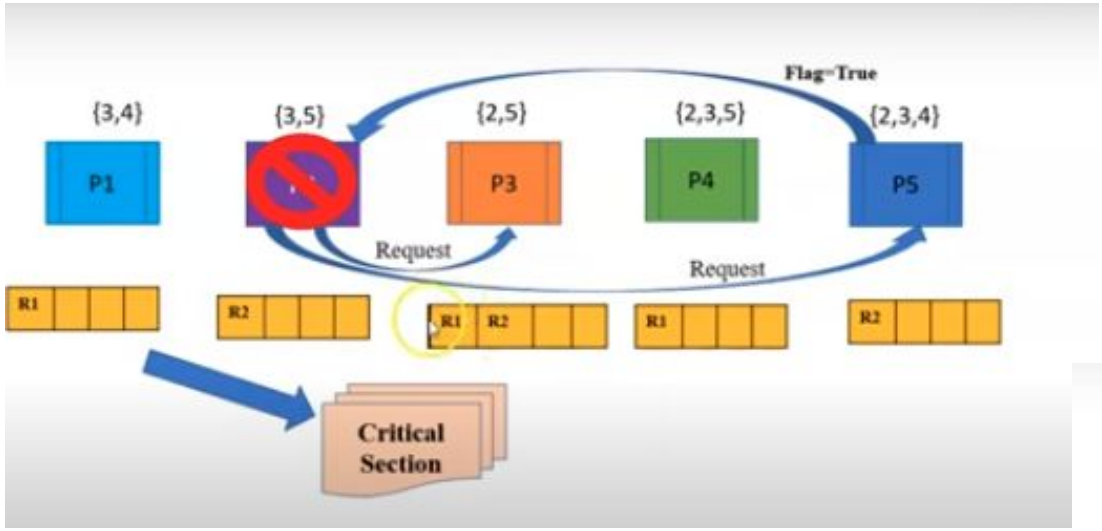
Maekawa's algorithm

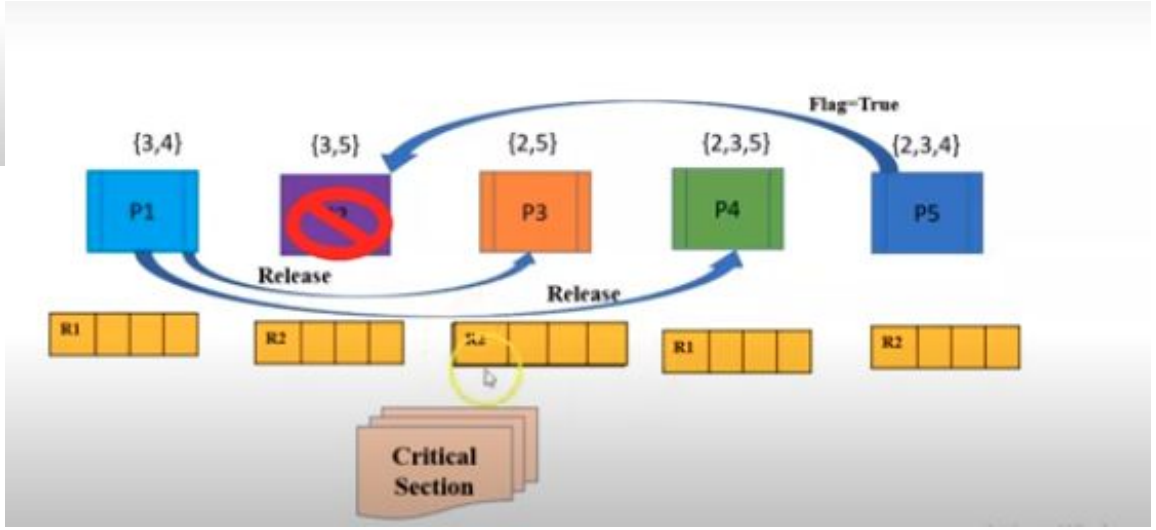
Example. Let there be Five processes 1, 2, 3, 4, 5

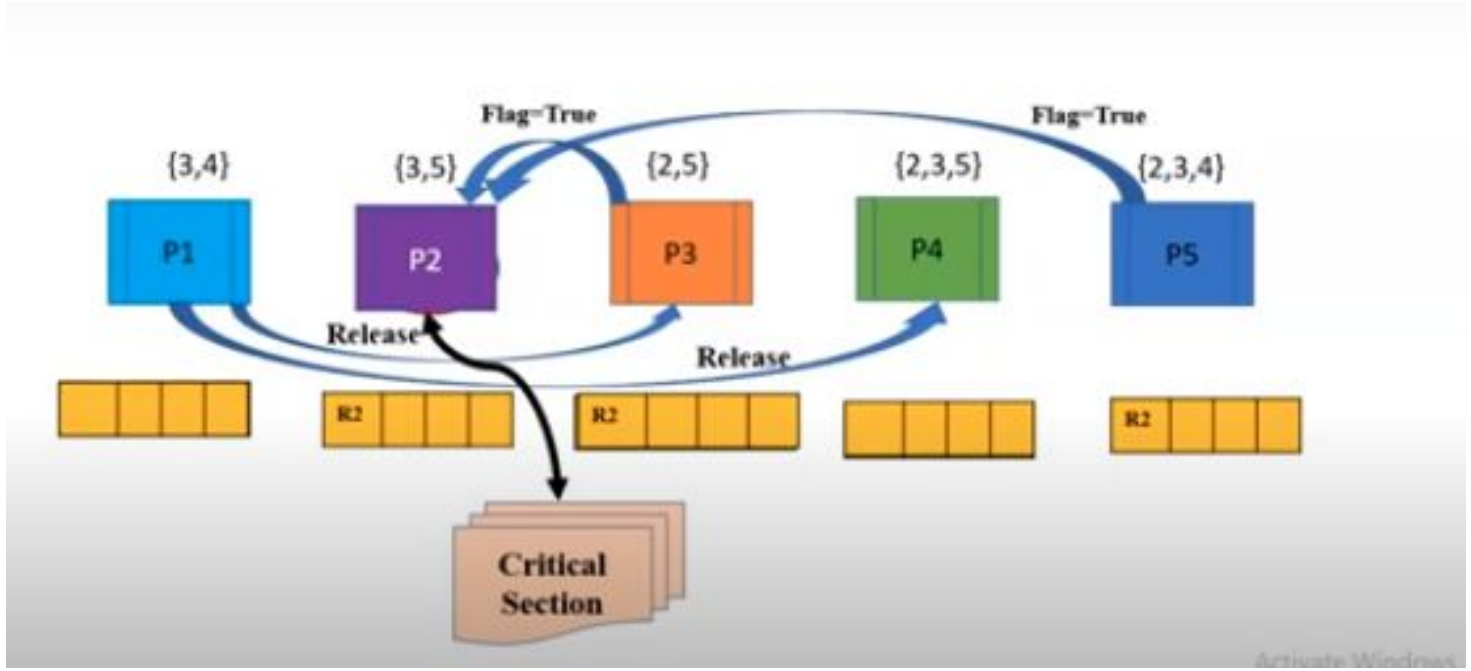
P1= {3,4} P2= {3,5} P3 = {2,5} P4 ={2,3,5} P5= { 2,3,4}



Maekawa's algorithm





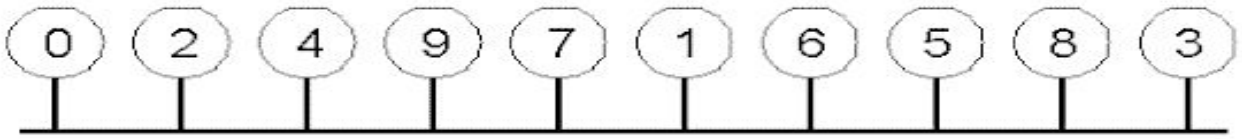

Flag=True
Flag=True
{3,4}
{3,5}
{2,5}
{2,3,5}
{2,3,4}
P1
P2
P3
P4
P5
Release
Release
R2
R2
R2
Critical Section

Token-based mutual exclusion algorithm

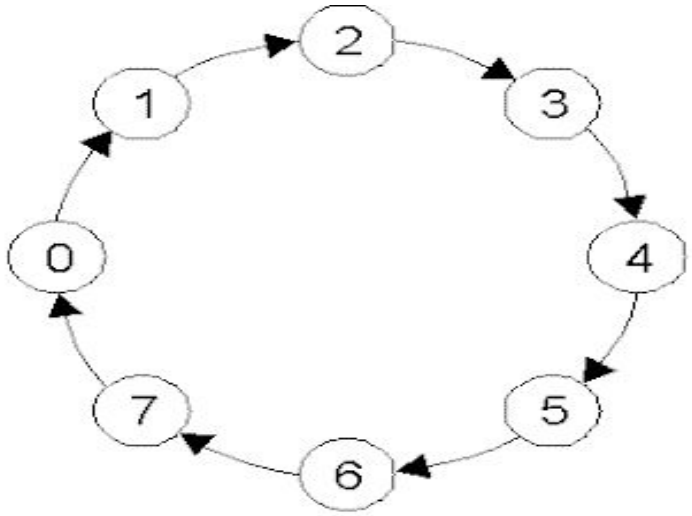
Types of Token-based mutual exclusion algorithm

1. Suzuki-Kasami‘s Broardcast Algorithms,
2. Raymond‘s Tree based Algorithm

Token Ring Algorithm



(a)



(b)

a) An unordered group of processes on a network.

b) A logical ring constructed in software.

Token Ring Algorithm

- When the ring is initialized, one process is given the token
- Unique token circulates among the participating sites.
- A site can enter CS if it has the token
- Token-based approaches use sequence numbers instead of time stamps.
  - Request for a token contains a sequence number.
  - Sequence number of sites advance independently.
- Correctness issue is unimportant since only one token is present -> only one site can enter CS.
- Deadlock and starvation issues to be addressed.

Pros & Cons of Token Ring alorithim

- Token ring approach provides deterministic mutual exclusion
  - There is one token, and the resource cannot be accessed without a token
- Token ring approach avoids starvation
  - Each process will receive the token
- Token ring has a high-message overhead
  - When no processes need the resource, the token circulates at a high-speed
- If the token is lost, it must be regenerated
  - Detecting the loss of token is difficult since the amount of time between successive appearances of the token is unbounded
- Dead processes must be purged from the ring
  - ACK based token delivery can assist in purging dead processes

Token Ring Algorithm

Performance analysis

- Number of messages per CS
- Synchronization delay
- Response time
- Problems
  - Lost token
  - Process crash

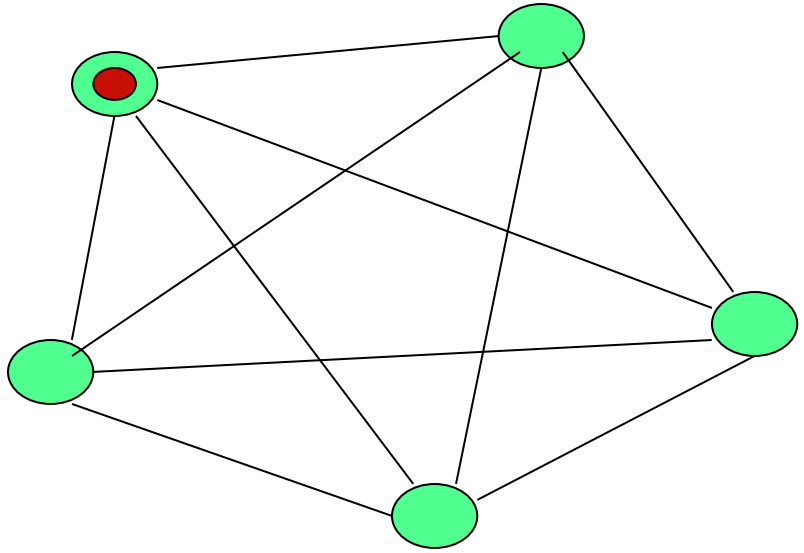
Suzuki-Kasami's Algorithm

The Main idea

Completely connected network of processes

There is **one token** in the network. The holder of the token has the permission to enter CS.

Any other process trying to enter CS must acquire that token. Thus the token will move from one process to another based on demand.



I want to enter CS

I want to enter CS

Suzuki-Kasami's Algorithm

Process i broadcasts **(i, num)**

Sequence number of the request

Each process maintains

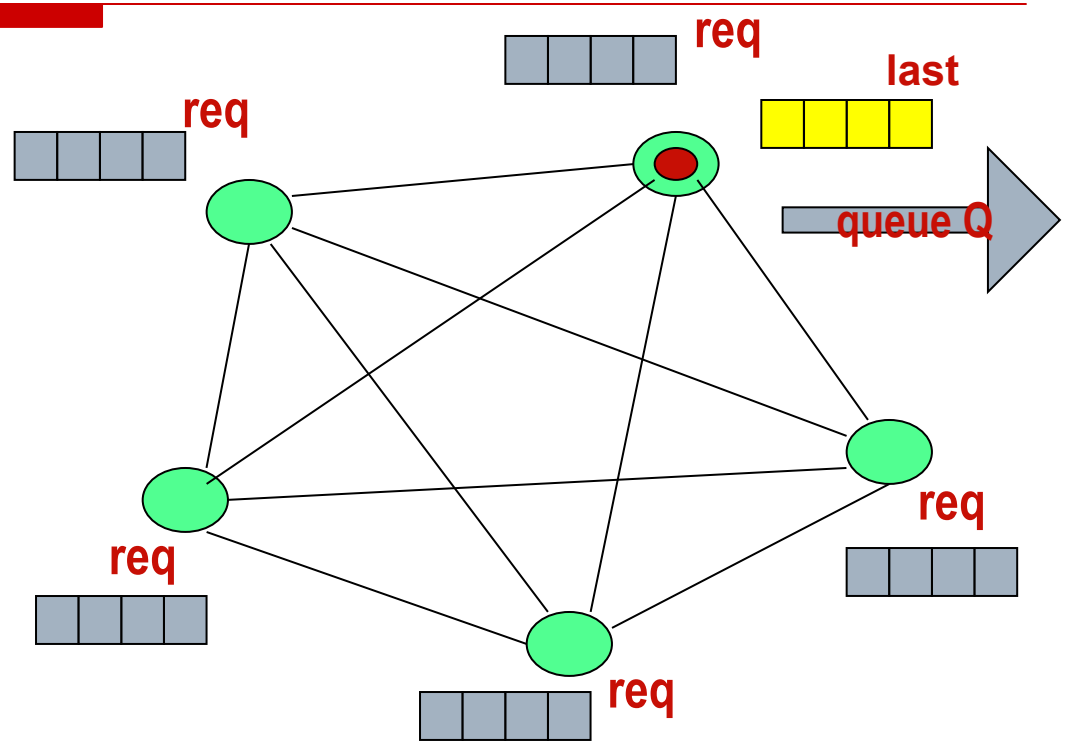
-an array **req**: **req[j]** denotes the sequence no of the ***latest request*** from process j

(Some requests will be stale soon)

Additionally, the holder of the token maintains

-an array **last**: **last[j]** denotes the sequence number of *the latest visit* to CS for process j.

- **a queue Q** of waiting processes



req: array[0..n-1] of integer

last: array [0..n-1] of integer

Suzuki-Kasami's Algorithm

Requesting the critical section

1. If the requesting site S_i does not have the token, then it increments its sequence number, RN_i[i], and sends a REQUEST(i, sn) message to all other sites. (sn is the updated value of RN_i[i].)
2. When a site S_j receives this message, it sets RN_j[i] to max(RN_j[i], sn). If S_j has the idle token, then it sends the token to S_i if RN_j[i]=LN[i]+1.

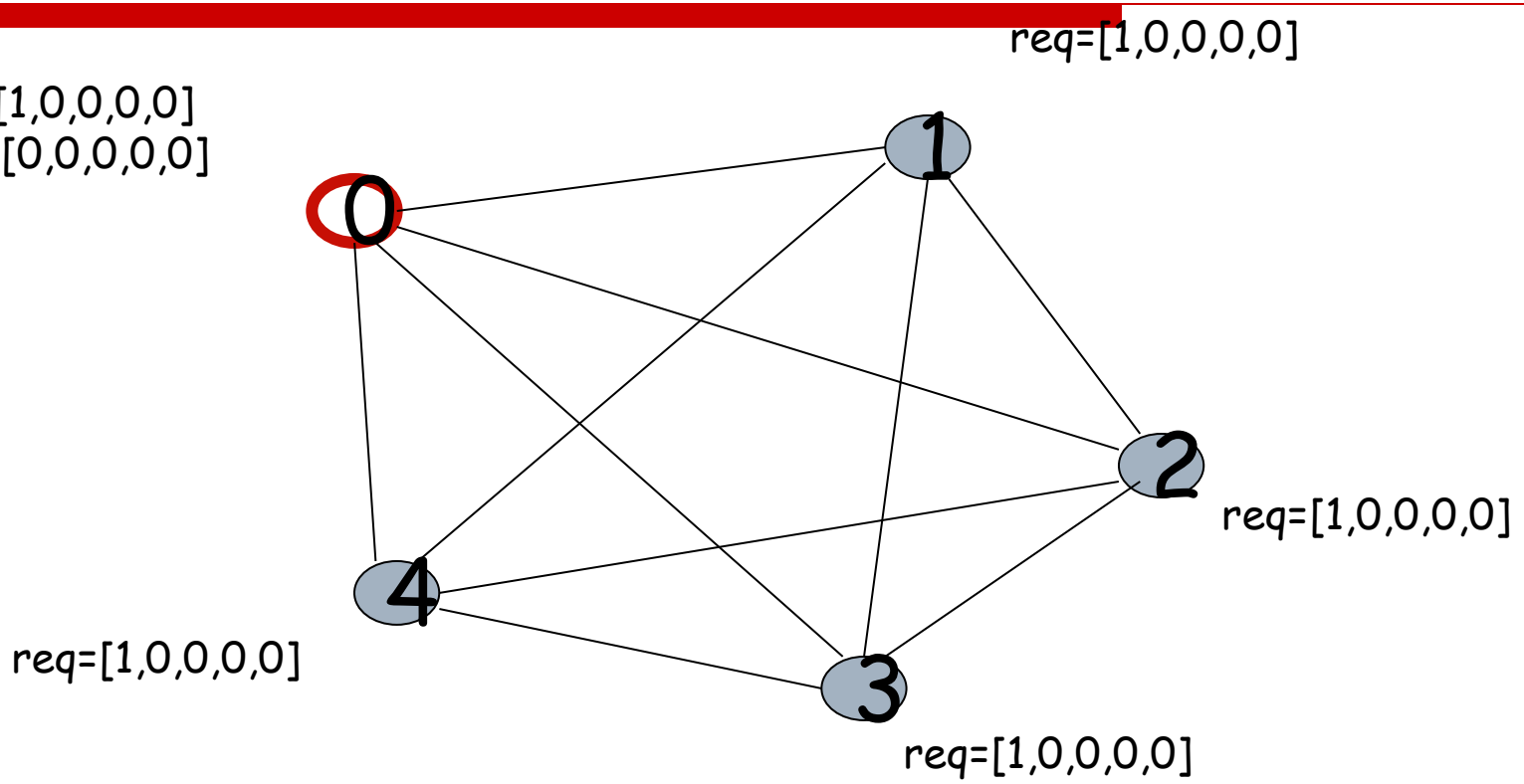
Executing the critical section.

3. Site S_i executes the CS when it has received the token.

Releasing the critical section. Having finished the execution of the CS, site S_i takes the following actions:

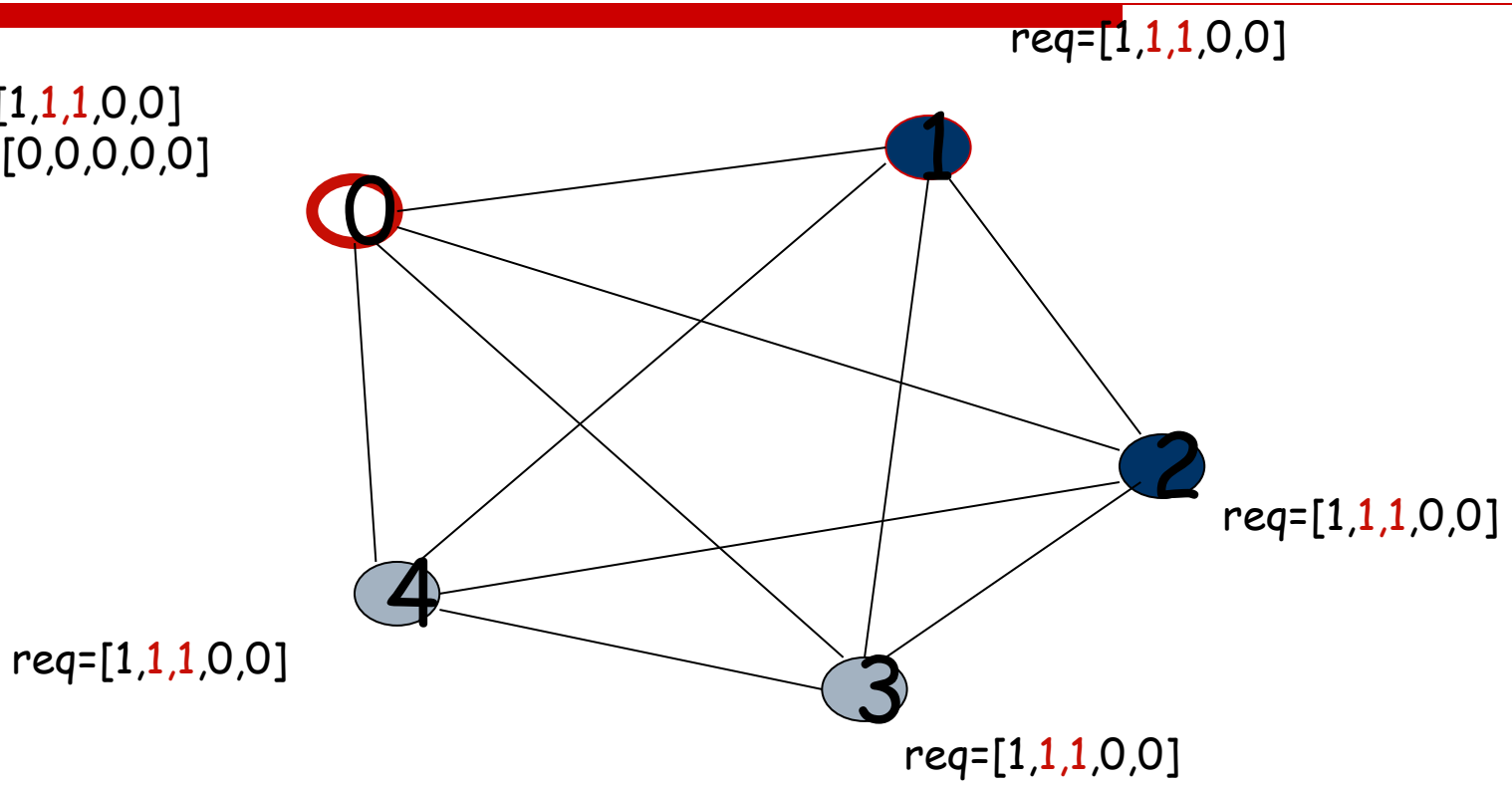
4. It sets LN[i] element of the token array equal to RN_i[i].
5. For every site S_j whose ID is not in the token queue, it appends its ID to the token queue if RN_i[j]=LN[j]+1.
6. If token queue is nonempty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Suzuki-Kasami's Algorithm : Example



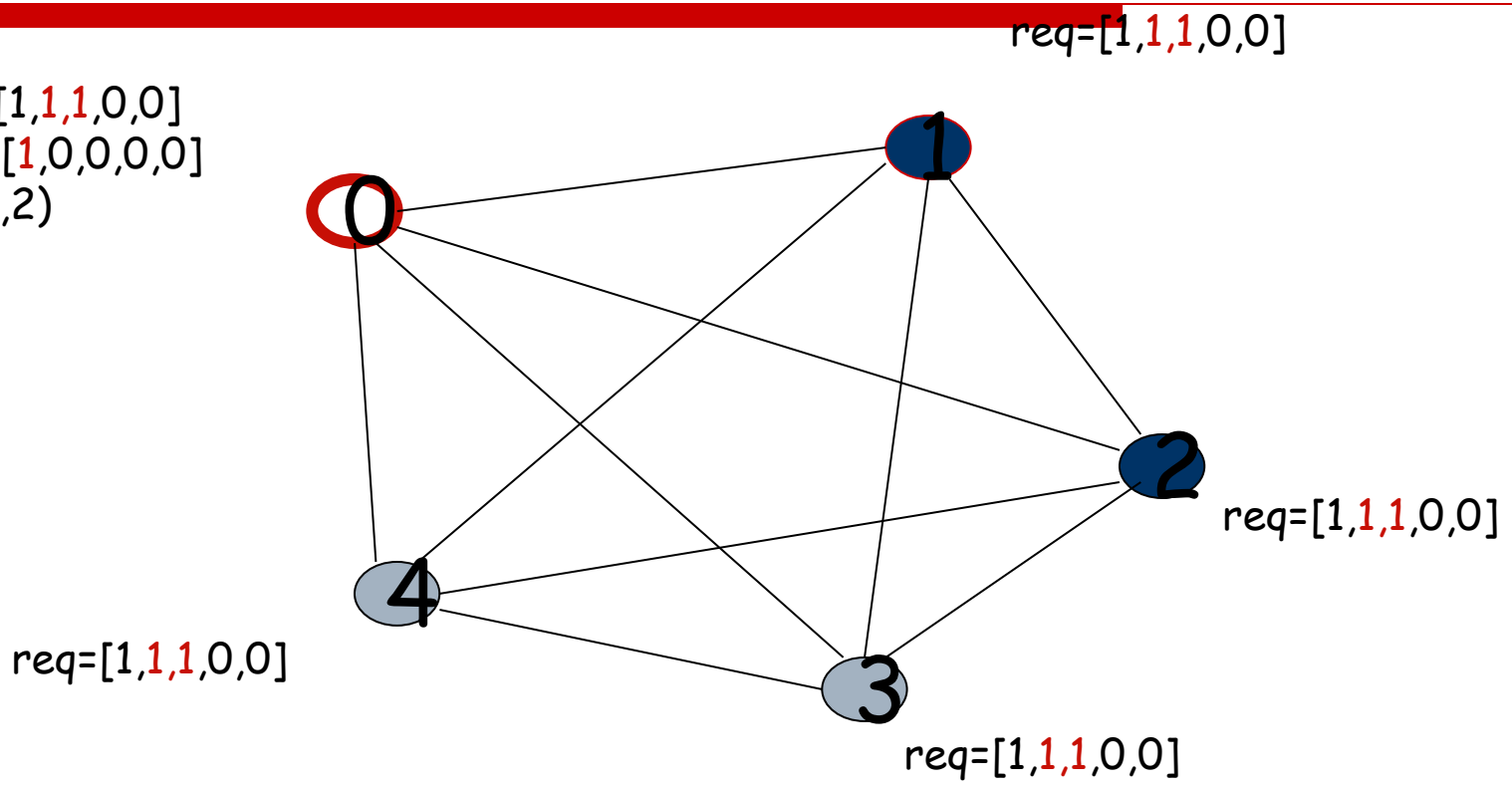
initial state: process 0 has sent a request to all, and grabbed the token

Suzuki-Kasami's Algorithm : Example



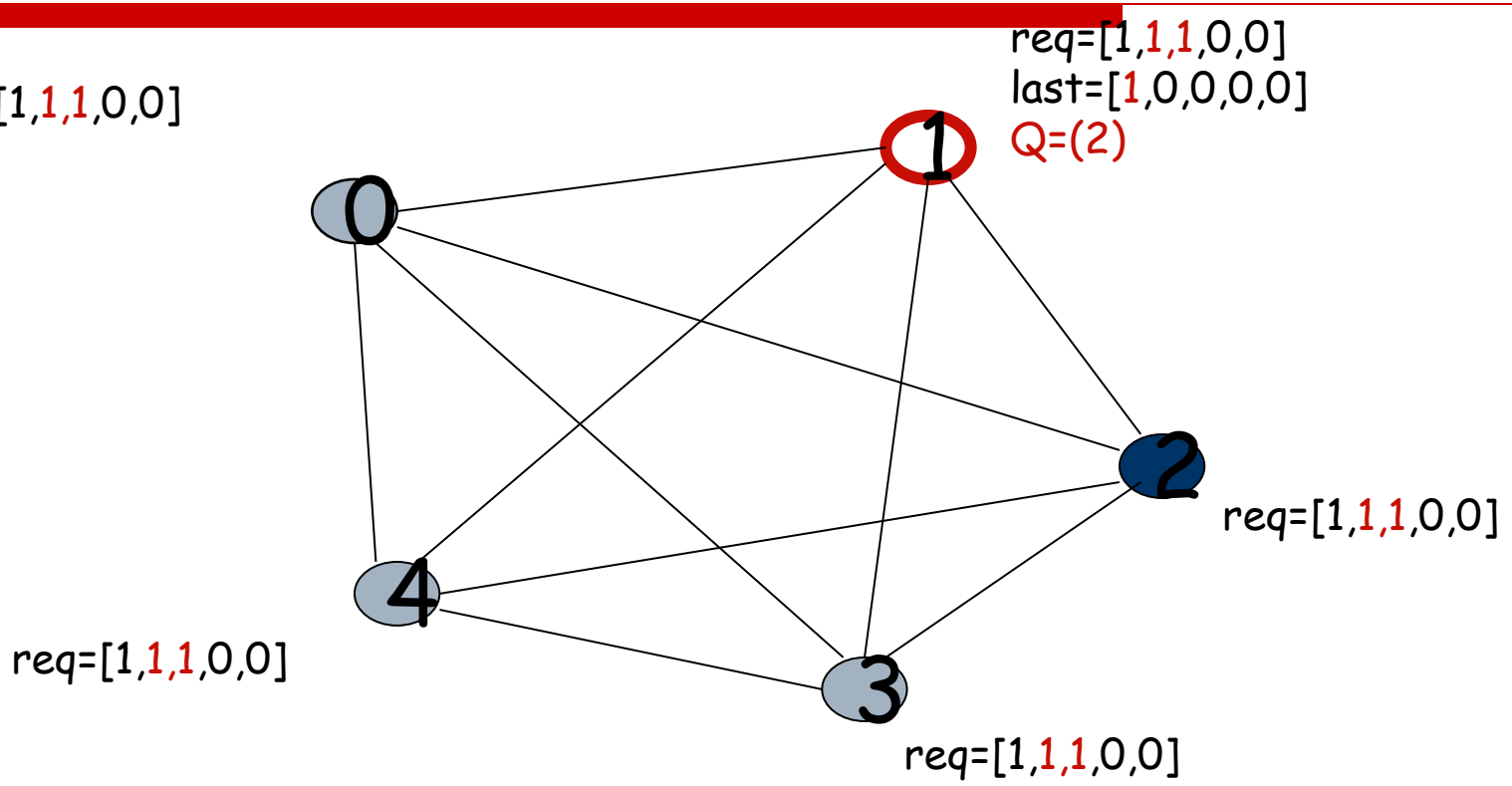
1 & 2 send requests to enter CS

Suzuki-Kasami's Algorithm : Example



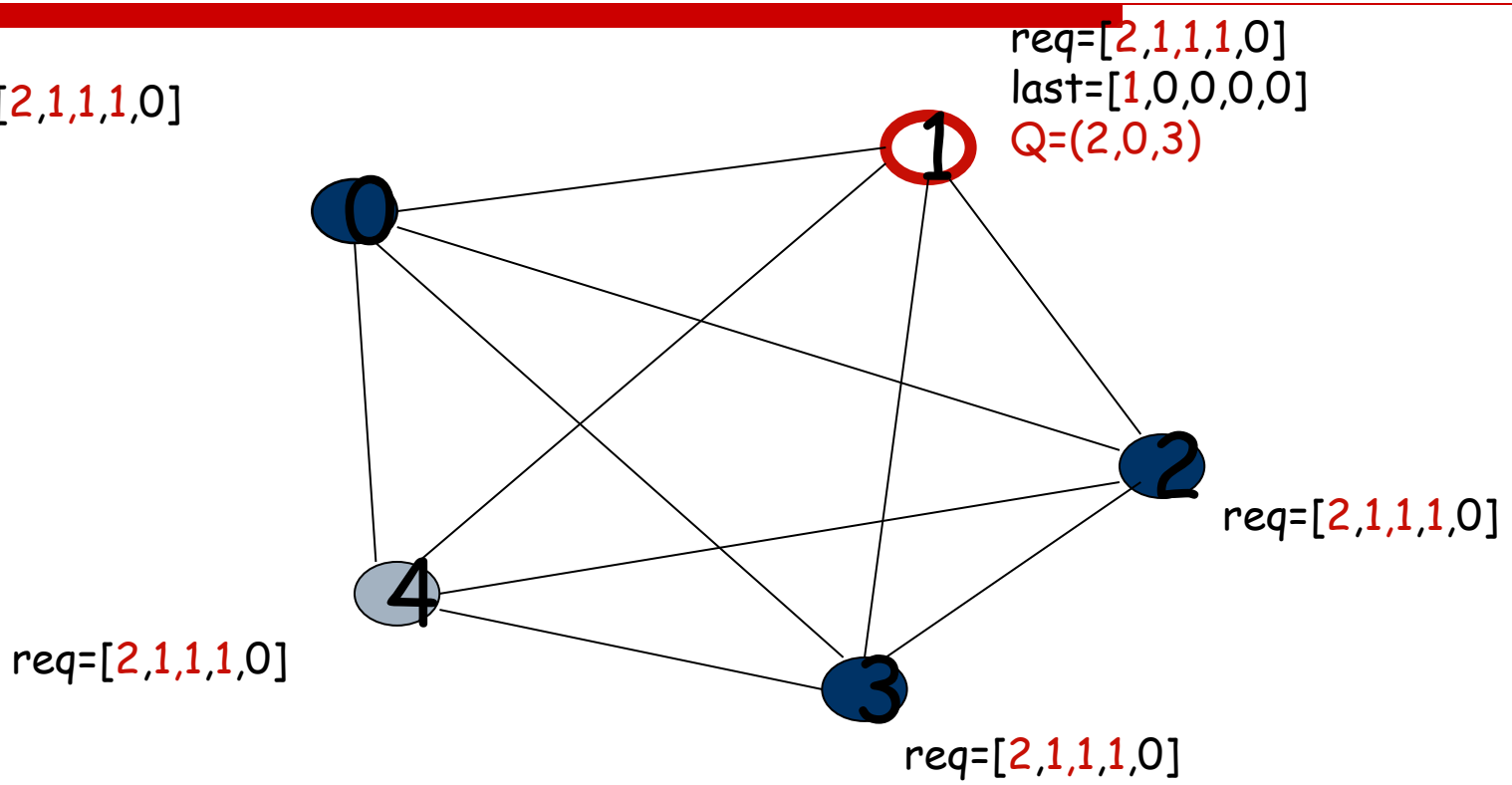
0 prepares to exit CS

Suzuki-Kasami's Algorithm : Example



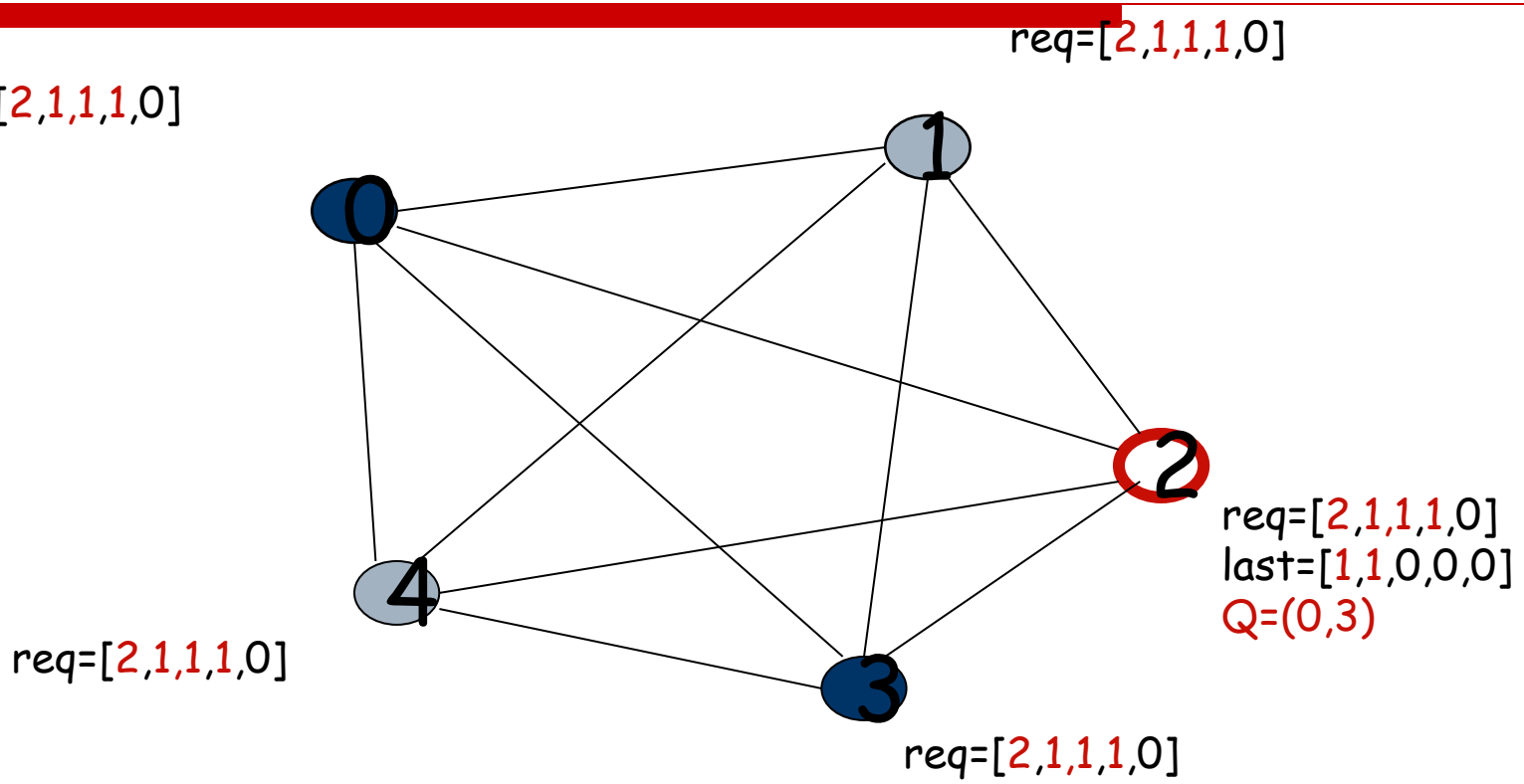
0 passes token (Q and last) to 1

Suzuki-Kasami's Algorithm : Example



0 and 3 send requests

Suzuki-Kasami's Algorithm : Example



1 sends token to 2

Raymond's Algorithm

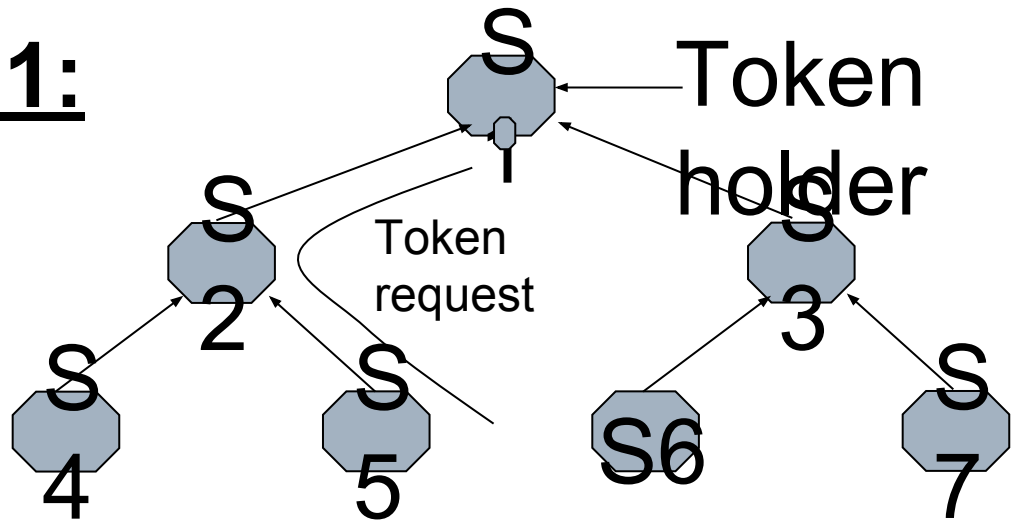
- Sites are arranged in a logical directed tree. Root: token holder. Edges: directed towards root.
- Every site has a variable *holder* that points to an immediate neighbor node, on the directed path towards root. (Root's holder point to itself).
- Requesting CS
  - If Si does not hold token and request CS, sends REQUEST *upwards* provided its *request_q* is empty. It then adds its request to *request_q*.
  - Non-empty *request_q* -> REQUEST message for top entry in q (if not done before).
  - Site on path to root receiving REQUEST -> propagate it up, if its *request_q* is empty. Add request to *request_q*.
  - Root on receiving REQUEST -> send token to the site that forwarded the message. Set *holder* to that forwarding site.
  - Any Si receiving token -> delete top entry from *request_q*, send token to that site, set *holder* to point to it. If request_q is non-empty now, send REQUEST message to the *holder* site.

Raymond's Algorithm …

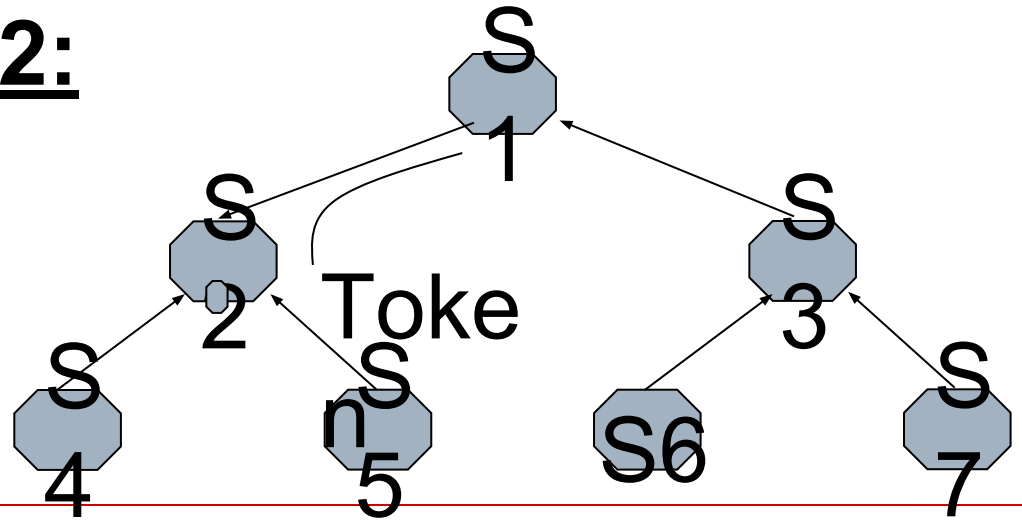
- Executing CS: getting token with the site at the top of *request_q*. Delete top of *request_q*, enter CS.
- Releasing CS
  - If *request_q* is non-empty, delete top entry from q, send token to that site, set *holder* to that site.
  - If *request_q* is non-empty now, send REQUEST message to the *holder* site.
- Performance
  - Average messages: O(log N) as average distance between 2 nodes in the tree is O(log N).
  - Synchronization delay: (T log N) / 2, as average distance between 2 sites to successively execute CS is (log N) / 2.
  - Greedy approach: Intermediate site getting the token may enter CS instead of forwarding it down. Affects fairness, may cause starvation.

Raymond's Algorithm: Example

Step 1:

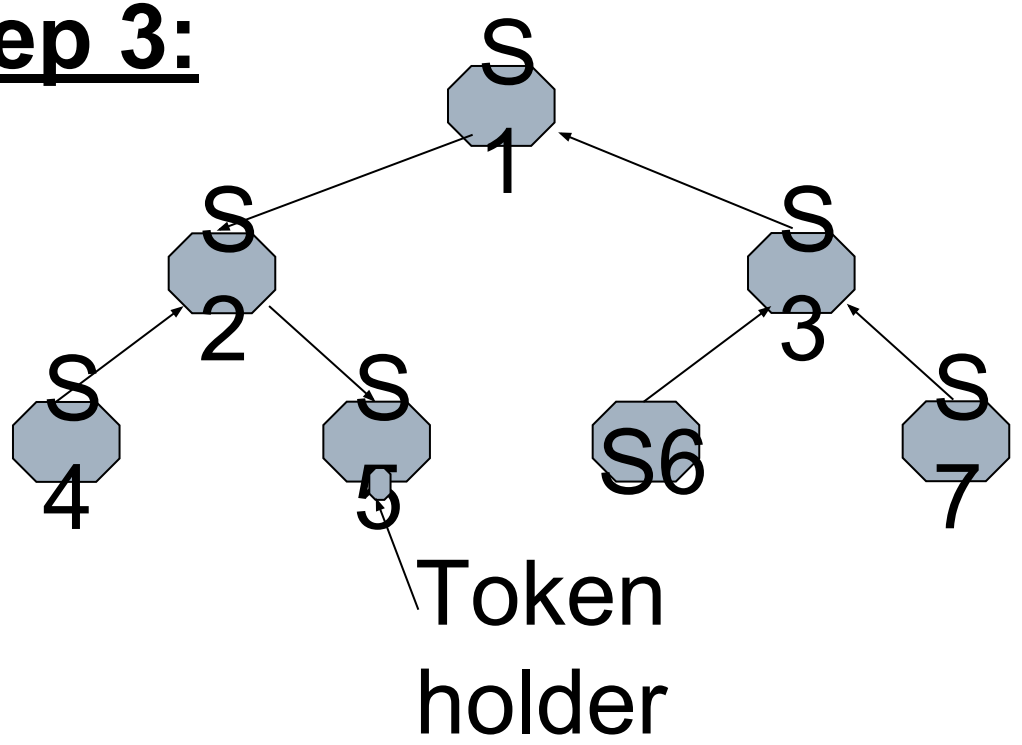


Step 2:



Raymond's Algorithm: Example

Step 3:



Comparison of Distributed Mutual Exclusion Algorithms

Non-Token	Resp. Time(ll)	Sync. Delay	Messages(ll)
Lamport	2T+E	T	3(N-1)
Ricart-Agrawala	2T+E	T	2(N-1)
Maekawa	2T+E	2T	3*sq.rt(N)

II – light load

Token	Resp. Time(ll)	Sync. Delay	Messages(ll)
Suzuki-Kasami	2T+E	T	N
Raymond	T(log N)+E	Tlog(N)/2	log(N)

Election Algorithms

- Introduction
- Traditional election algorithms
  1. Bully algorithm
  2. Ring algorithm

Introduction

Need for a Coordinator

- Many algorithms used in distributed systems require a coordinator
  - For example, see the centralized mutual exclusion algorithm.
- In general, all processes in the distributed system are equally suitable for the role
- Election algorithms are designed to choose a coordinator.

Election Algorithms

- Any process can serve as coordinator
- Any process can “call an election” (initiate the algorithm to choose a new coordinator).
  - There is no harm (other than extra message traffic) in having multiple concurrent elections.
- Elections may be needed when the system is initialized, or if the coordinator crashes or retires.

Election Algorithms

Assumptions

- Every process/site has a unique ID; e.g.
  - the network address
  - a process number
- Every process in the system should know the values in the set of ID numbers, although not which processors are up or down.
- The process with the highest ID number will be the new coordinator.
- Process groups (as with ISIS toolkit or MPI) satisfy these requirements.

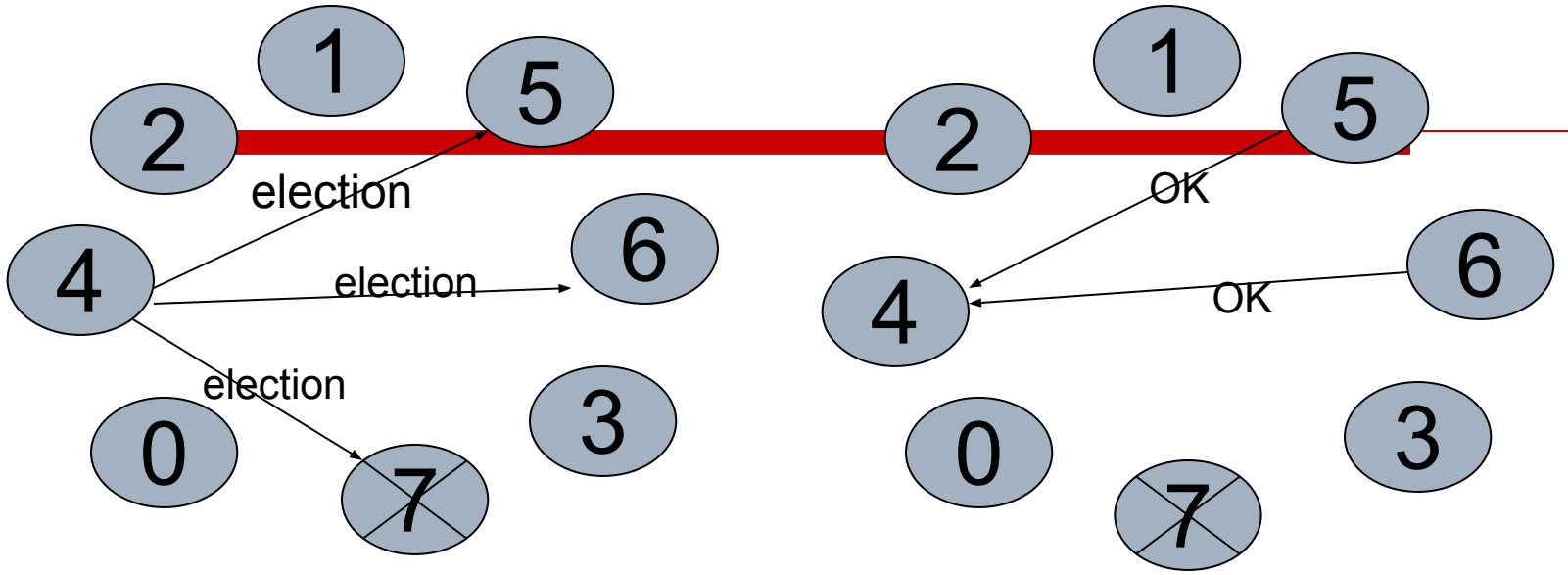
Election Algorithms

Requirements

- When the election algorithm terminates a single process has been selected and every process knows its identity.
- Formalize: every process p_i has a variable e_i to hold the coordinator's process number.
  - $\forall$ i, e_i = undefined or e_i = P, where P is the non-crashed process with highest id
  - All processes (that have not crashed) eventually set e_i = P.

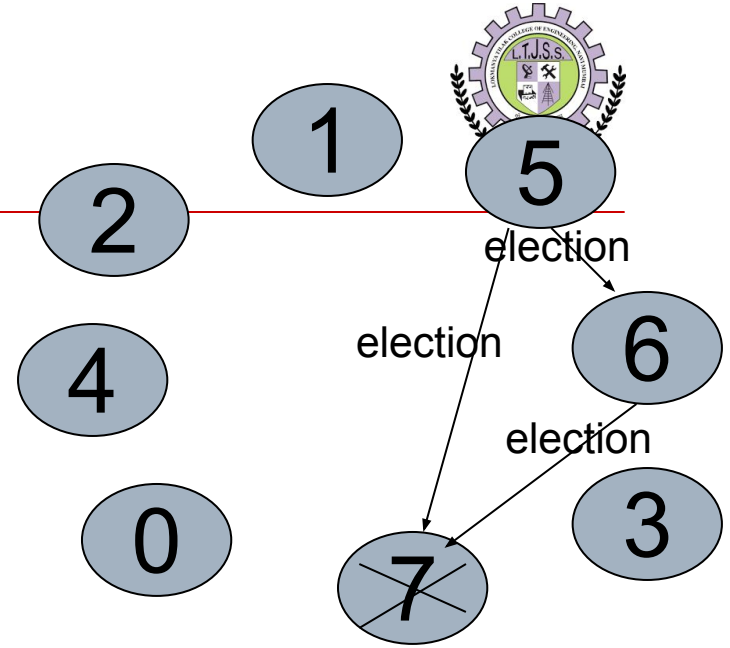
The Bully Algorithm - Overview

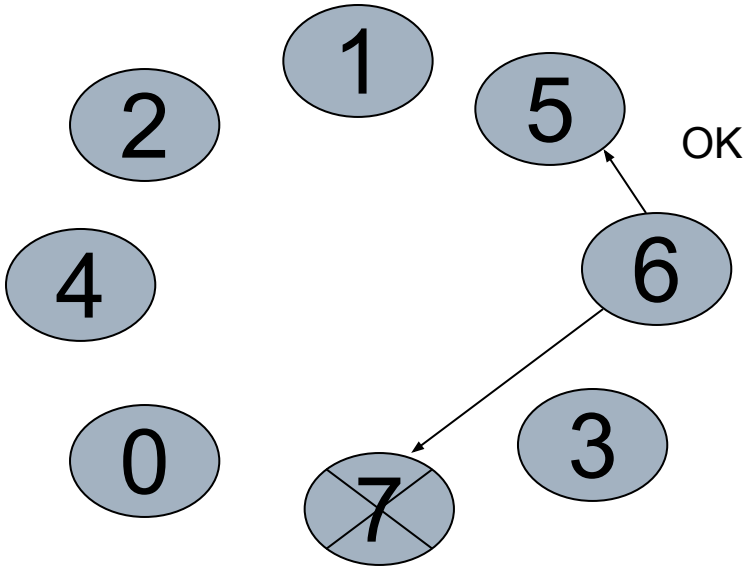
- Process *p* calls an election when it notices that the coordinator is no longer responding.
- High-numbered processes “bully” low-numbered processes out of the election, until only one process remains.
- When a crashed process reboots, it holds an election. If it is now the highest-numbered live process, it will win.

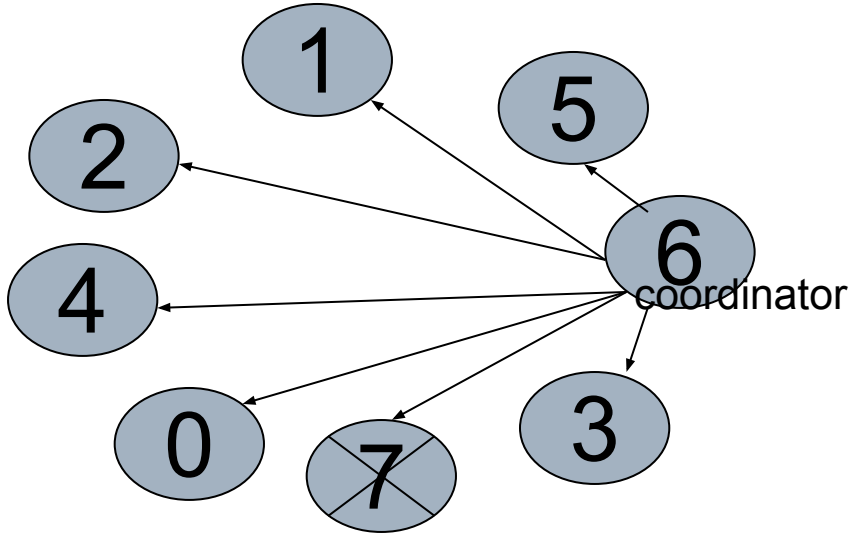


- Process *p* sends an election message to all *higher-numbered* processes in the system.
  If no process responds, then *p* becomes the coordinator.
  If a higher-level process (*q*) responds, it sends *p* a message that terminates *p*'s role in the algorithm

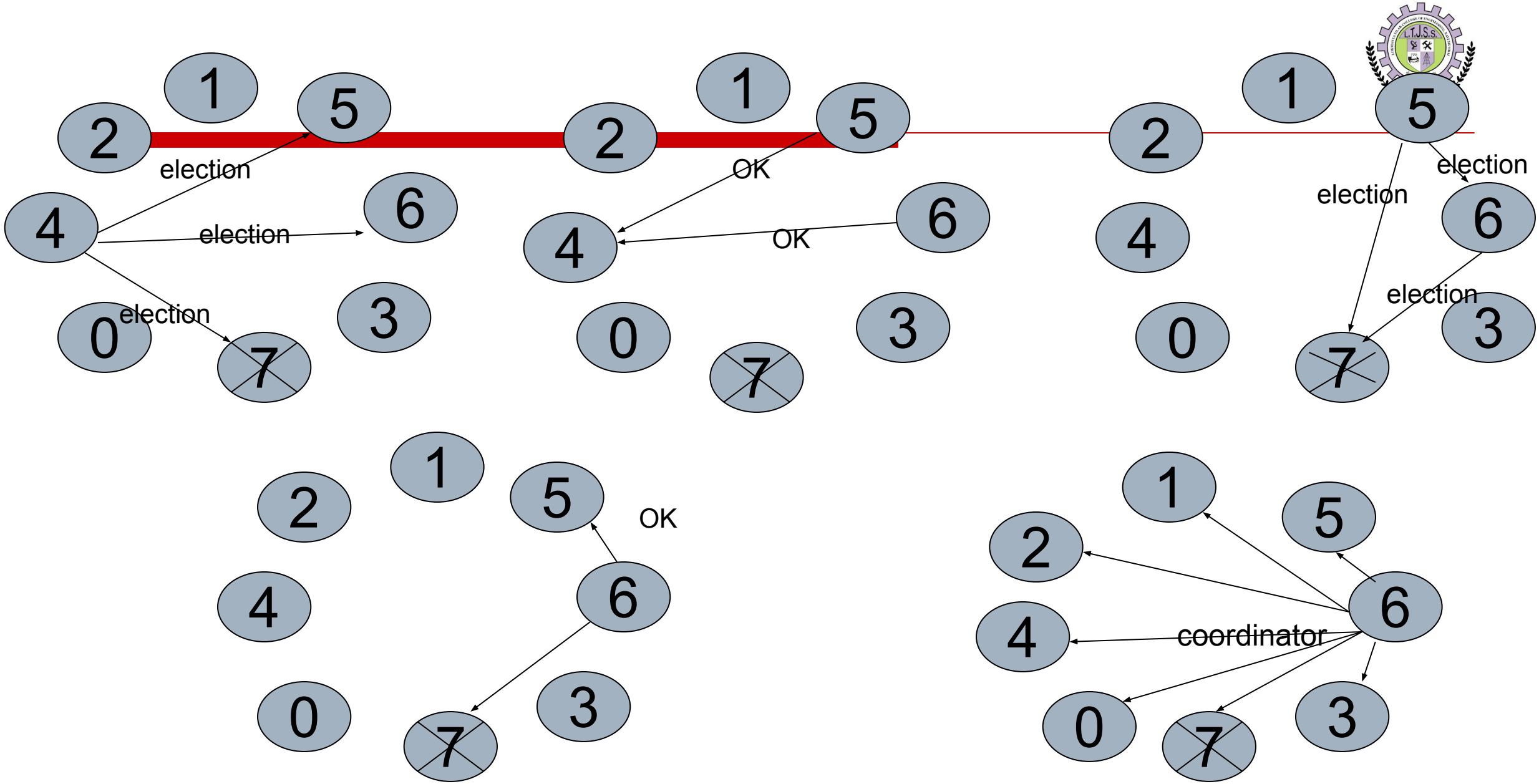
- The process *q* now calls an election (if it has not already done so).
- Repeat until no higher-level process responds. The last process to call an election "wins" the election.
- The winner sends a message to other processes announcing itself as the new coordinator.







If 7 comes back on line, it will call an election


1
2
5
election
4
election
6
election
0
3
7
1
2
5
OK
4
OK
6
0
3
7
1
2
5
election
election
4
6
election
0
3
7
1
2
5
OK
4
6
0
3
7
1
2
5
4
coordinator
6
0
3
7

Ring Algorithm - Overview

- The ring algorithm assumes that the processes are arranged in a logical ring and each process is knows the order of the ring of processes.
- Processes are able to “skip” faulty systems: instead of sending to process j, send to j + 1.
- Faulty systems are those that don’t respond in a fixed amount of time.

Ring Algorithm

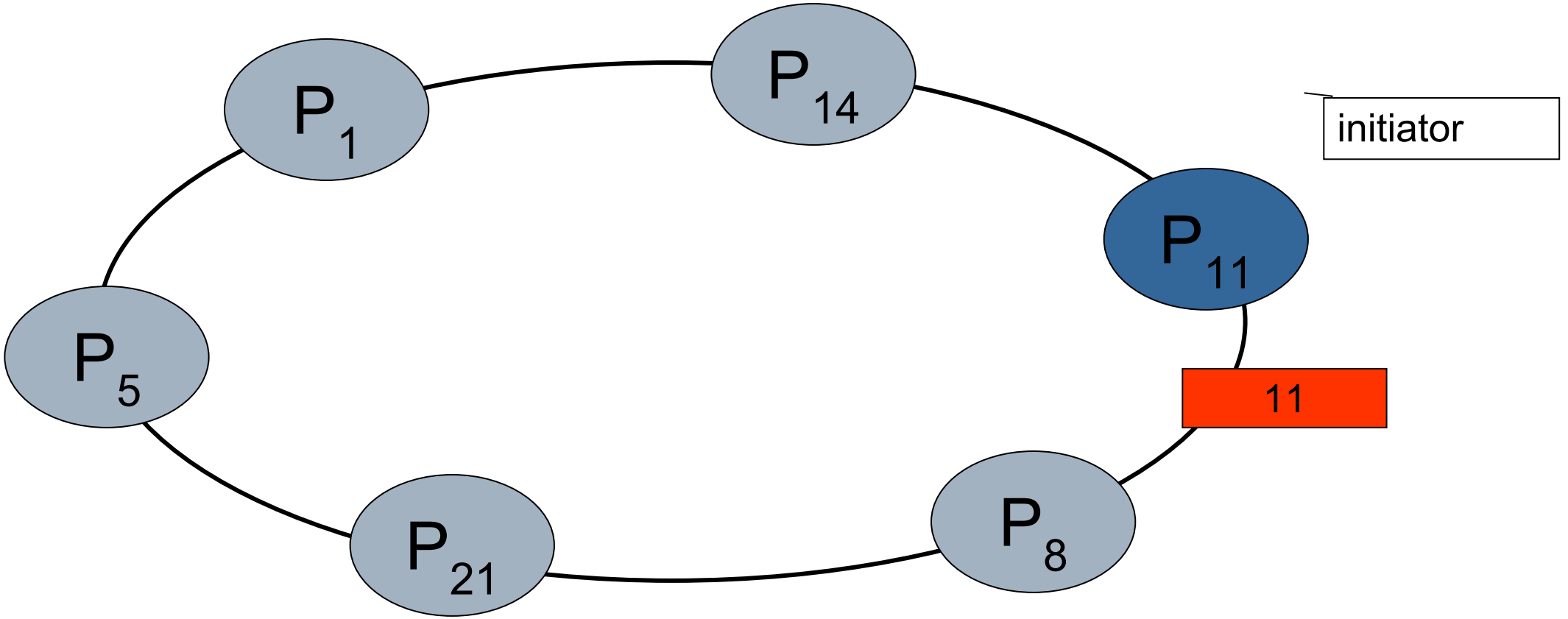
- Messages:
  - forwarded over logical ring
  - 2 types:
    - election: used during election contains identifier
    - elected: used to announce new coordinator
- Process States:
  - participant
  - non-participant

Ring Algorithm

Algorithm

- process initiating an election
  - becomes participant
  - sends election message to its neighbour

Ring Algorithm



Ring Algorithm

Algorithm

- upon receiving an election message, a process compares identifiers:
  - Received: identifier in message
  - own: identifier of process
- 3 cases:
  - Received > own
  - Received < own
  - Received = own

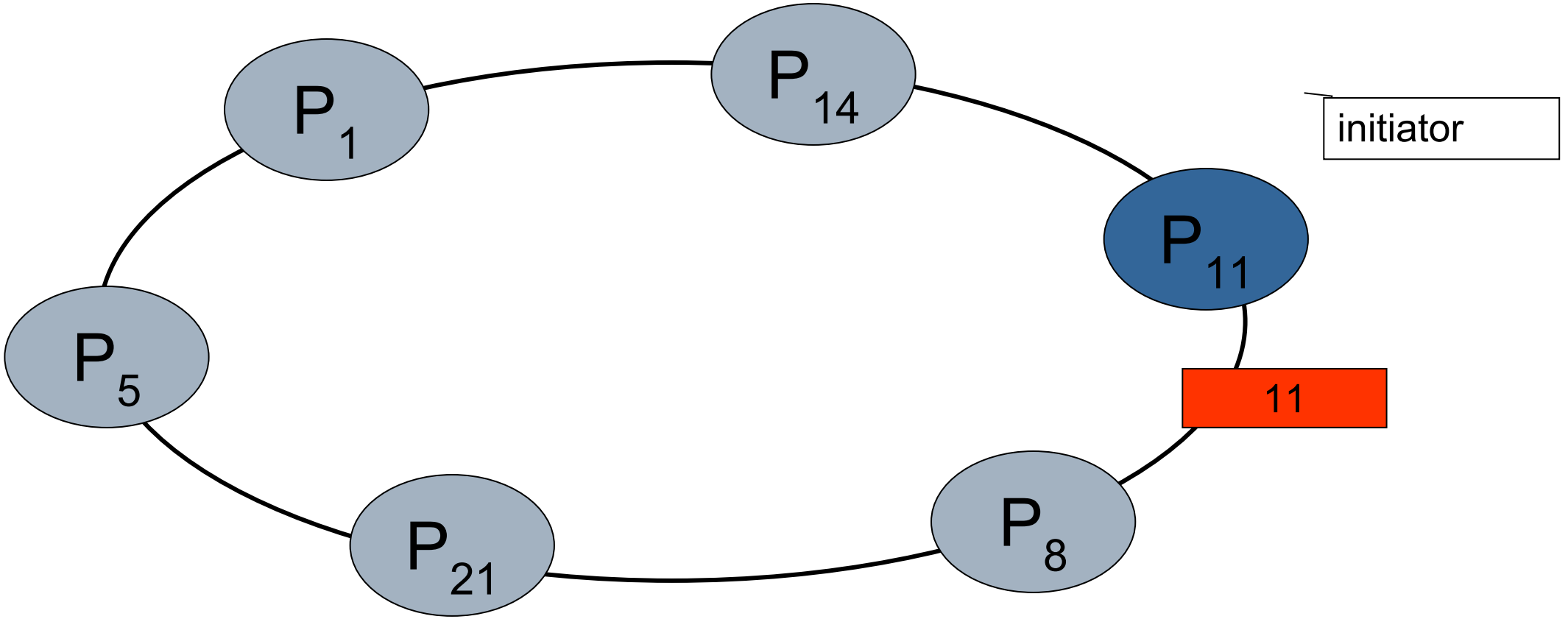
Ring Algorithm

Algorithm

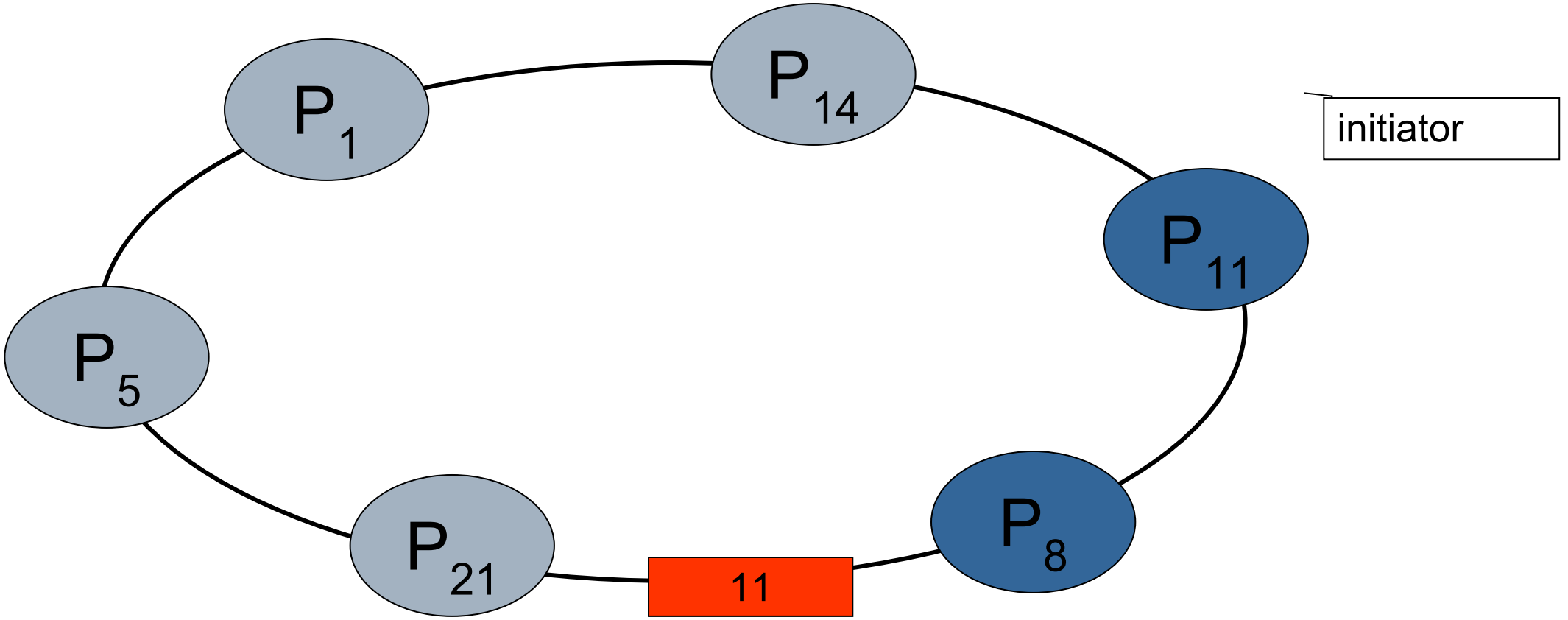
receive election message

- Received > own
  - message forwarded
  - process becomes participant

Ring Algorithm



Ring Algorithm

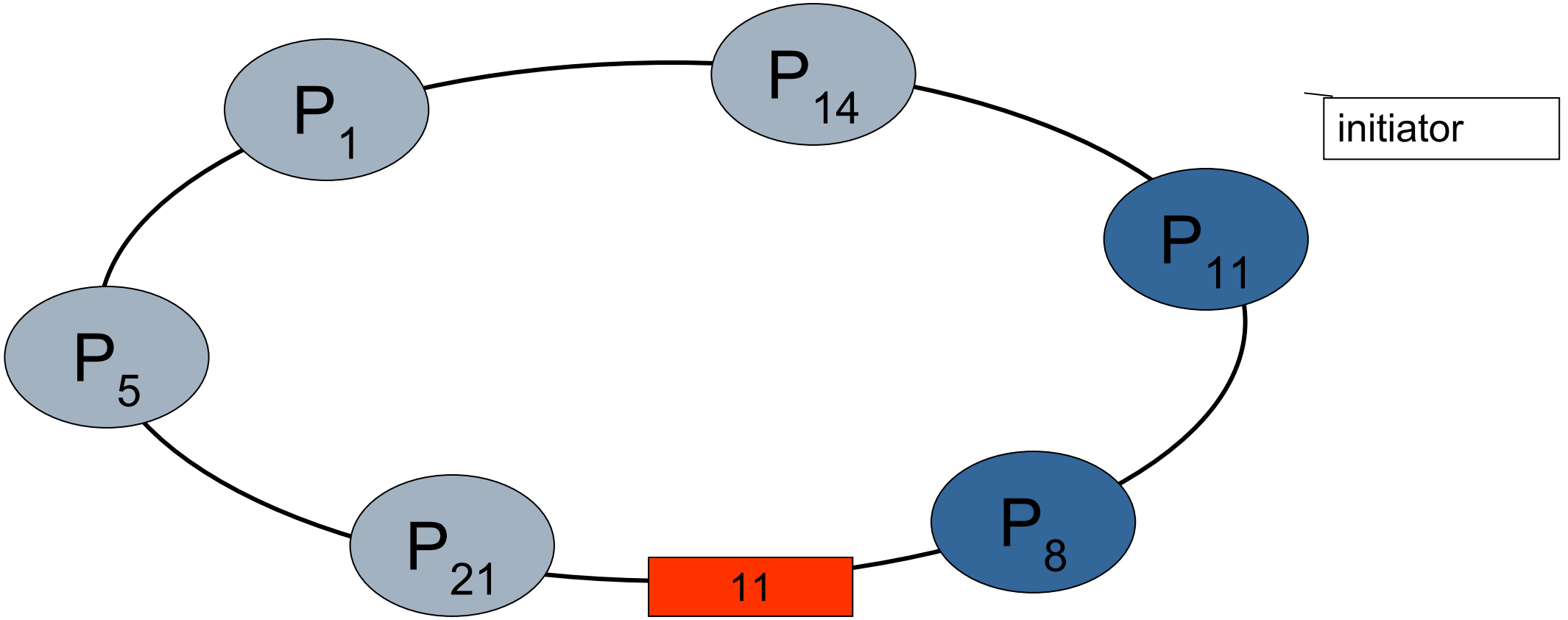


Ring Algorithm

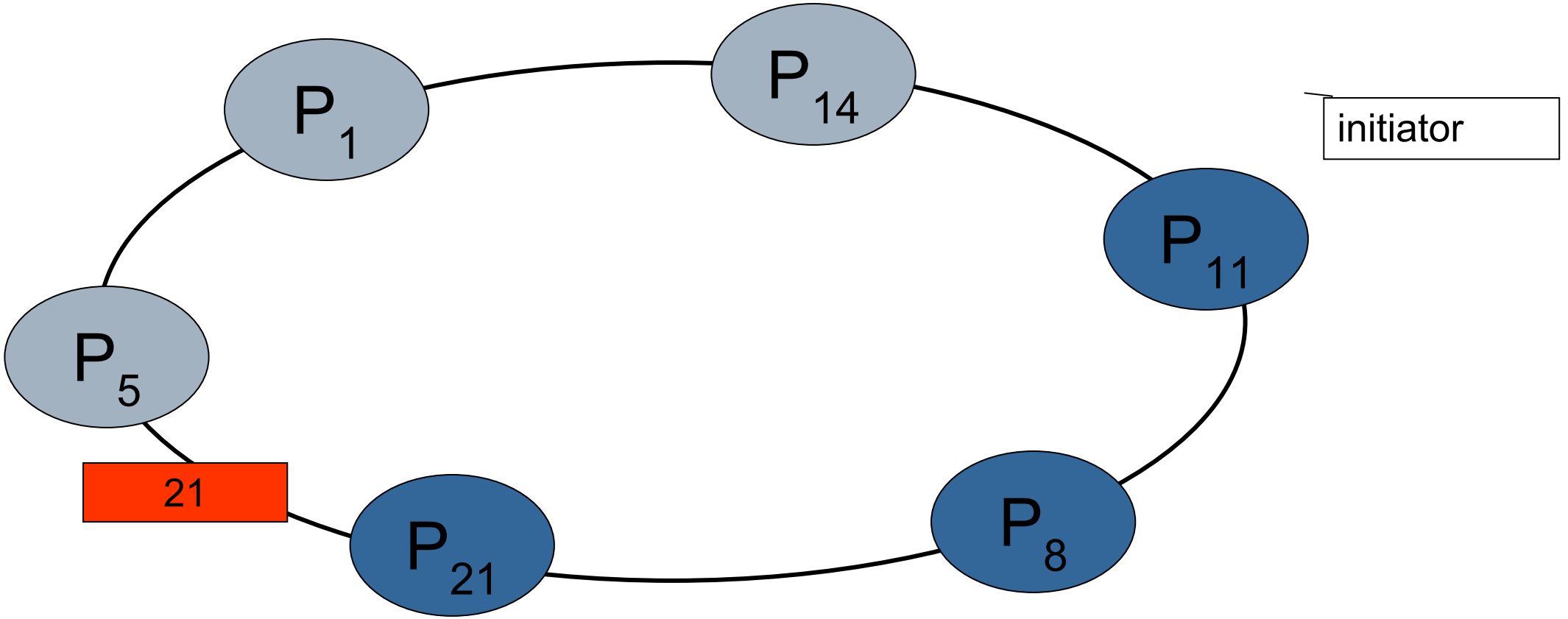
Algorithm

- receive election message
  - *Received > own*
    - *message forwarded*
    - *process becomes participant*
  - Received < own and process is non-participant
    - substitutes own identifier in message
    - message forwarded
    - process becomes participant

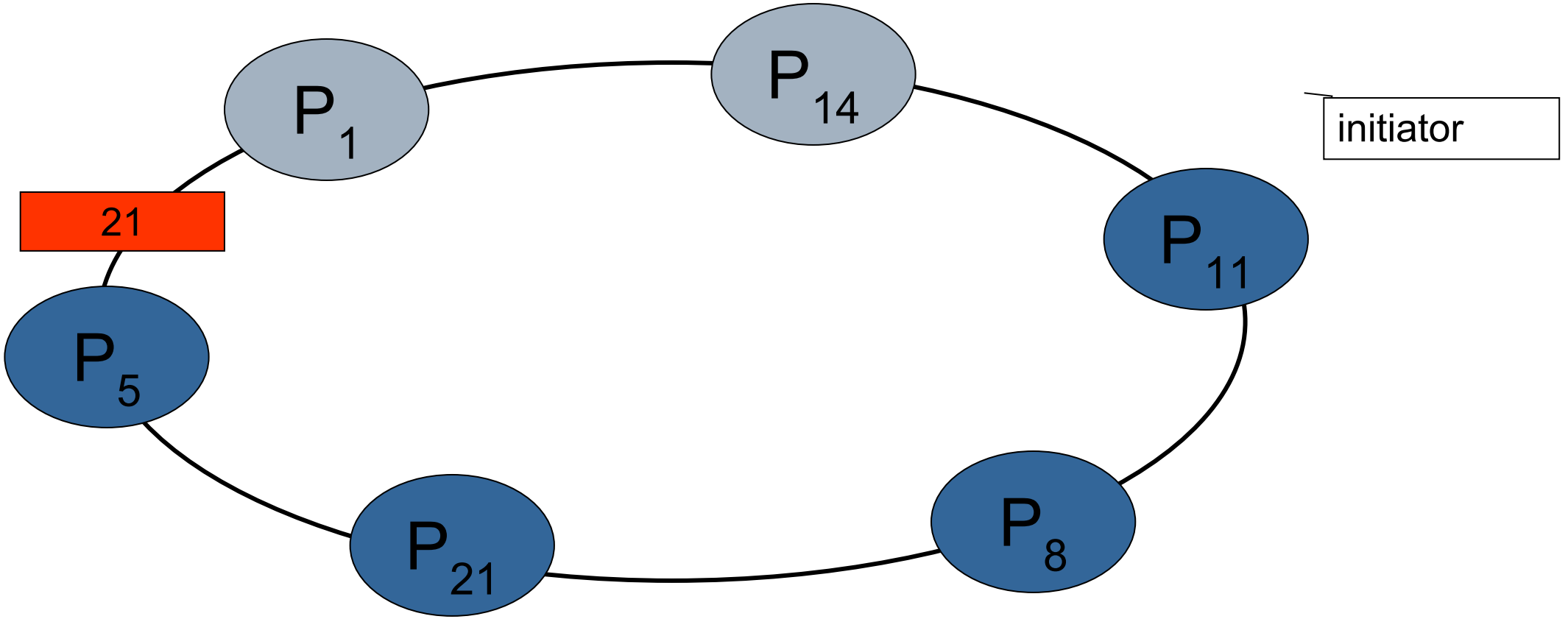
Ring Algorithm



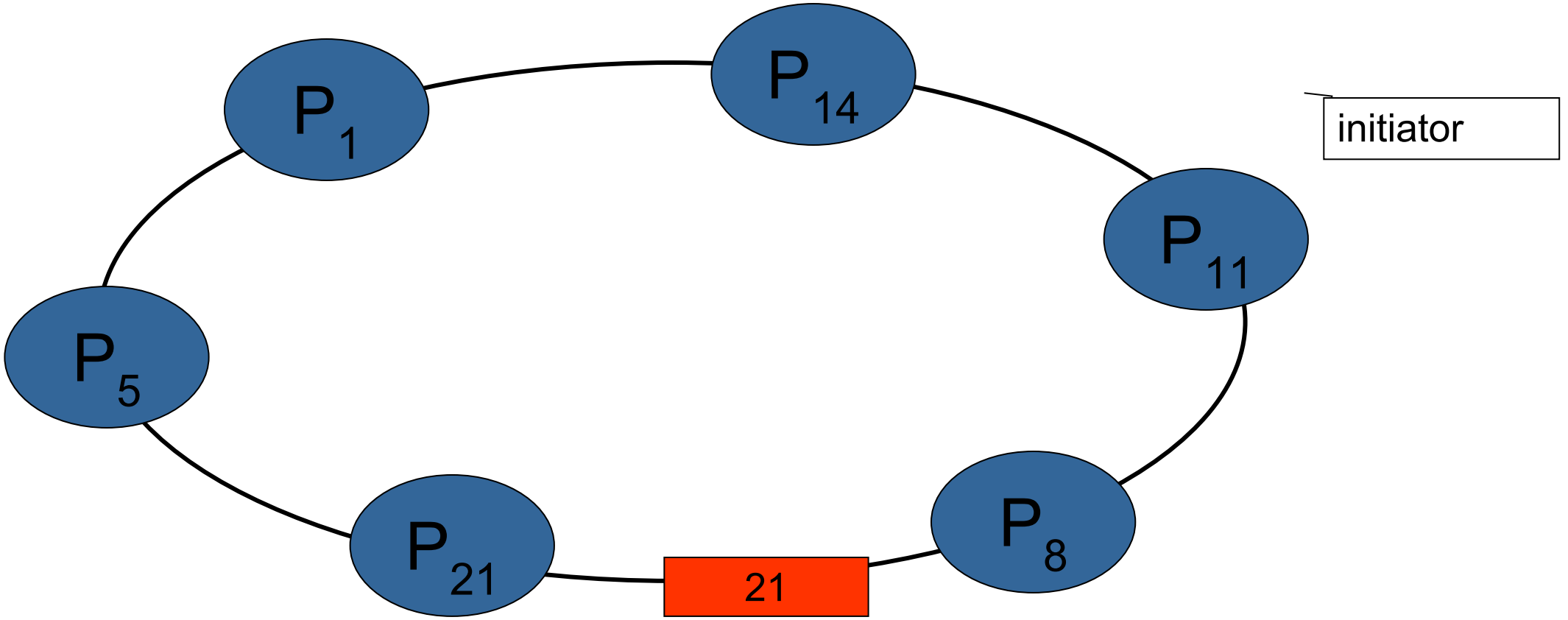
Ring Algorithm



Ring Algorithm



Ring Algorithm

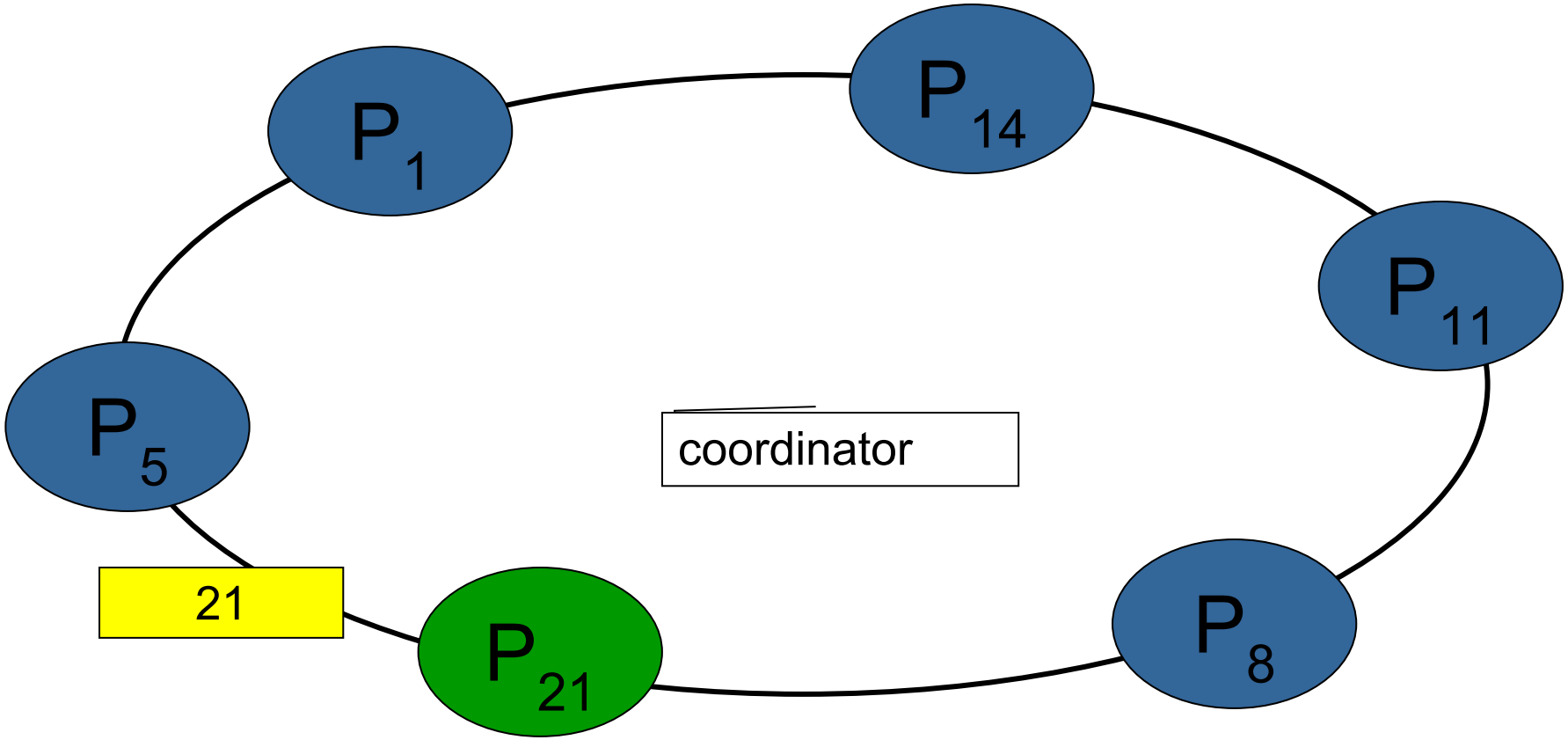


Ring Algorithm

Algorithm:

- receive election message
  - *Received > own*
    - ...
  - *Received < own and process is non-participant*
    - ...
  - Received = own
    - identifier must be greatest
    - process becomes coordinator
    - new state: non-participant
    - sends elected message to neighbour

Ring Algorithm



Ring Algorithm

Algorithm receive election message

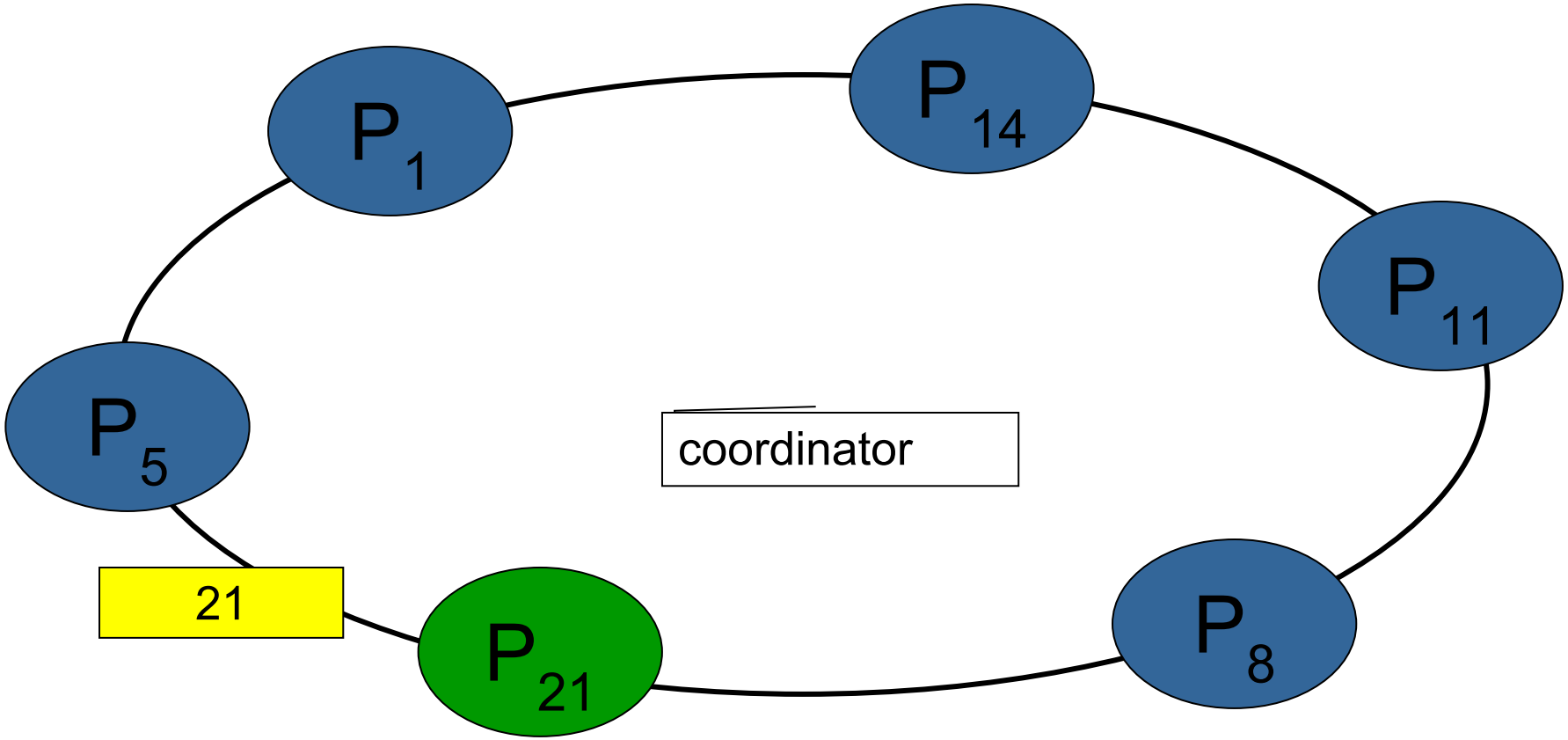
- Received > own
  - message forwarded
  - process becomes participant
- Received < own and process is non-participant
  - substitutes own identifier in message
  - message forwarded
  - process becomes participant
- Received = own
  - identifier must be greatest
  - process becomes coordinator
  - new state: non-participant
  - sends elected message to neighbour

Ring Algorithm

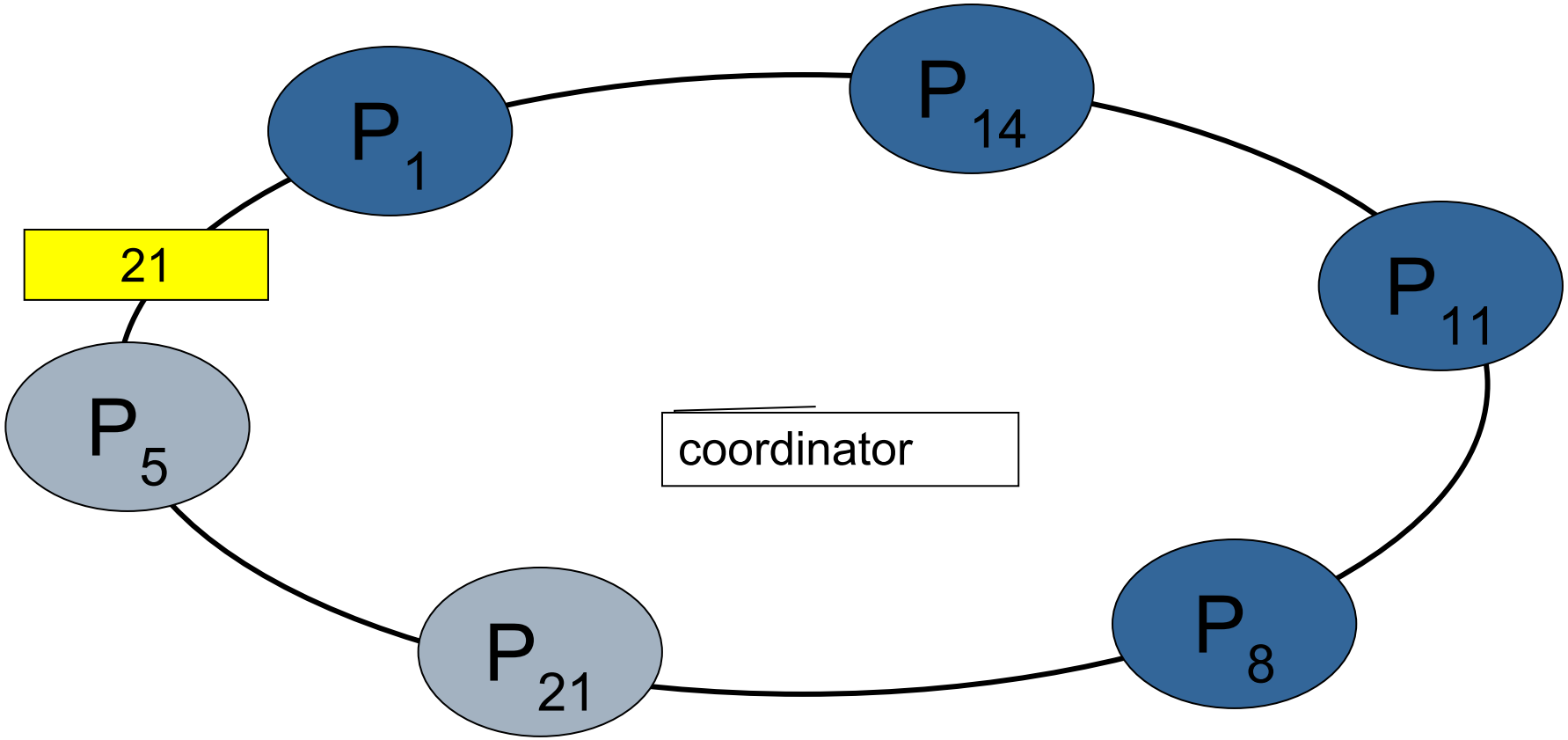
Algorithm

- receive elected message
  - participant:
    - new state: non-participant
    - forwards message
  - coordinator:
    - election process completed

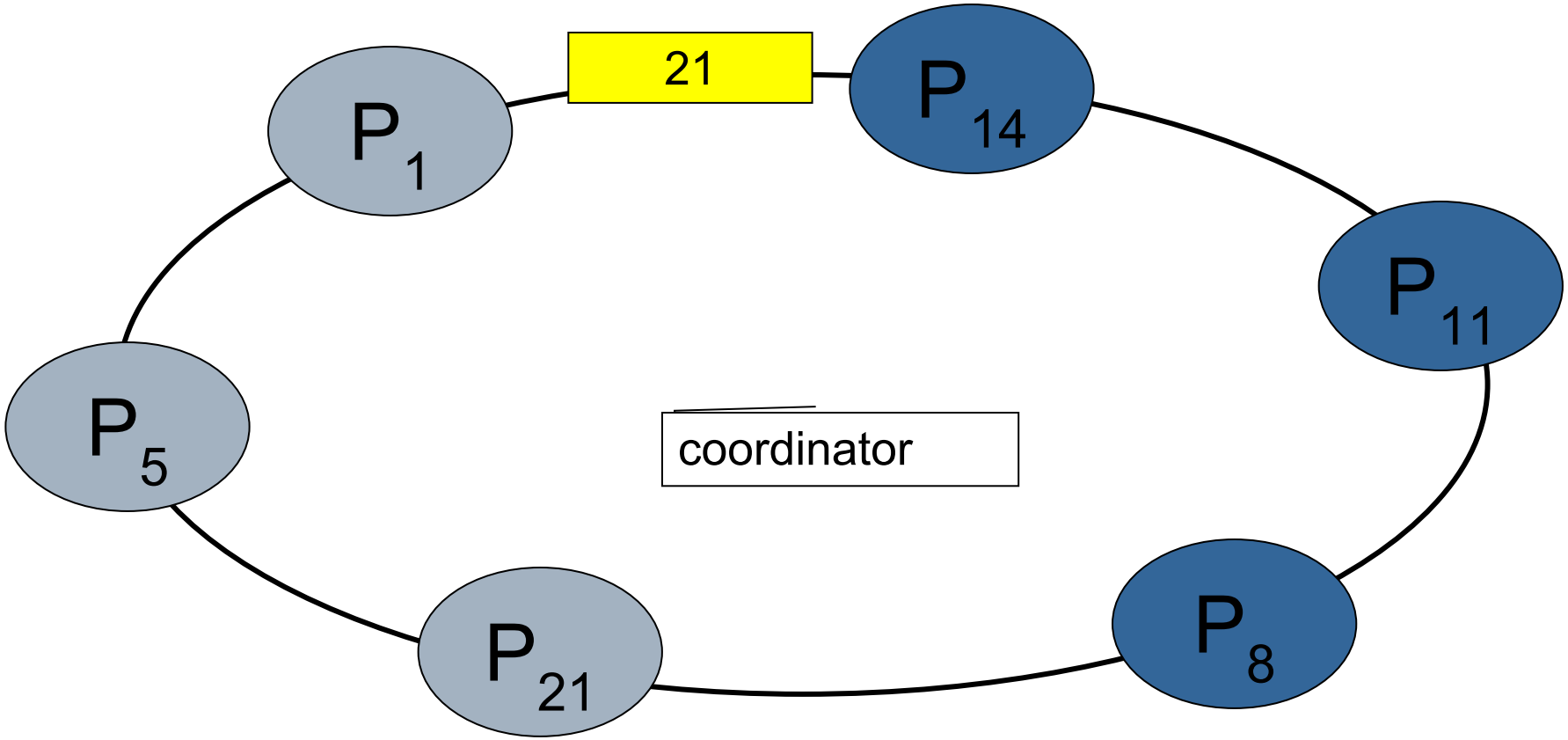
Ring Algorithm



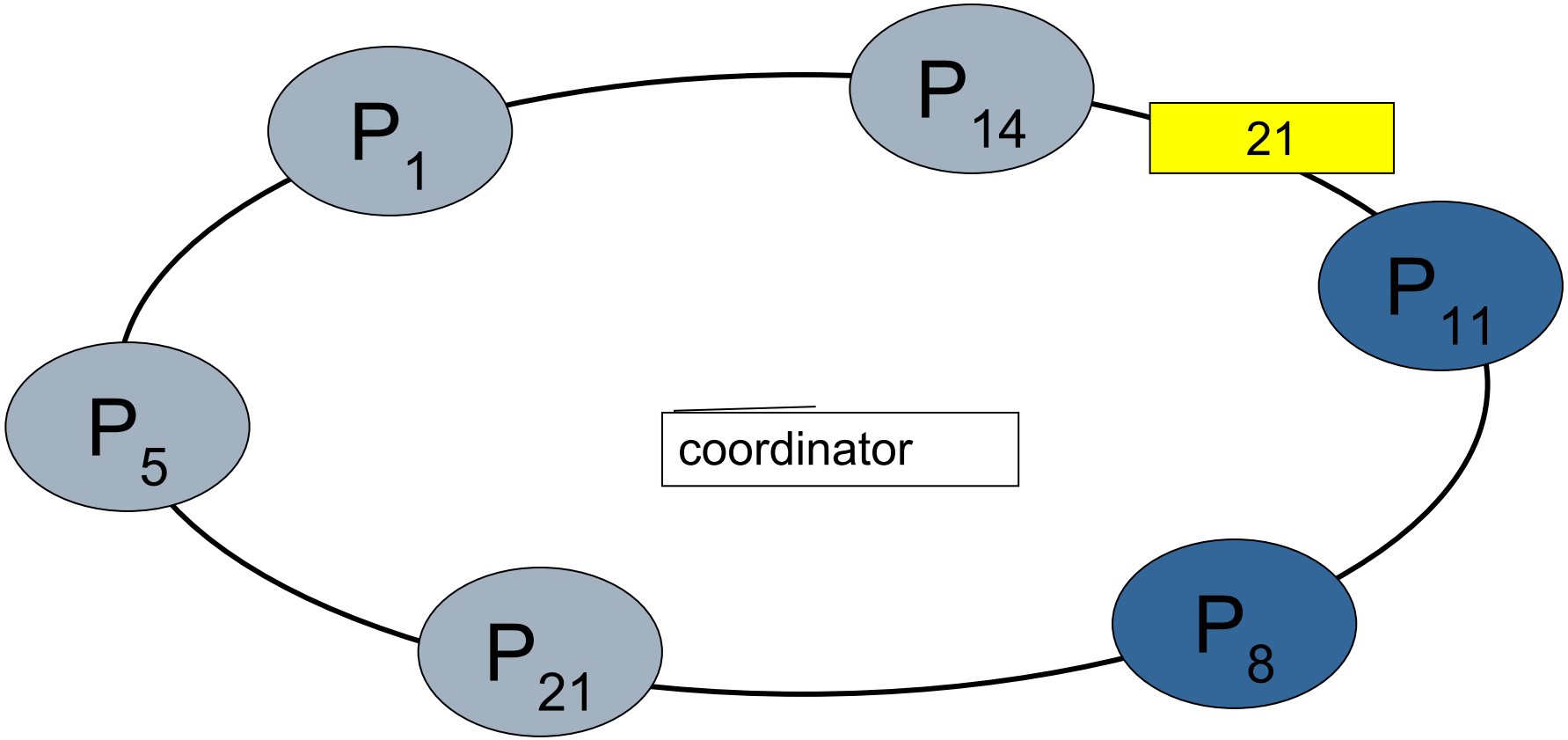
Ring Algorithm



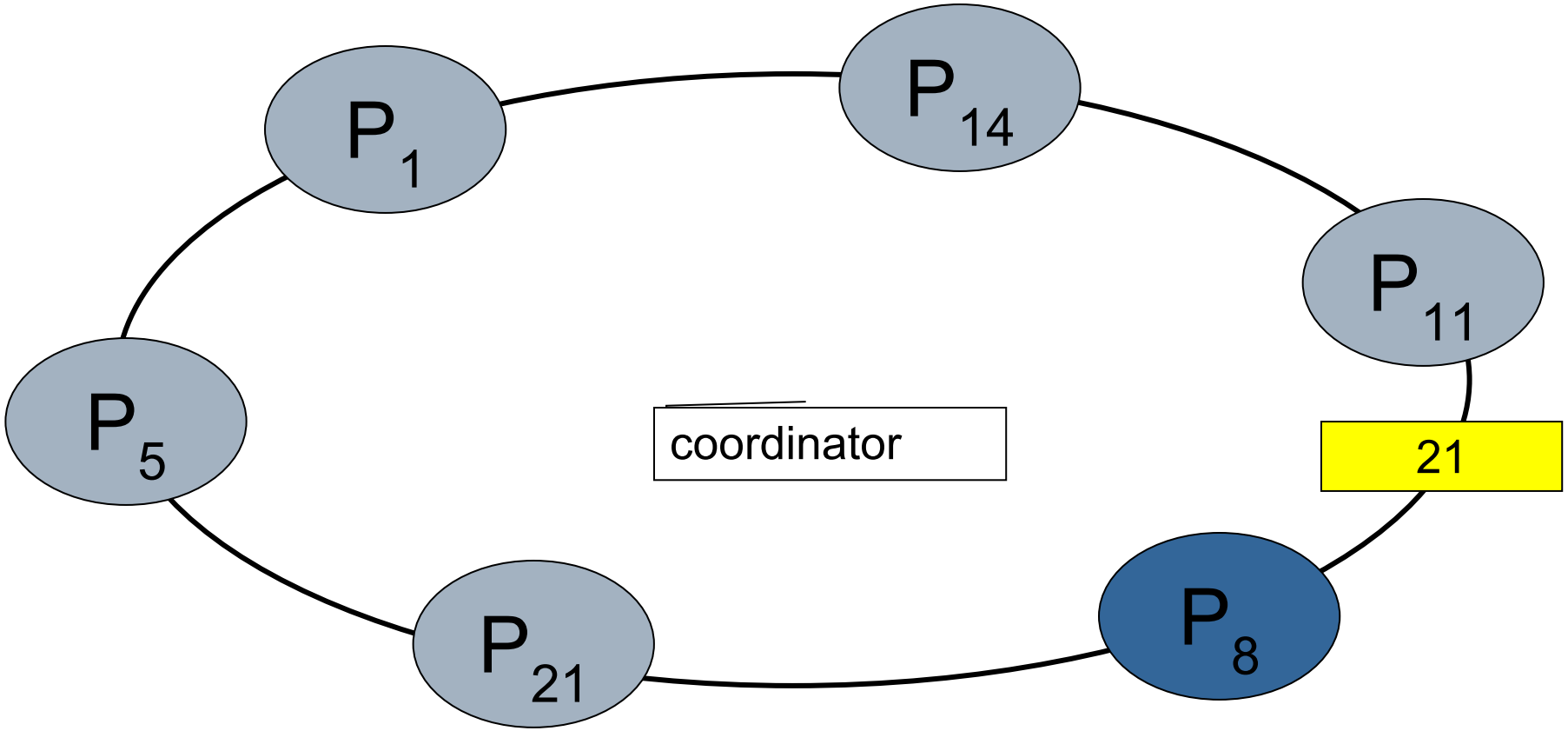
Ring Algorithm



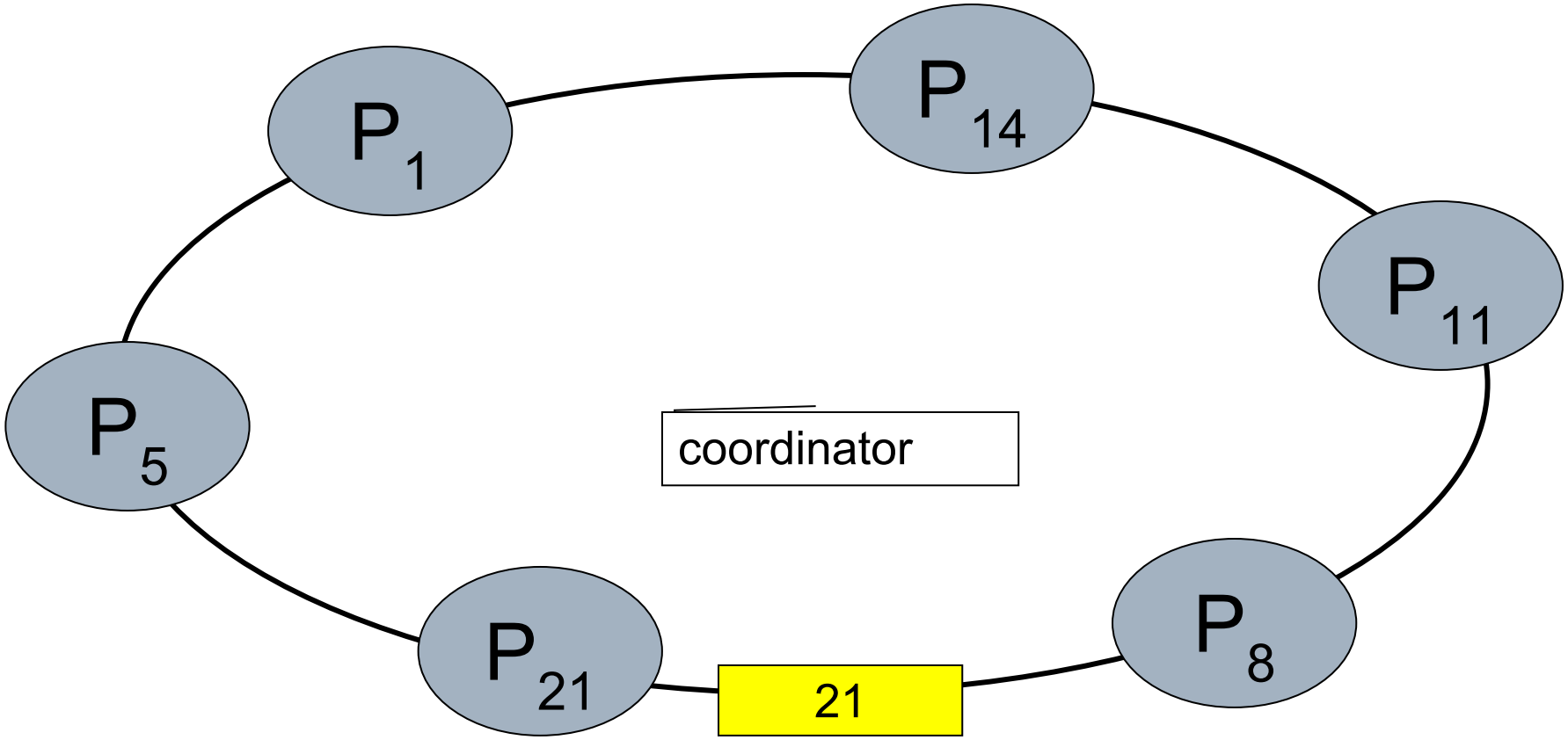
Ring Algorithm



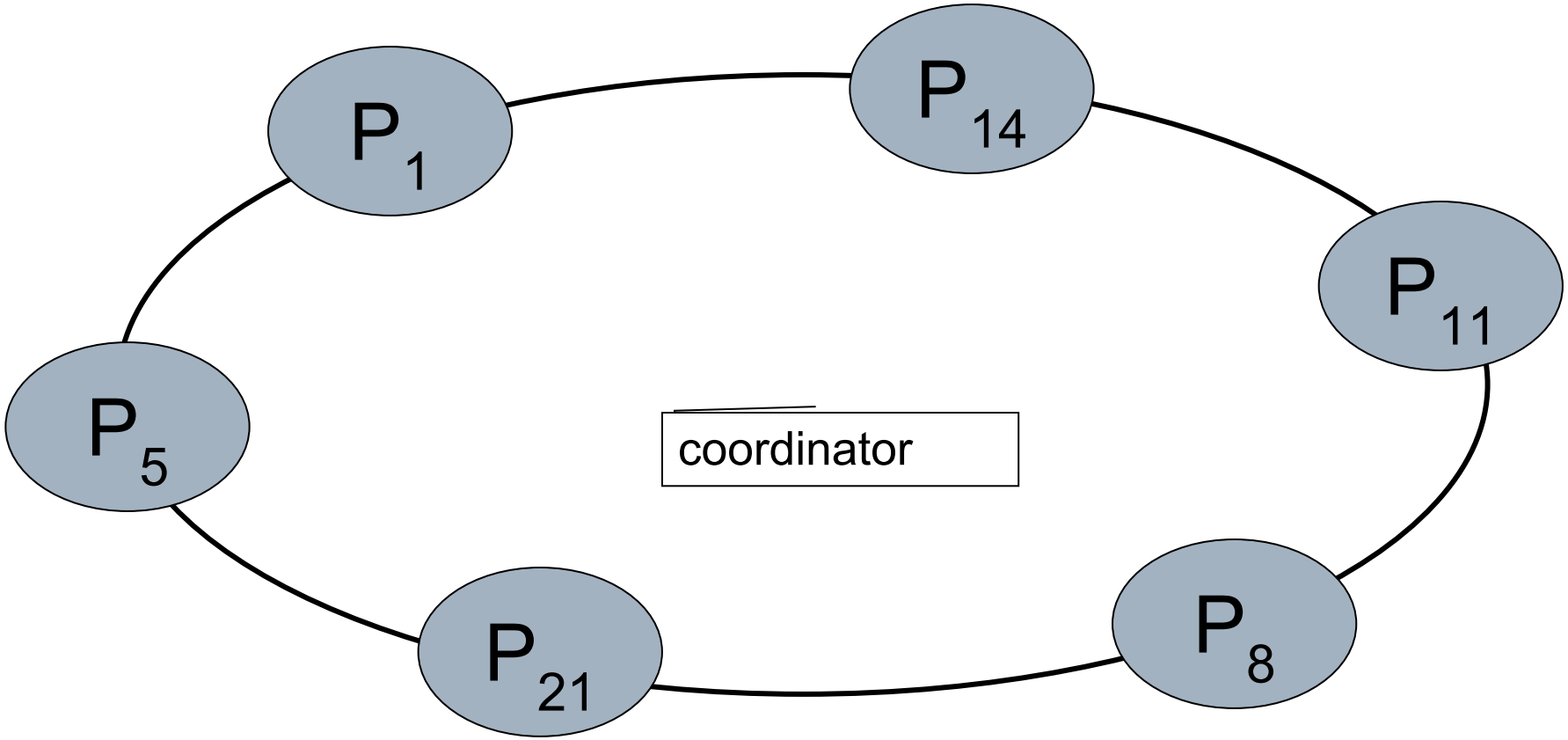
Ring Algorithm



Ring Algorithm



Ring Algorithm



References

- Andrew S. Tanenbaum and Maarten Van Steen, “Distributed Systems: Principles and Paradigms”, 2nd edition, Pearson Education.

- Pradeep K. Sinha , “Distributed Operating System” PHI Publication 20008.

- George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems: Concepts and Design", 4th Edition, Pearson Education, 2005.

Thank you !!!