Chapter 9: Distributed Mutual Exclusion Algorithms

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Introduction

- Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- Only one process is allowed to execute the critical section (CS) at any given time.
- In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

Introduction

- Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.
- Three basic approaches for distributed mutual exclusion:
  1. Token based approach
  2. Non-token based approach
  3. Quorum based approach
- Token-based approach:
  - A unique token is shared among the sites.
  - A site is allowed to enter its CS if it possesses the token.
  - Mutual exclusion is ensured because the token is unique.

Introduction

- Non-token based approach:
  - Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
- Quorum based approach:
  - Each site requests permission to execute the CS from a subset of sites (called a quorum).
  - Any two quorums contain a common site.
  - This common site is responsible to make sure that only one request executes the CS at any time.

Preliminaries

System Model

- The system consists of N sites, S_1, S_2, ..., S_N.
- We assume that a single process is running on each site. The process at site S_i is denoted by p_i.
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the 'requesting the CS' state, the site is blocked and can not make further requests for the CS. In the 'idle' state, the site is executing outside the CS.
- In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called the *idle token* state).
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

Requirements

Requirements of Mutual Exclusion Algorithms

1. **Safety Property:** At any instant, only one process can execute the critical section.
2. **Liveness Property:** This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
3. **Fairness:** Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

Performance Metrics

The performance is generally measured by the following four metrics:

- **Message complexity:** The number of messages required per CS execution by a site.
- **Synchronization delay:** After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1).

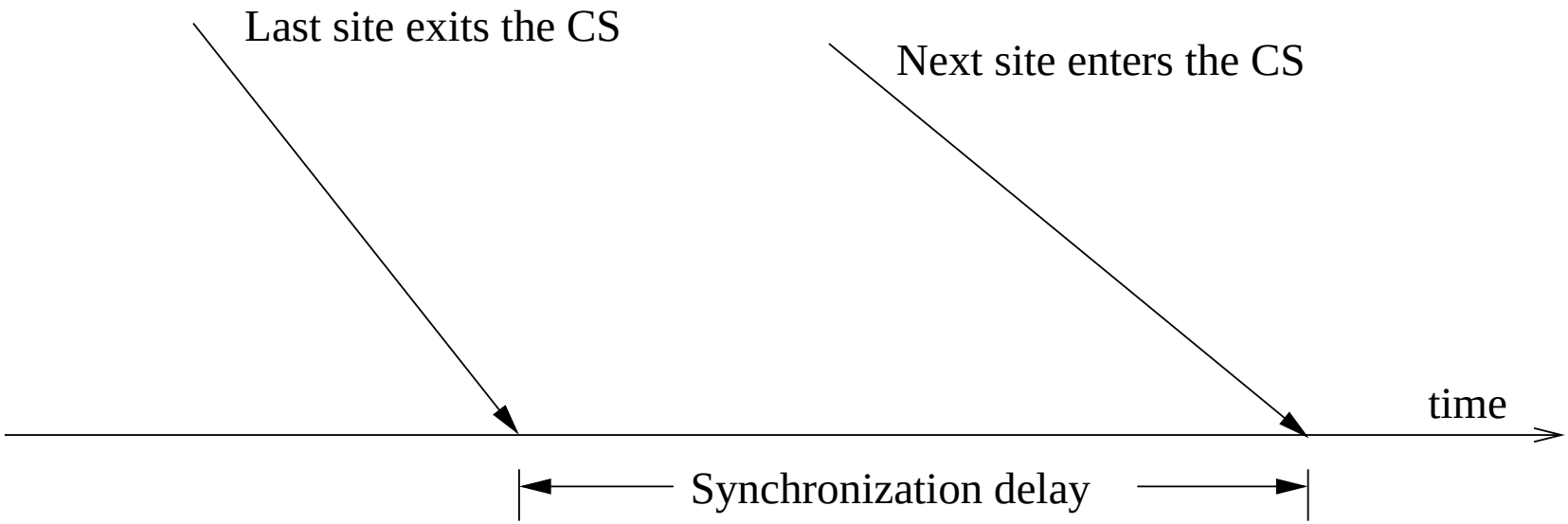


Figure 1: Synchronization Delay.

Performance Metrics

- **Response time:** The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).

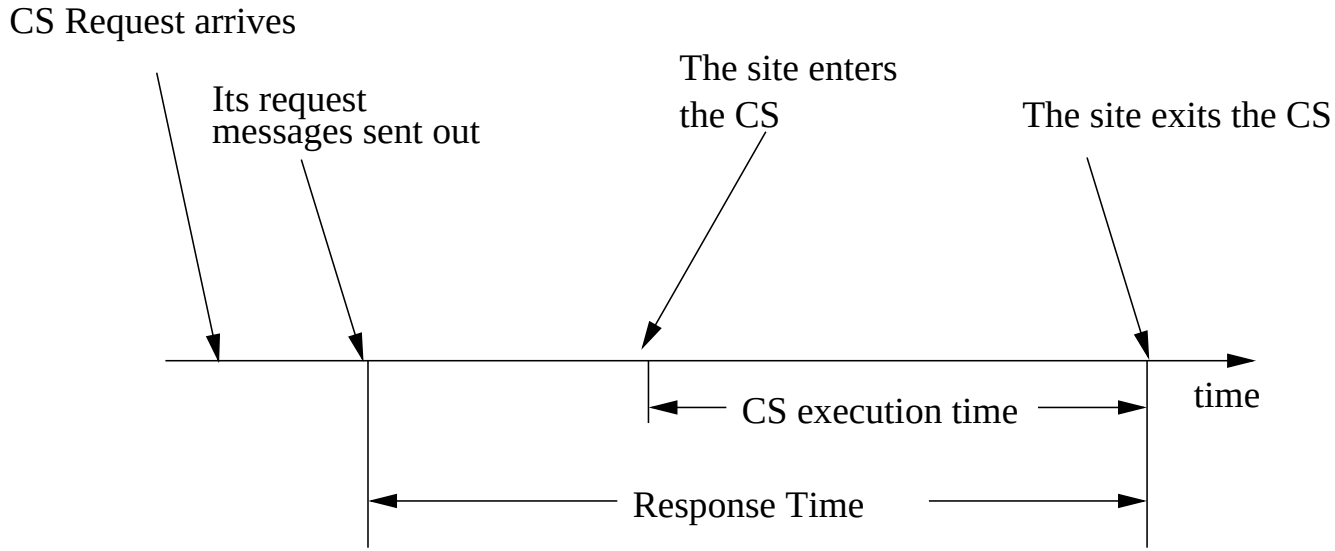


Figure 2: Response Time.

- **System throughput:** The rate at which the system executes requests for the CS.

$$\text{system throughput} = 1/(SD+E)$$

where SD is the synchronization delay and E is the average critical section execution time.

Performance Metrics

Low and High Load Performance:

- We often study the performance of mutual exclusion algorithms under two special loading conditions, viz., "low load" and "high load".
- The load is determined by the arrival rate of CS execution requests.
- Under *low load* conditions, there is seldom more than one request for the critical section present in the system simultaneously.
- Under *heavy load* conditions, there is always a pending request for critical section at a site.

Lamport's Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site S_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps.
- This algorithm requires communication channels to deliver messages the FIFO order.

The 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,places site S_i's request on $request\_queue_j$ and it 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 $request\_queue_i$.

The Algorithm

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.

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.

correctness

Theorem: Lamport's algorithm achieves mutual exclusion.
Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*.
- This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their *request_queues* and condition L1 holds at them. Without loss of generality, assume that S_i's request has smaller timestamp than the request of S_j.
- From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of S_i must be present in $request\_queue_j$ when S_j was executing its CS. This implies that S_j's own request is at the top of its own *request_queue* when a smaller timestamp request, S_i's request, is present in the $request\_queue_j$ – a contradiction!

correctness

Theorem: Lamport's algorithm is fair.
Proof:

- The proof is by contradiction. Suppose a site S_i's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i.
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But *request_queue* at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i's request must be placed ahead of the S_j's request in the $request\_queue_j$. This is a contradiction!

Performance

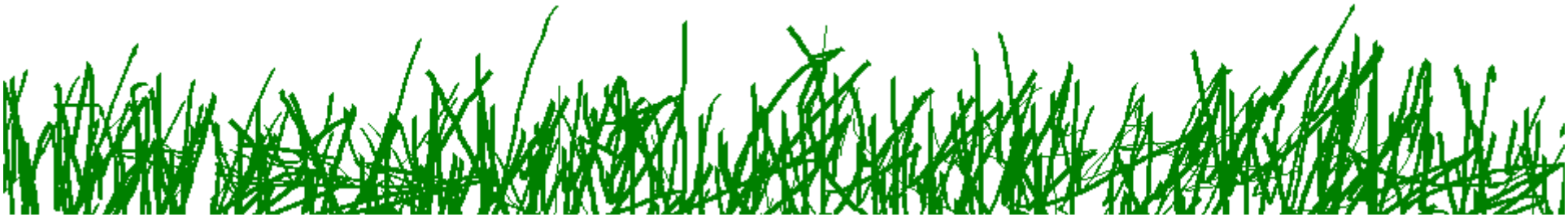
- For each CS execution, Lamport's algorithm requires $(N-1)$ REQUEST messages, $(N-1)$ REPLY messages, and $(N-1)$ RELEASE messages.
- Thus, Lamport's algorithm requires $3(N-1)$ messages per CS invocation.
- Synchronization delay in the algorithm is T.

An optimization

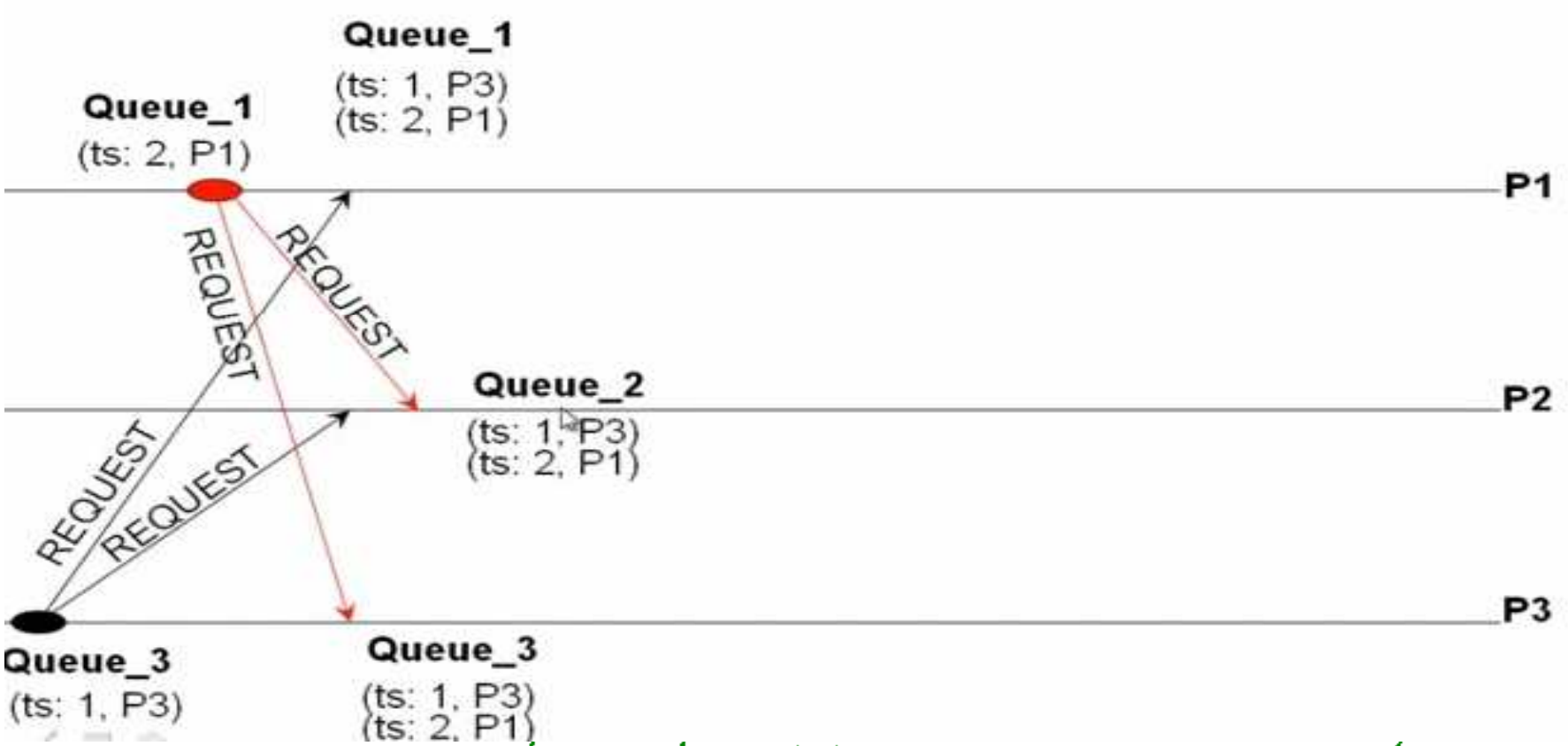
- In Lamport's algorithm,REPLY messages can be omitted in certain situations. For example, if site S_j receives a REQUEST message from site S_i after it has sent its own REQUEST message with timestamp higher than the timestamp of site S_i's request, then site S_j need not send a REPLY message to site S_i.
- This is because when site S_i receives site S_j's request with timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between $3(N-1)$ and $2(N-1)$ messages per CS execution.

Lamport's Distributed Mutual Exclusion Algorithm

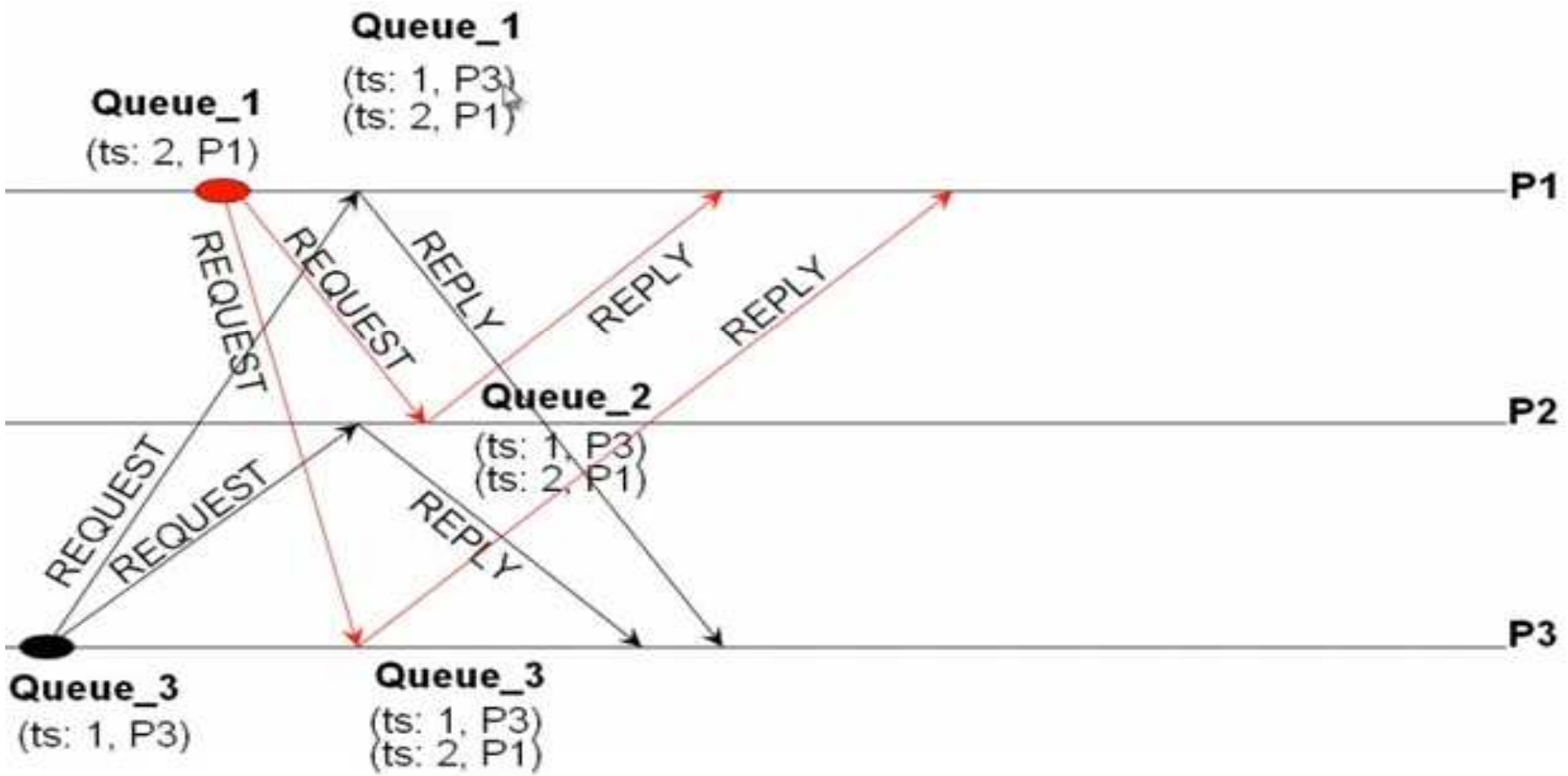
Reference : Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems,



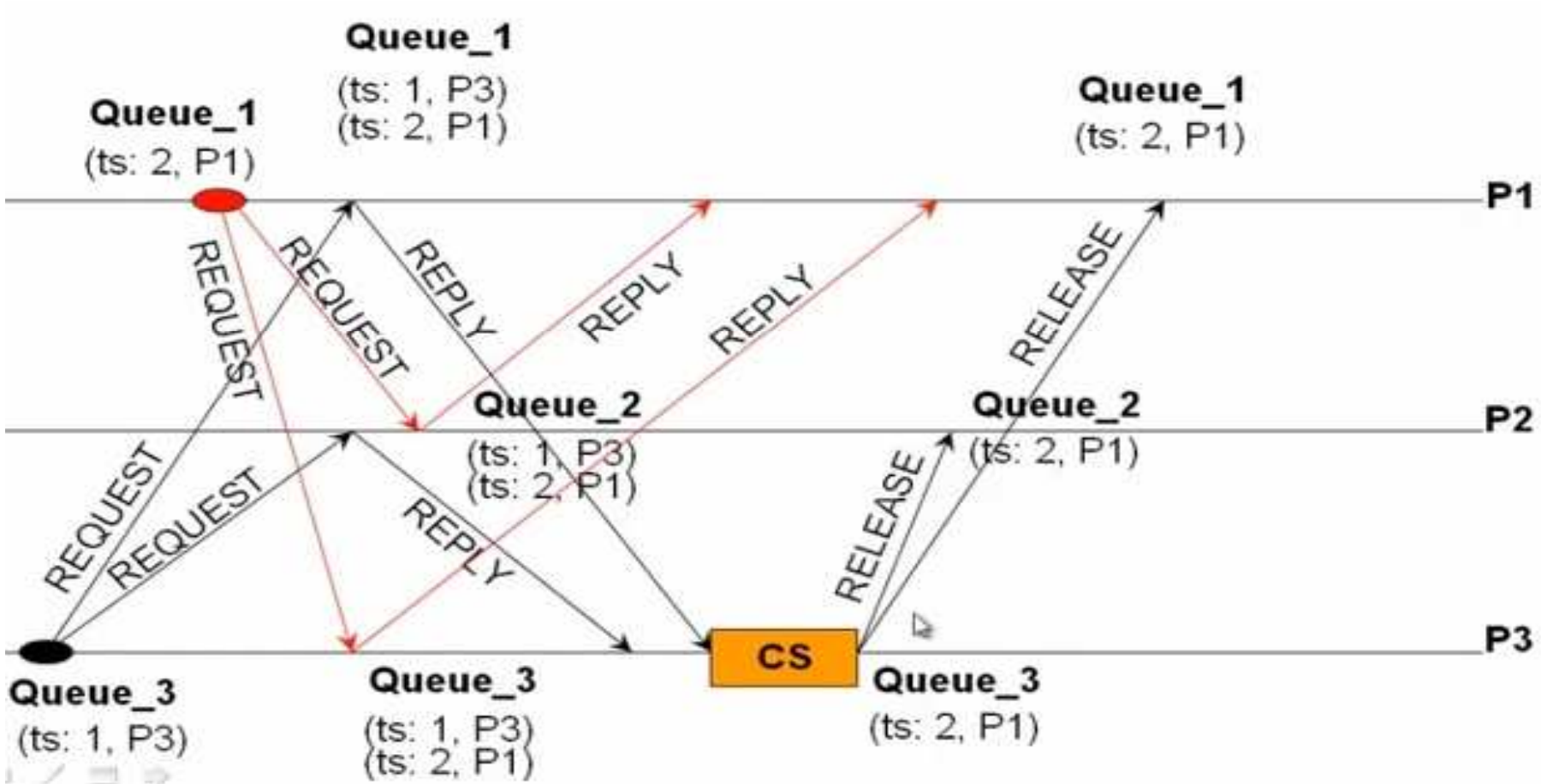
Lamport's Distributed Mutual Exclusion Algorithm



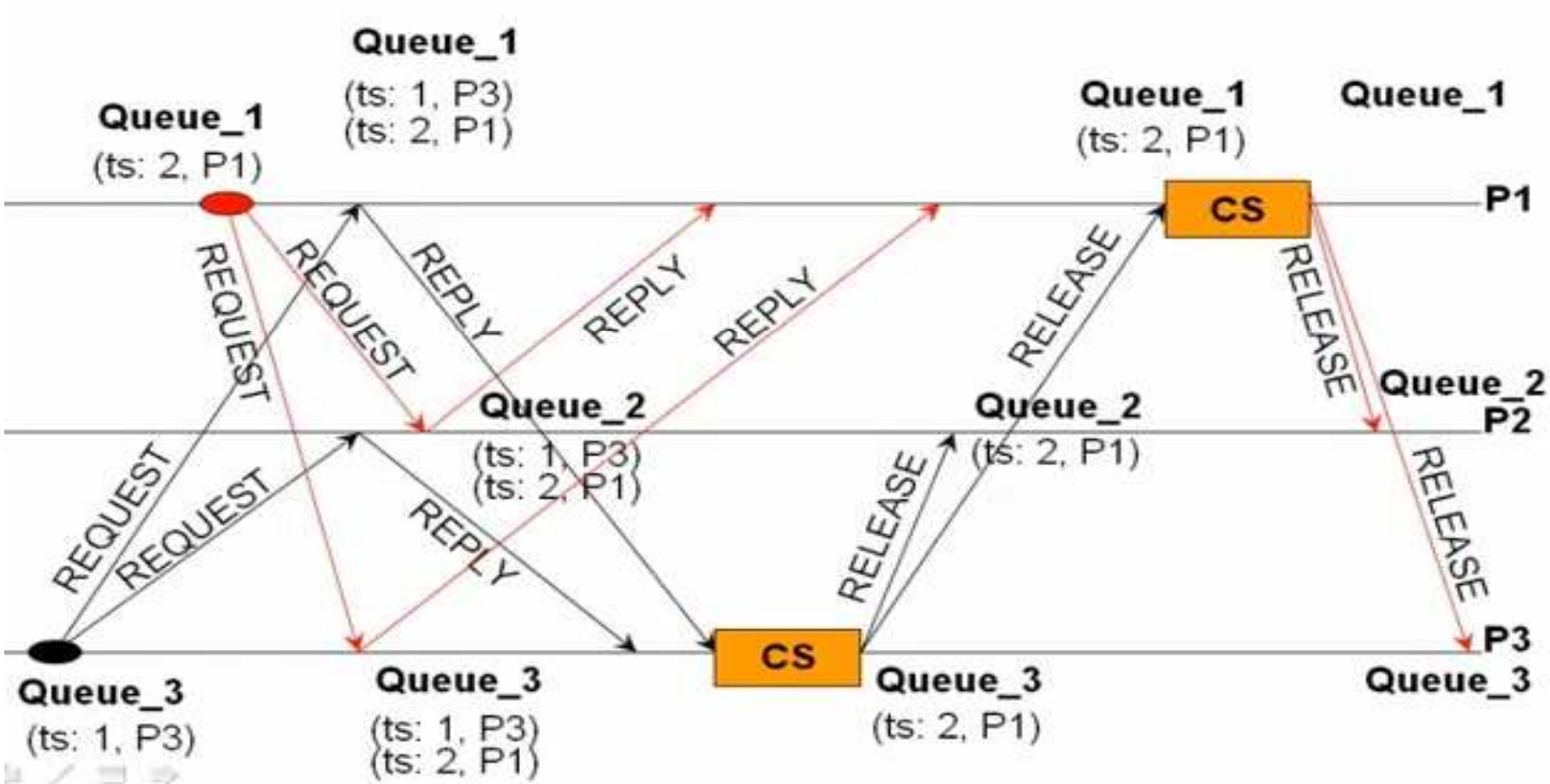
Lamport's Distributed Mutual Exclusion Algorithm



Lamport's Distributed Mutual Exclusion Algorithm



Lamport's Distributed Mutual Exclusion Algorithm

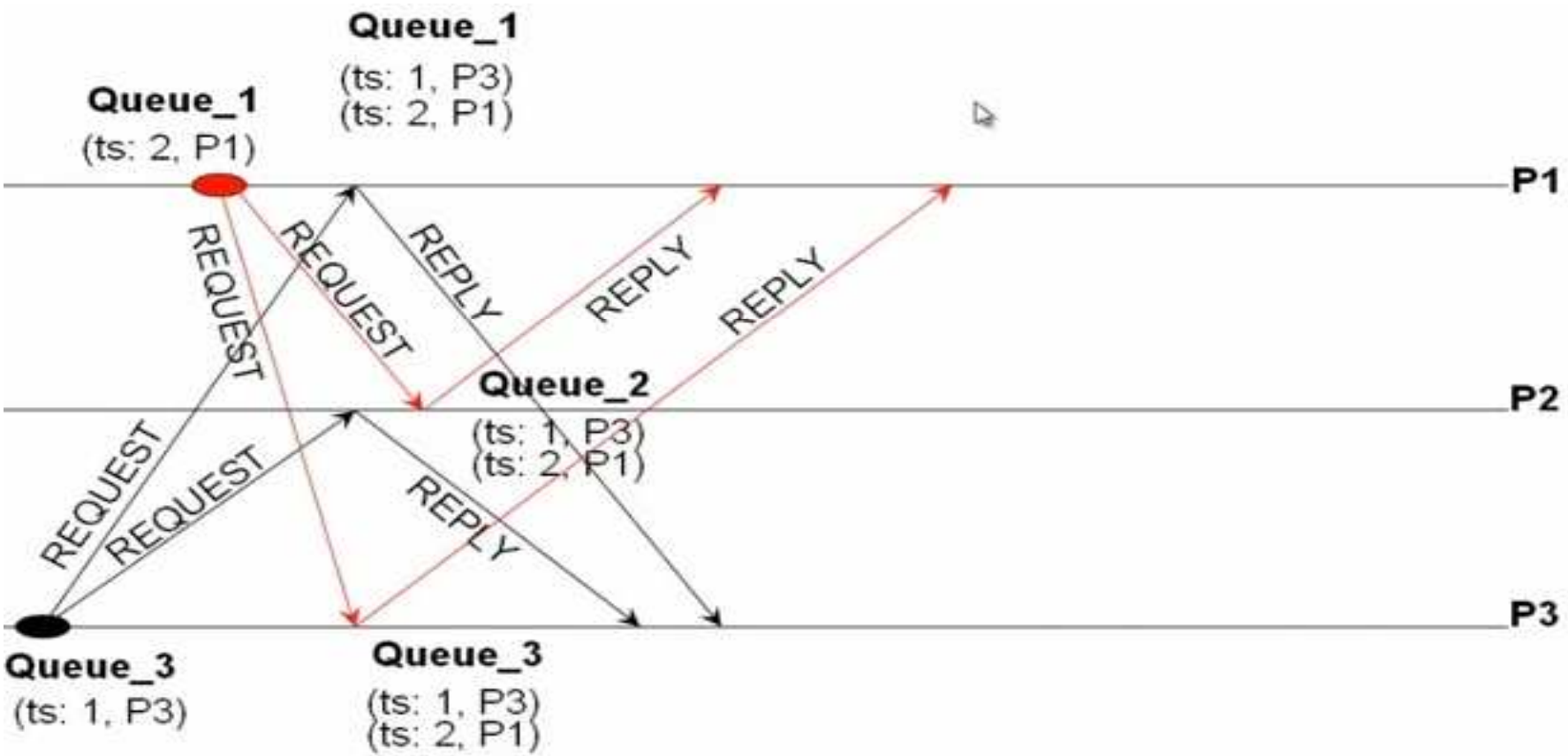


Lamport's Distributed Mutual Exclusion Algorithm

- **# Messages:** 3 (N-1)
  - N-1 REQUESTS; N-1 REPLY; N-1 RELEASE
- **Synchronization Delay:** T
- **System Throughput:** 1/(T+E)

- **Optimization:** Process Pi need not send a REPLY message to process Pj if the timestamp of the REQUEST message of Pj is higher than that of Pi. Process Pi can simply send a RELEASE message when done with its CS execution.
  - So, a process has to basically wait for a REPLY or RELEASE message from every other process and its REQUEST should be in the front of the queue to enter the CS.
  - With this, the # messages is between 2*(N-1) to 3*(N-1).

Lamport's Distributed Mutual Exclusion Algorithm



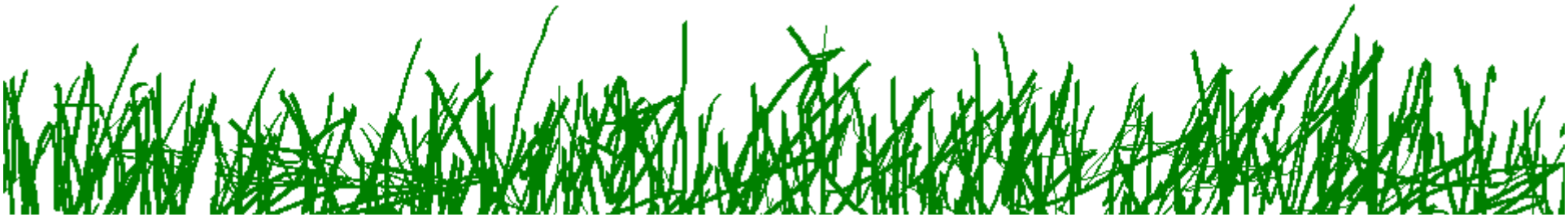
Lamport's Distributed Mutual Exclusion Algorithm

- **# Messages:** 3 (N-1)
  - N-1 REQUESTS; N-1 REPLY; N-1 RELEASE
- **Synchronization Delay:** T
- **System Throughput:** 1/(T+E)

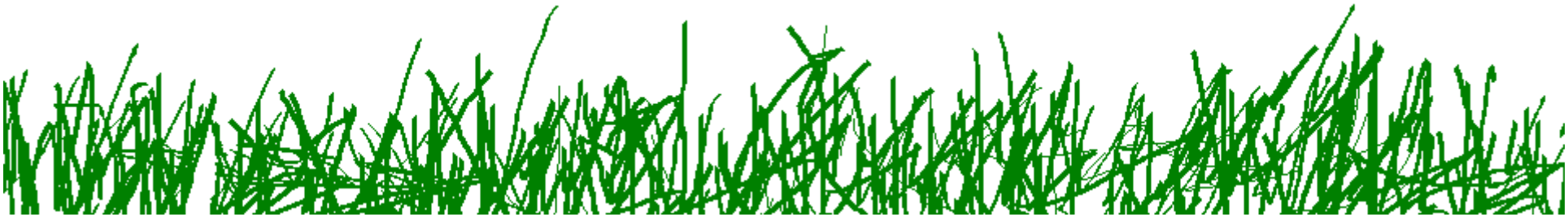
- **Optimization:** Process Pi need not send a REPLY message to process Pj if the timestamp of the REQUEST message of Pj is higher than that of Pi. Process Pi can simply send a RELEASE message when done with its CS execution.
  - So, a process has to basically wait for a REPLY or RELEASE message from every other process and its REQUEST should be in the front of the queue to enter the CS.
  - With this, the # messages is between 2*(N-1) to 3*(N-1).

Source

https://www.youtube.com/watch?v=r7SJOhGF4Nc



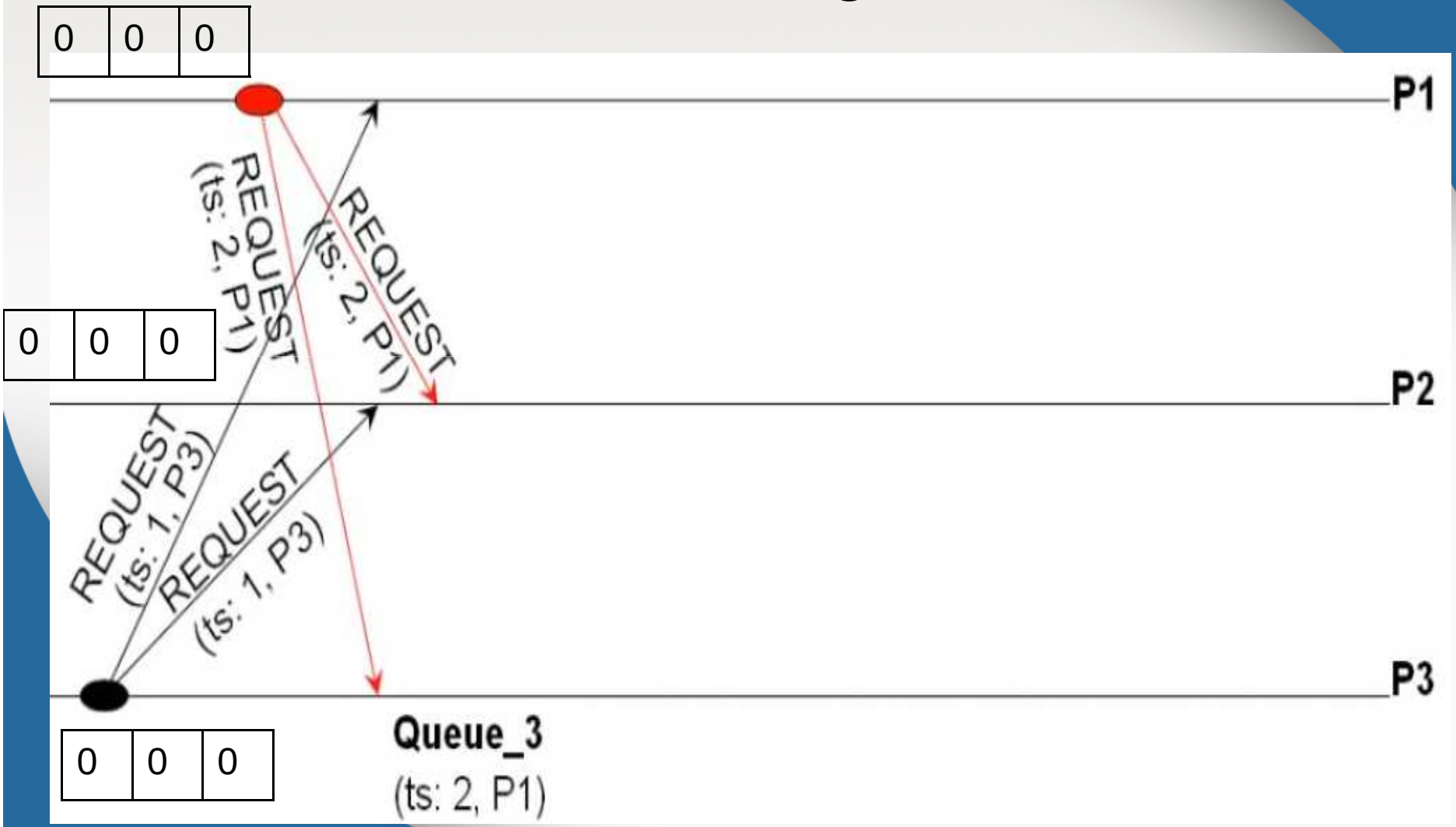
Thank You



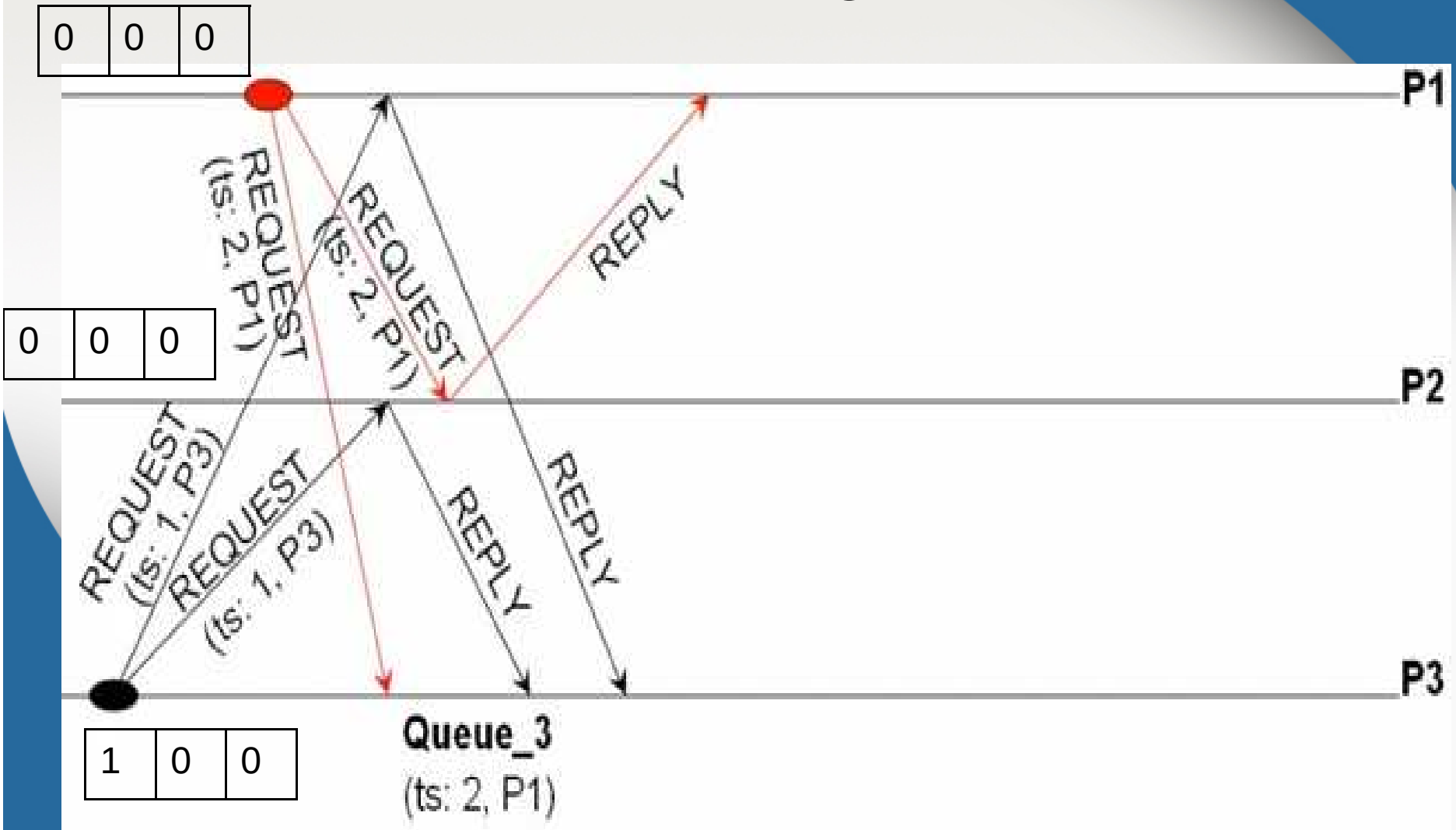
Ricart Agrawala's Distributed Mutual Exclusion Algorithm

Reference : Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems, 5th Edition

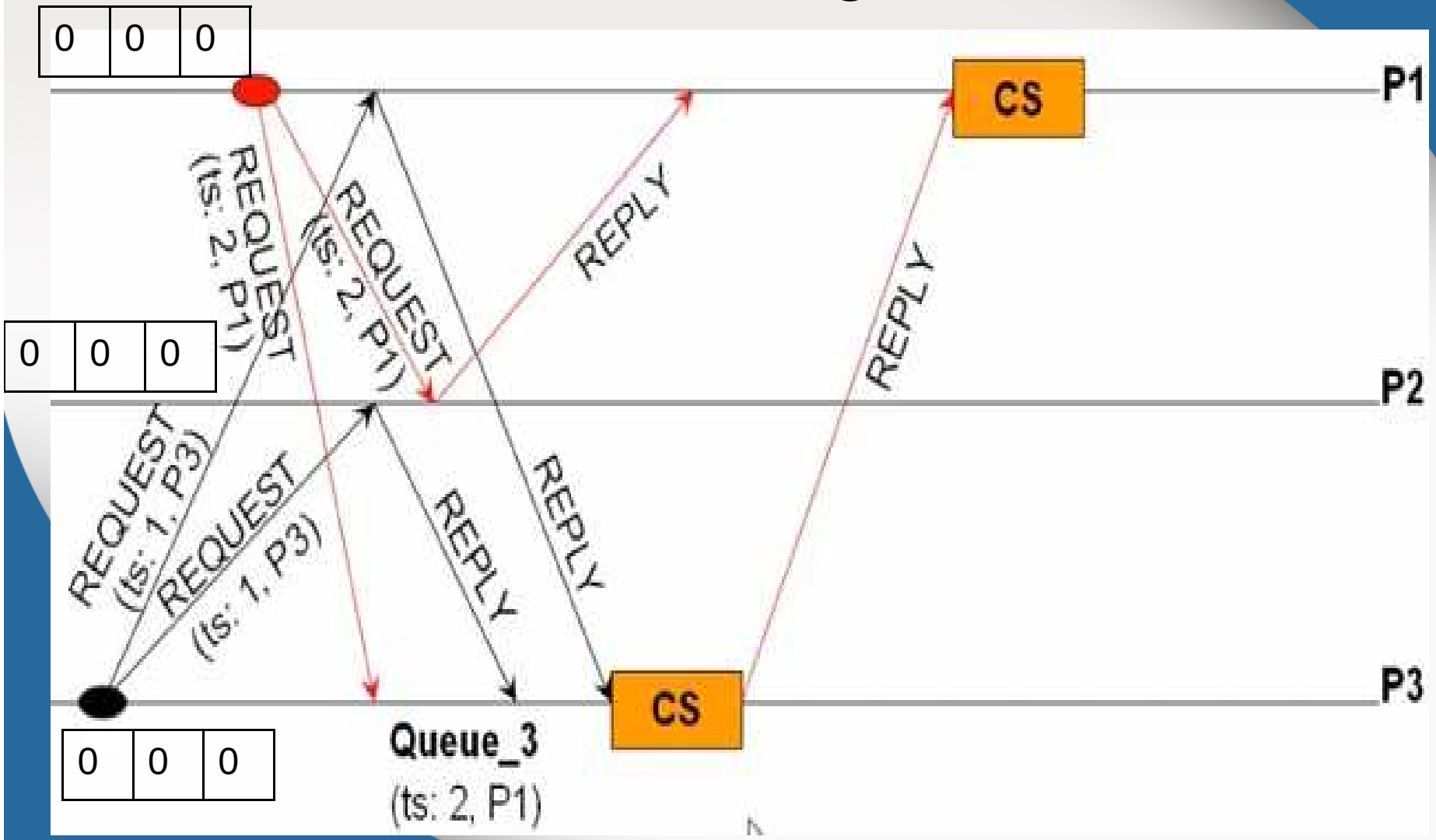
Ricart Agrawala's Distributed Mutual Exclusion Algorithm



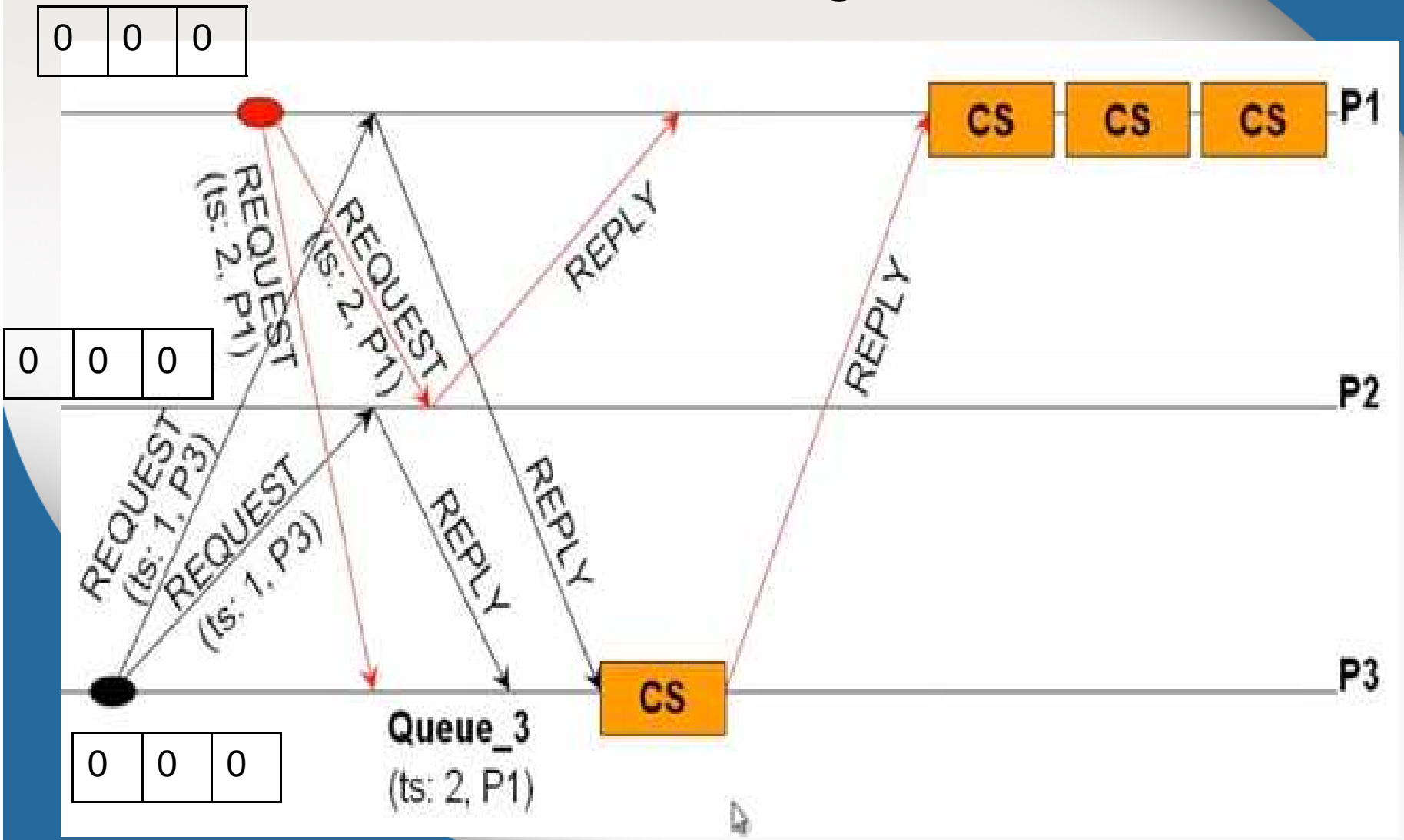
Ricart Agrawala's Distributed Mutual Exclusion Algorithm



Ricart Agrawala's Distributed Mutual Exclusion Algorithm



Ricart Agrawala's Distributed Mutual Exclusion Algorithm

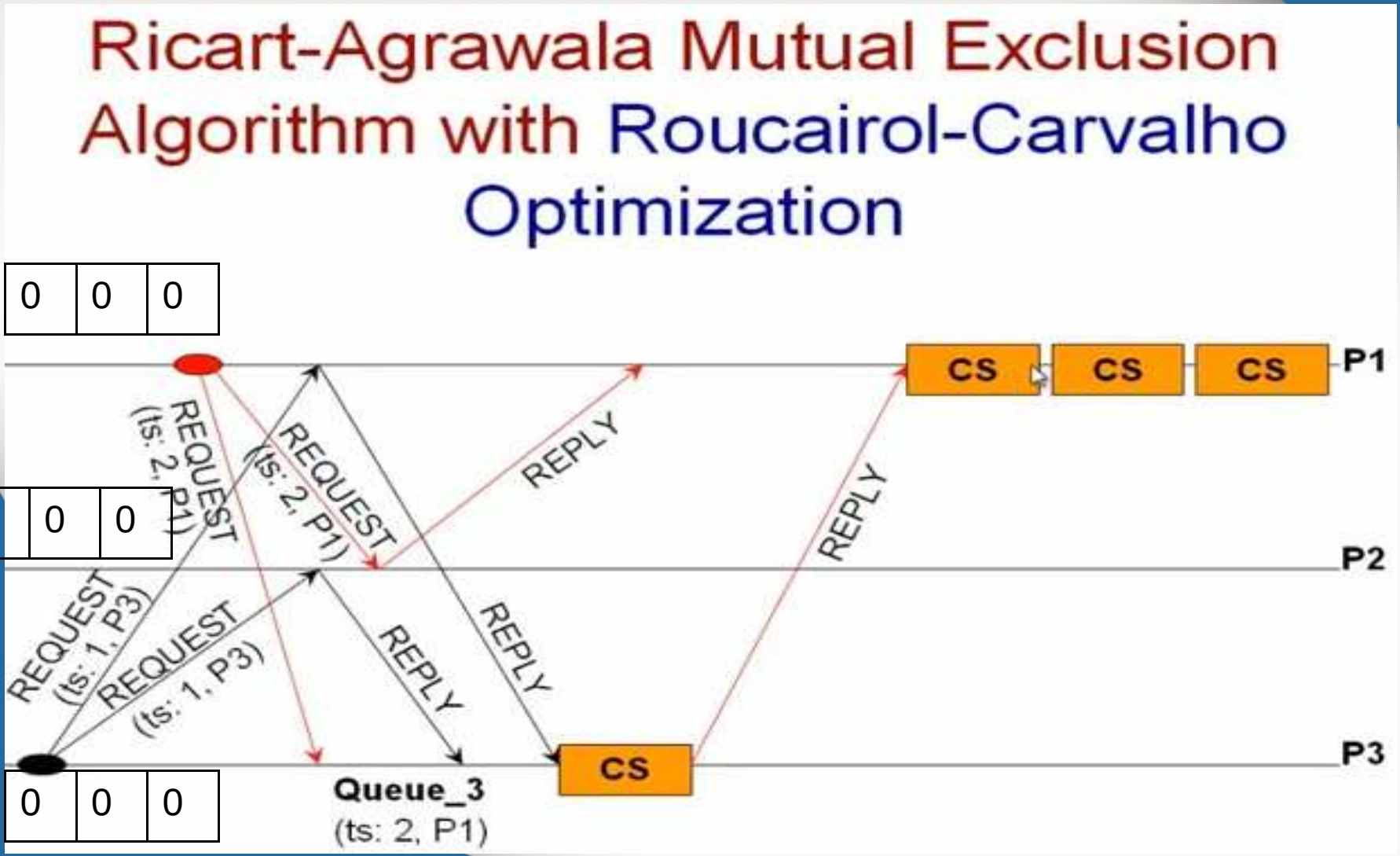


Ricart Agrawala's Distributed Mutual Exclusion Algorithm

- **# Messages:** 2 (N-1)
  - N-1 REQUESTS; N-1 REPLY
- **Synchronization Delay:** T
- **System Throughput:** 1/(T+E)

- **Optimization: Roucairol-Carvalho optimization**
- A process Pi that got the REPLY message from Pj for a prior CS access REQUEST does not need to send REQUEST(s) to Pj for subsequent CS accesses, until Pj sends a REQUEST message to access the CS.
  - The # messages for a CS invocation is in between 0 to 2*(N-1).
  - The optimization is achieved by spending an additional memory space of O(N) to keep track of the processes that have not yet sent a REQUEST message after sending their REPLY.

Ricart Agrawala's Distributed Mutual Exclusion Algorithm



Ricart Agrawala's Distributed Mutual Exclusion Algorithm

- **Safety:**
  - A process Pi can access the CS only after getting REPLY messages from all other processes.
  - A process Pj sends a REPLY for Pi's REQUEST, only if Pj is not executing or requesting access to the CS or if Pj's own REQUEST timestamp is greater than that of Pi's REQUEST
    - (in the latter case, Pi will defer sending a REPLY to Pj's REQUEST until it is done with its CS access).
- **Liveness:**
  - Upon completing its CS execution, a process sends out a REPLY message to all its deferred REQUEST messages. An idle process immediately sends out REPLY for a CS REQUEST. So, a process is guaranteed to get access to the CS by obtaining REPLY messages from all other processes.

Ricart Agrawala's Distributed Mutual Exclusion Algorithm

- **Fairness:**
  - If there are one or more deferred CS REQUESTS in its queue, upon completing its current CS execution, a process has to immediately send REPLY to all of the deferred CS REQUESTS.
  - Even if the process wants to access the CS again, it has to send out REQUEST messages to all the processes for which it has sent a REPLY.
  - A process that has already sent out CS REQUEST decides whether or not to defer a CS access REQUEST from a peer process based on the timestamp of the REQUEST from the peer process.
    - A process sends REPLY for REQUEST with a timestamp smaller than its own.
    - Hence, every process is guaranteed to get access to the CS in the order of the timestamps of the REQUESTs.

Source

https://www.youtube.com/watch?v=r7SJOhGF4Nc

THANK YOU

Ricart-Agrawala Algorithm

- The Ricart-Agrawala algorithm assumes the communication channels are FIFO. 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.
- Processes use Lamport-style logical clocks to assign a timestamp to critical section requests and timestamps are used to decide the priority of requests.
- Each process p_i maintains the Request-Deferred array, RD_i, the size of which is the same as the number of processes in the system.
- Initially, $\forall i\ \forall j$: RD_i[j]=0. Whenever p_i defer the request sent by p_j, it sets RD_i[j]=1 and after it has sent a REPLY message to p_j, it sets RD_i[j]=0.

Description of the 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.

Algorithm

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 send a REPLY message to S_j and set RD_i[j]=0.

Notes:

- When a site receives a message, it updates its clock using the timestamp in the message.
- When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.

Correctness

Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.
Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j' are executing the CS concurrently and S_i's request has higher priority than the request of S_j. Clearly, S_i received S_j's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j's request) before S_i exits the CS.
- However, this is impossible because S_j's request has lower priority.Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

Performance

- For each CS execution, Ricart-Agrawala algorithm requires $(N-1)$ REQUEST messages and $(N-1)$ REPLY messages.
- Thus, it requires $2(N-1)$ messages per CS execution.
- Synchronization delay in the algorithm is T.

Token-Based Algorithms

- In token-based algorithms, a unique token is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Token-based algorithms use sequence numbers instead of timestamps. (Used to distinguish between old and current requests.)

Suzuki-Kasami's Broadcast Algorithm

- If a site wants to enter the CS and it does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
- If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

continuation..

This algorithm must efficiently address the following two design issues:
(1) How to distinguish an outdated REQUEST message from a current REQUEST message:

- Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied.
- If a site can not determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it.
- This will not violate the correctness, however, this may seriously degrade the performance.

(2) How to determine which site has an outstanding request for the CS:

- After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them.

continuation..

The first issue is addressed in the following manner:

- A REQUEST message of site S_j has the form REQUEST(j, n) where n (n=1, 2, ...) is a sequence number which indicates that site S_j is requesting its n^{th} CS execution.
- A site S_i keeps an array of integers RN_i[1..N] where RN_i[j] denotes the largest sequence number received in a REQUEST message so far from site S_j.
- When site S_i receives a REQUEST(j, n) message, it sets RN_i[j]:= max(RN_i[j], n).
- When a site S_i receives a REQUEST(j, n) message, the request is outdated if RN_i[j]>n.

continuation..

The second issue is addressed in the following manner:

- The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site S_j executed most recently.
- After executing its CS, a site S_i updates LN[i]:=RN_i[i] to indicate that its request corresponding to sequence number RN_i[i] has been executed.
- At site S_i if RN_i[j]=LN[j]+1, then site S_j is currently requesting token.

The Algorithm

Requesting the critical section

(a) If 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 $RN_i[i]$.)

(b) 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

(c) Site S_i executes the CS after it has received the token.

The Algorithm

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

(d) It sets LN[i] element of the token array equal to RN_i[i].

(e) 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.

(f) If the token queue is nonempty after the above update, S_i deletes the top site id from the token queue and sends the token to the site indicated by the id.

Correctness

Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

Theorem: A requesting site enters the CS in finite time.

Proof:

- Token request messages of a site S_i reach other sites in finite time.
- Since one of these sites will have token in finite time, site S_i's request will be placed in the token queue in finite time.
- Since there can be at most $N - 1$ requests in front of this request in the token queue, site S_i will get the token and execute the CS in finite time.

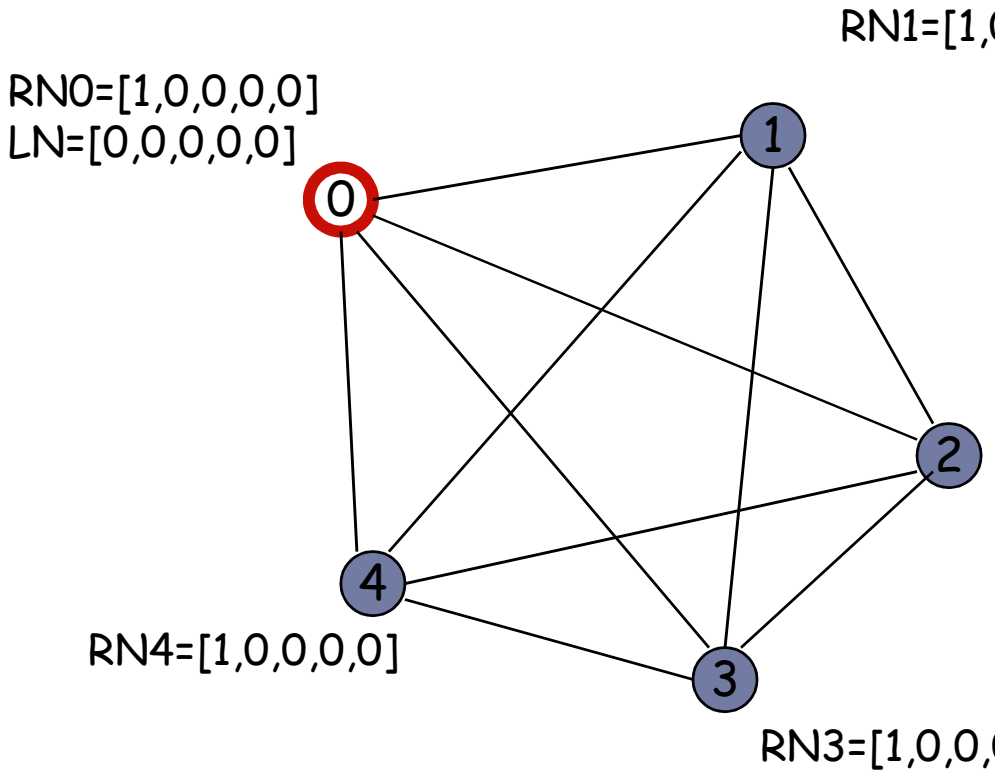
Performance

- No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.
- If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token. Synchronization delay in this algorithm is 0 or T.

Suzuki Kasami's Distributed Mutual Exclusion Algorithm

Reference : Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems, 5th Edition

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



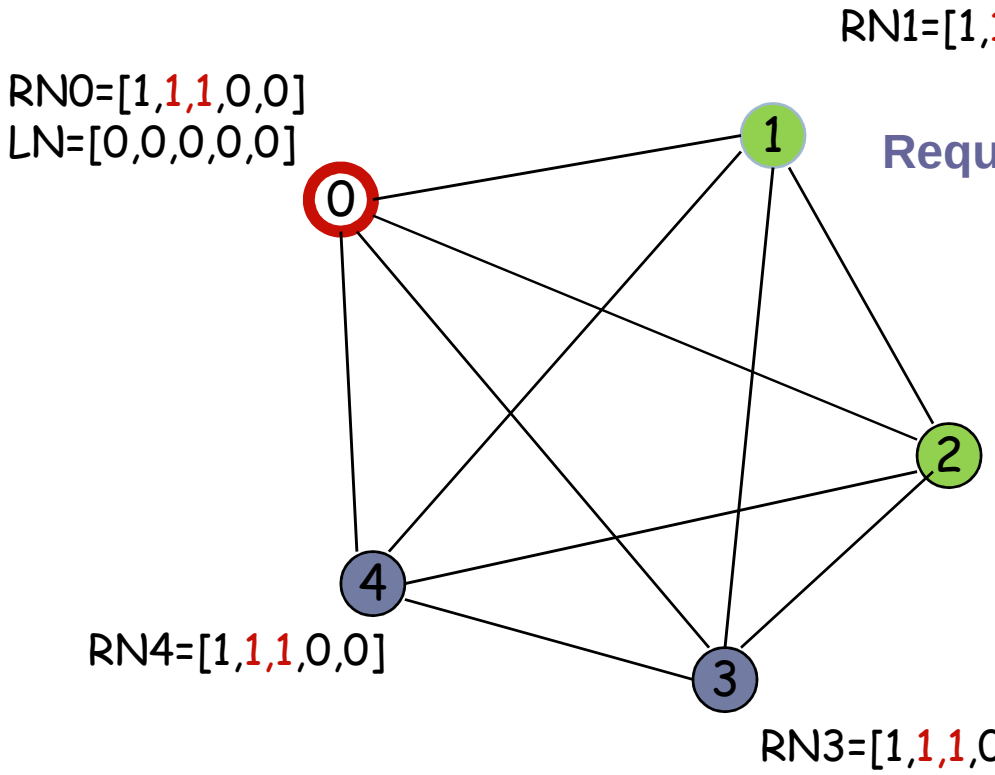
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

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

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



Request(Pid, sequence no.)

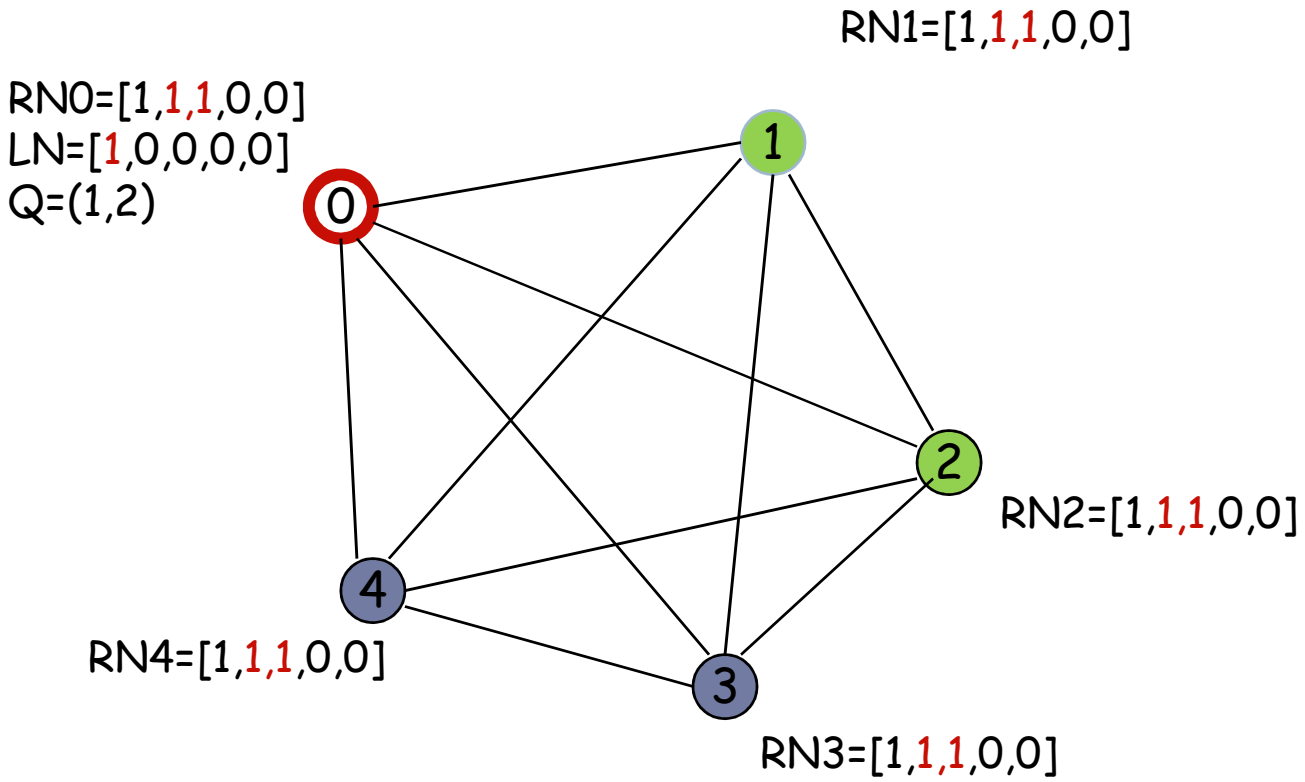
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

1 & 2 send requests to enter CS

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



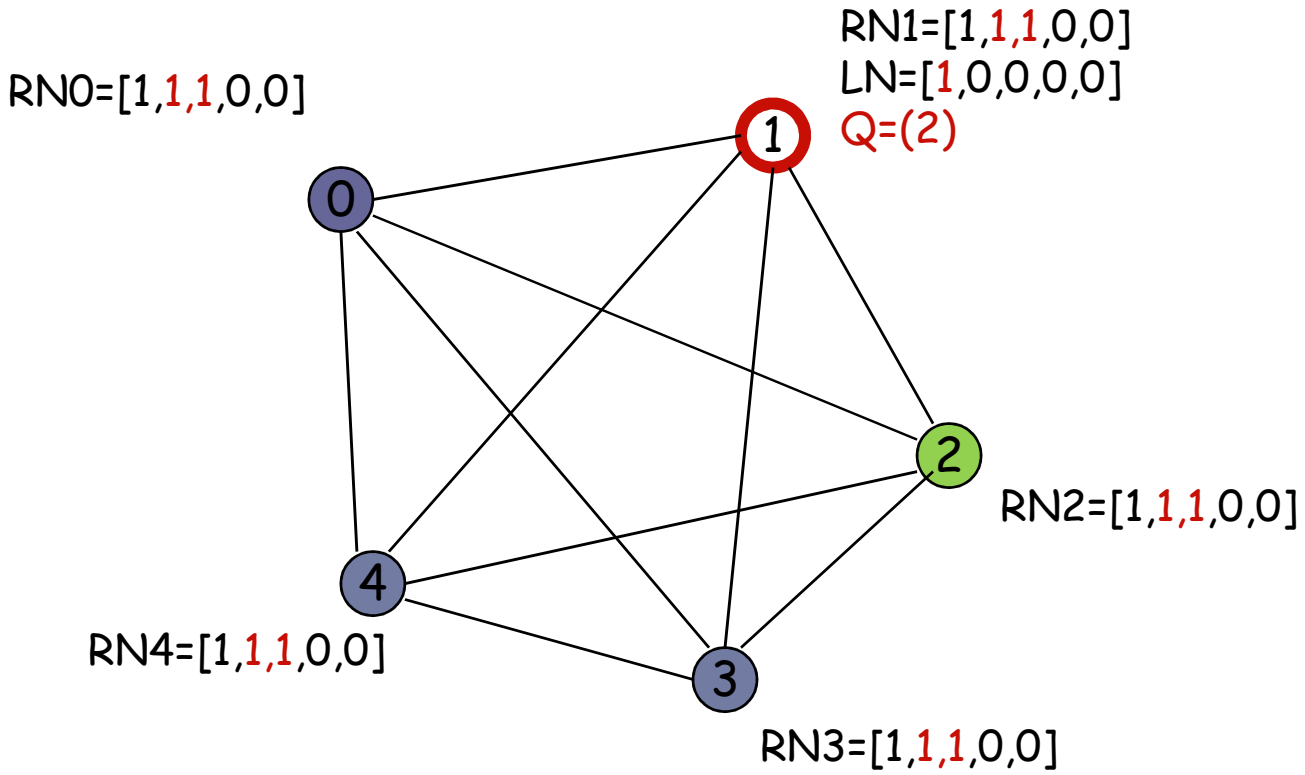
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

0 prepares to exit CS

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



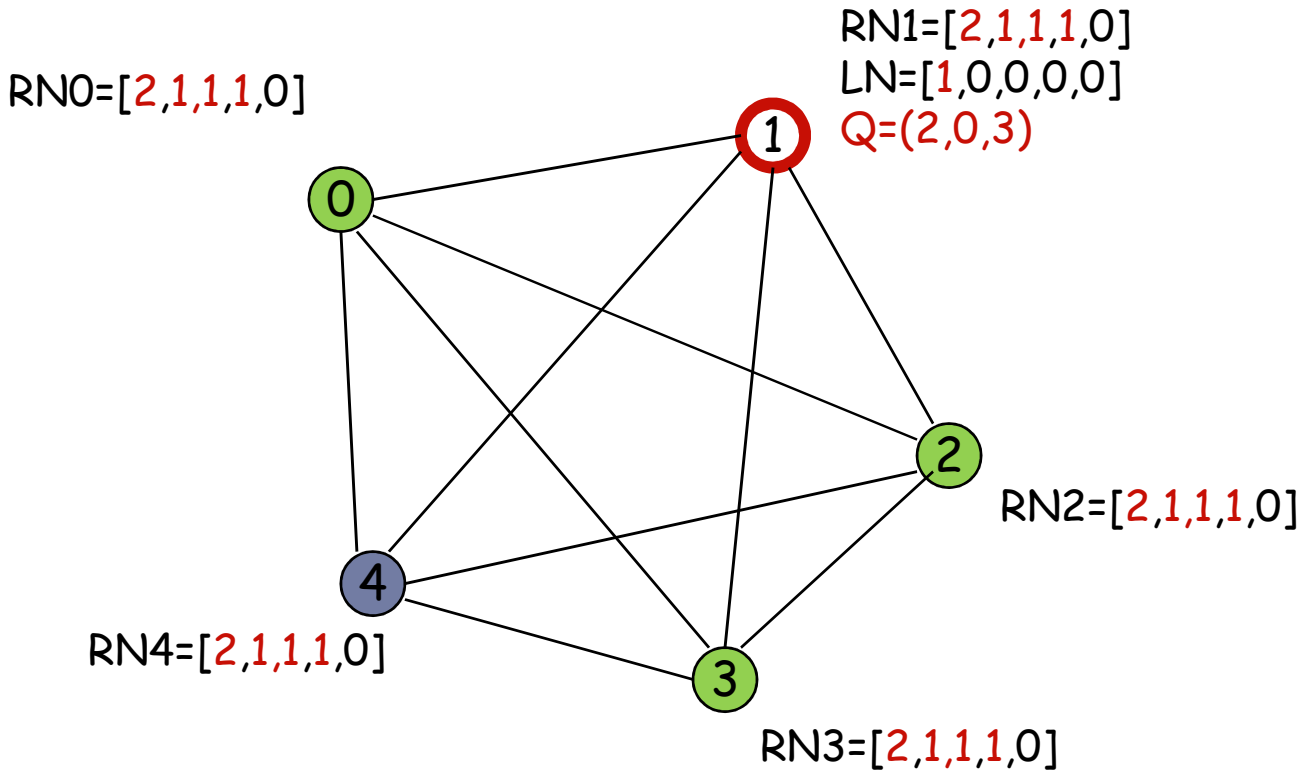
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

0 passes token (Q and last) to 1

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



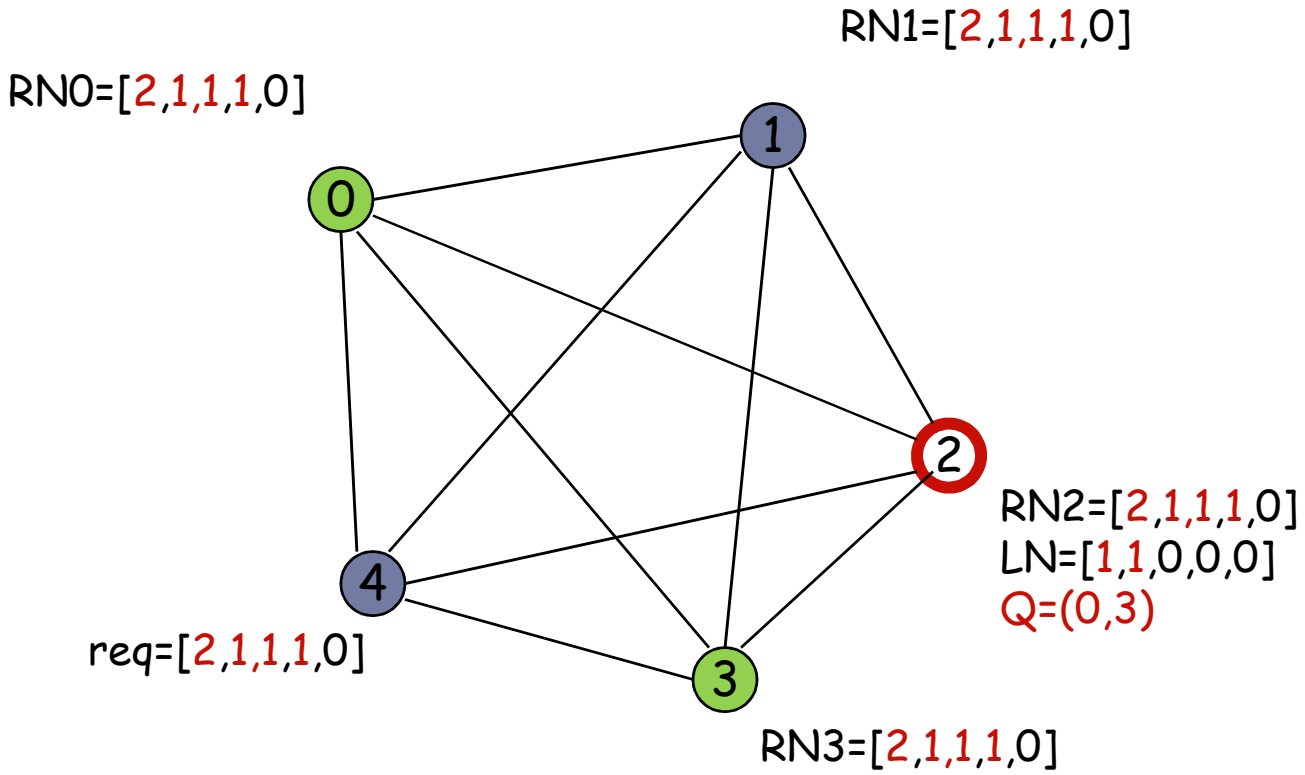
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

0 and 3 send requests

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



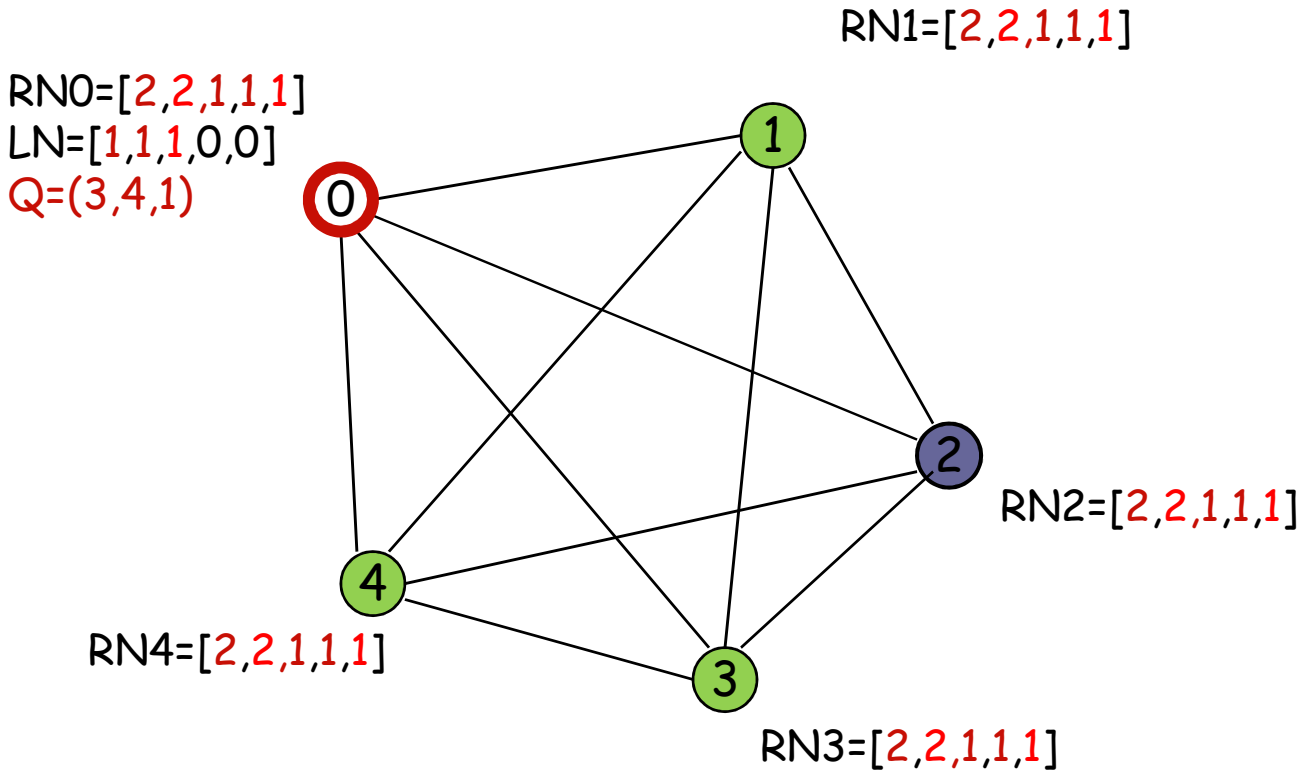
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

1 sends token to 2

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



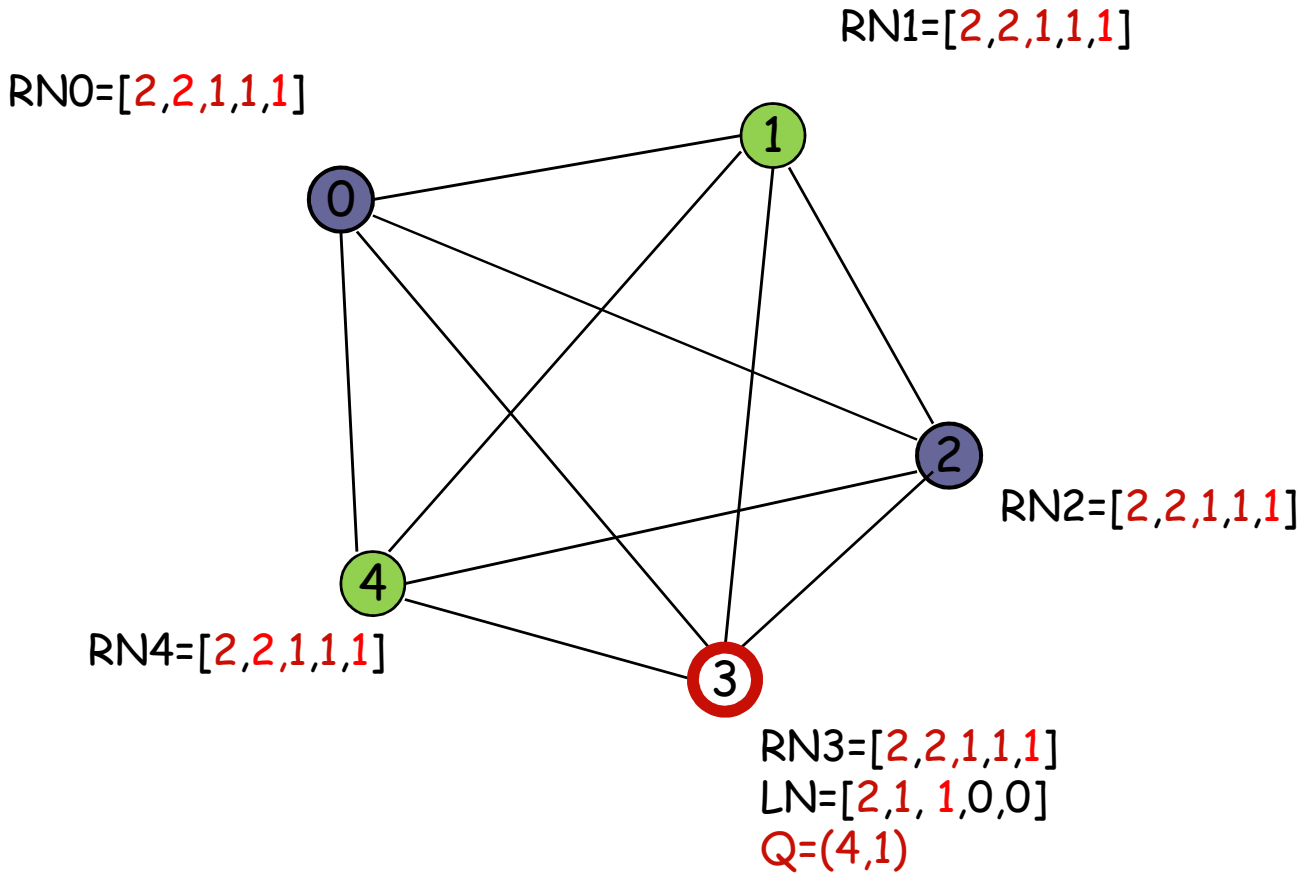
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

2 prepares to exit & sends token to 0
1 & 4 sends request.

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



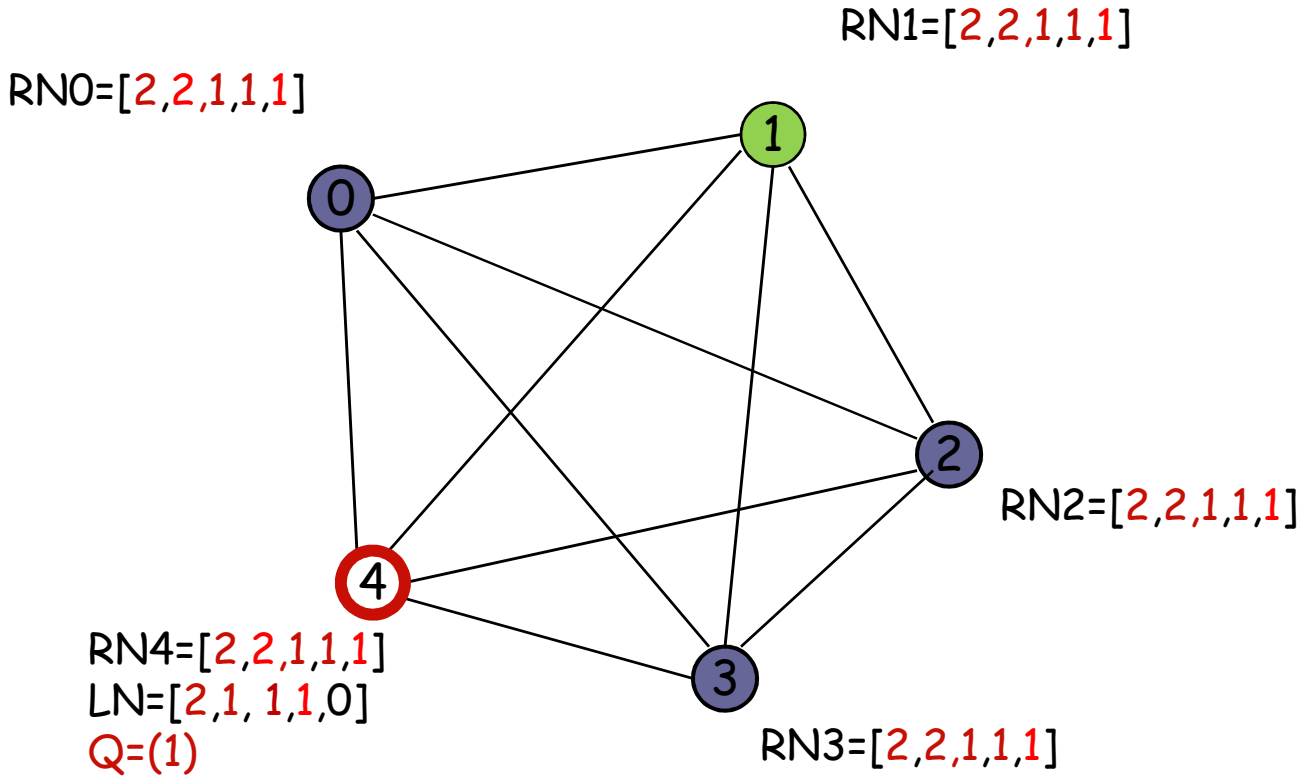
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

0 exits & sends token to 3

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



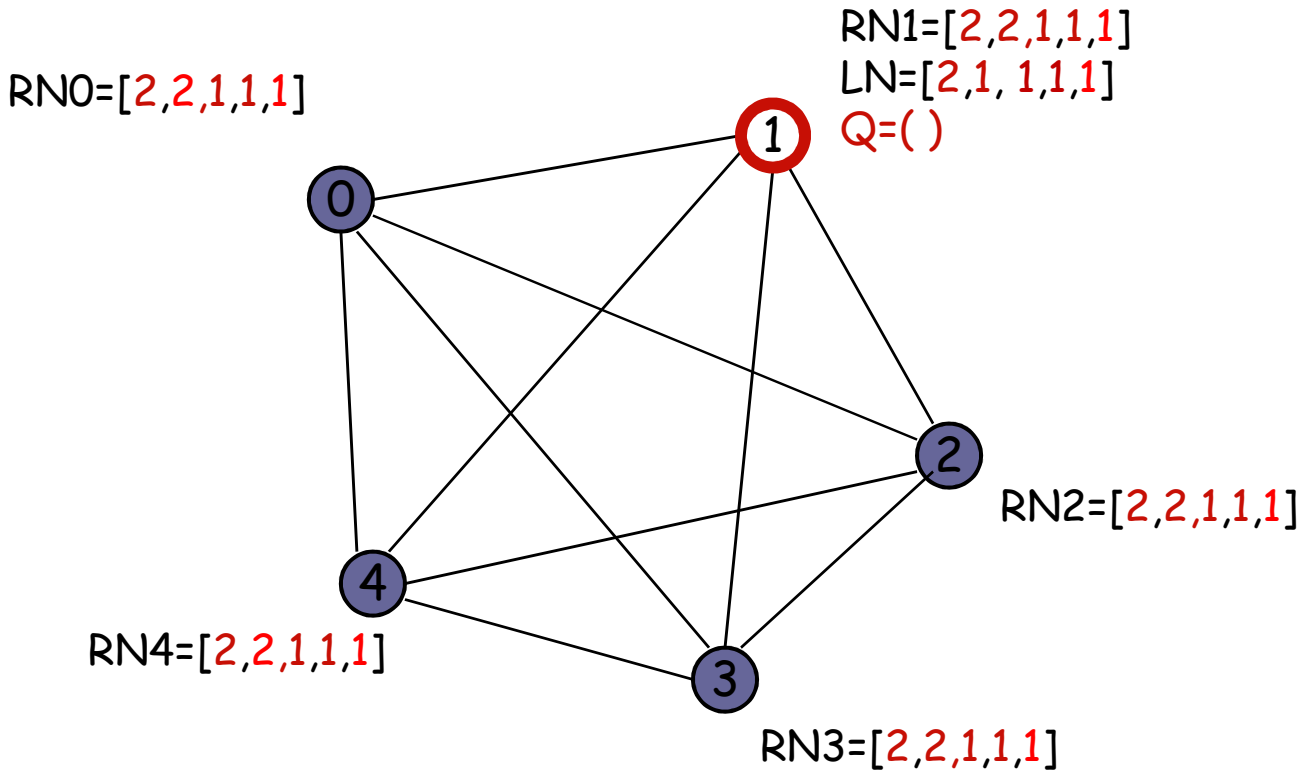
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

3 exits & sends token to 4

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



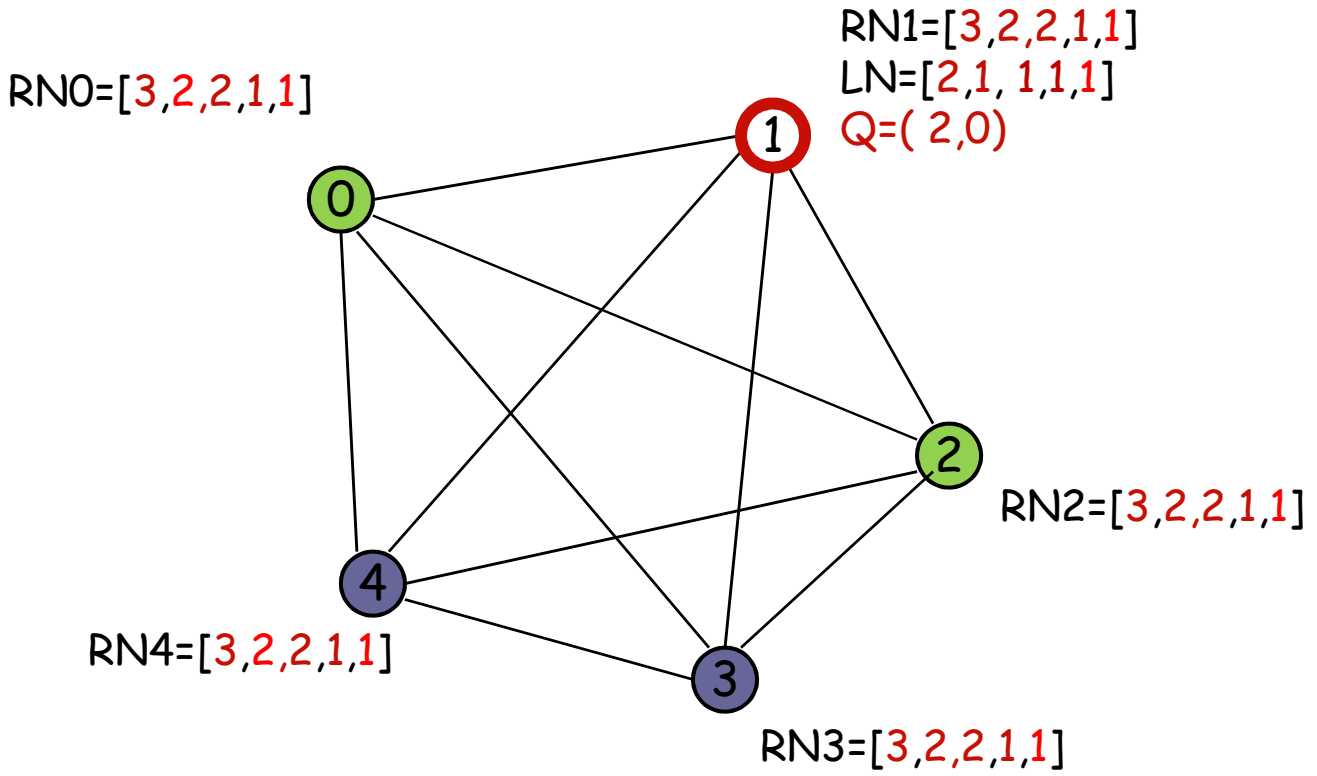
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

4 exits & sends token to 1

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



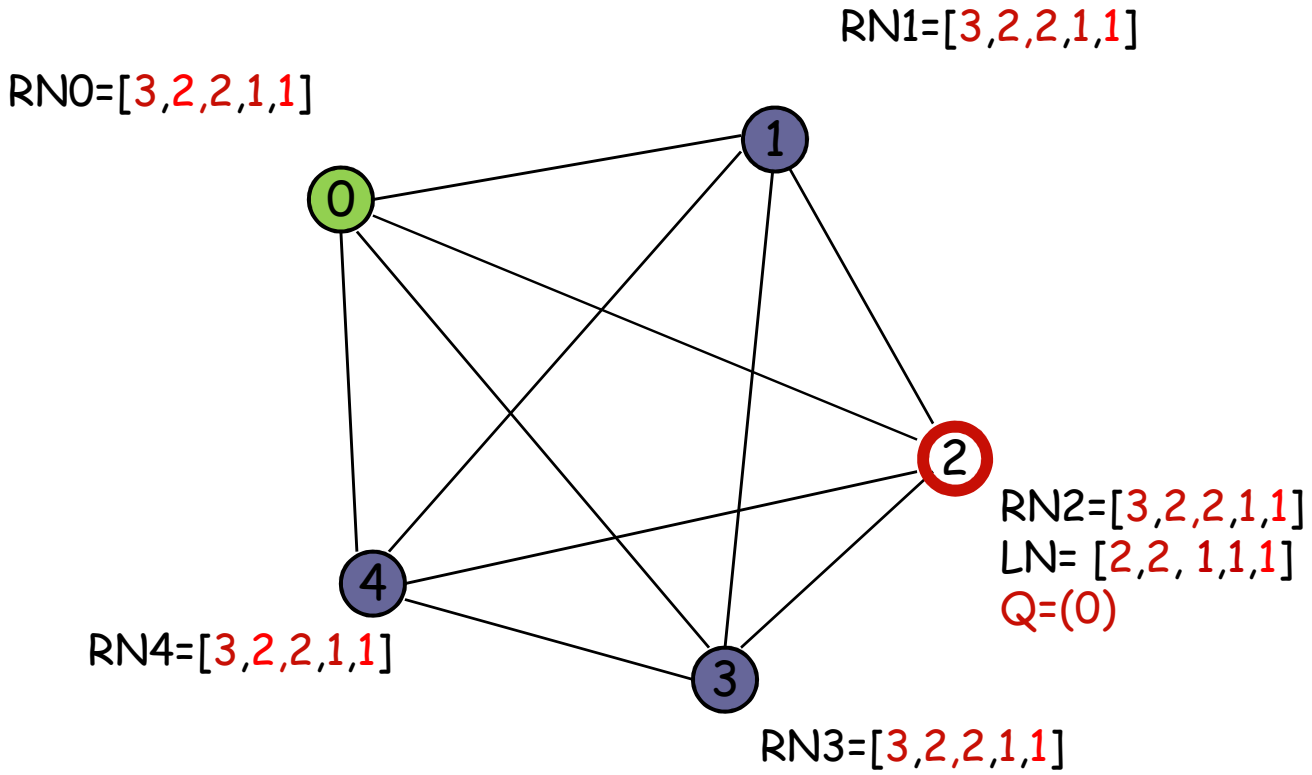
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

2 & 0 request for token

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



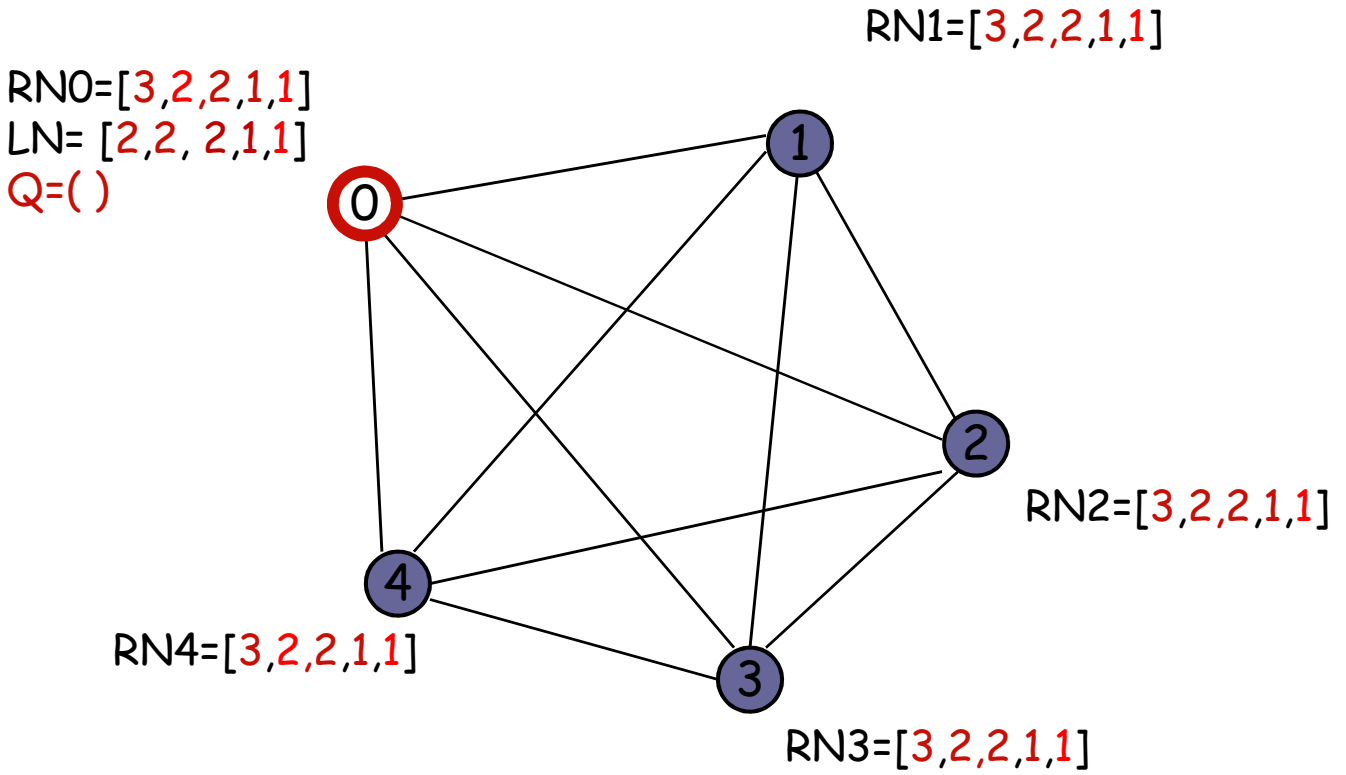
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

Token is given to 2
2 exits from CS and gives token to 0

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



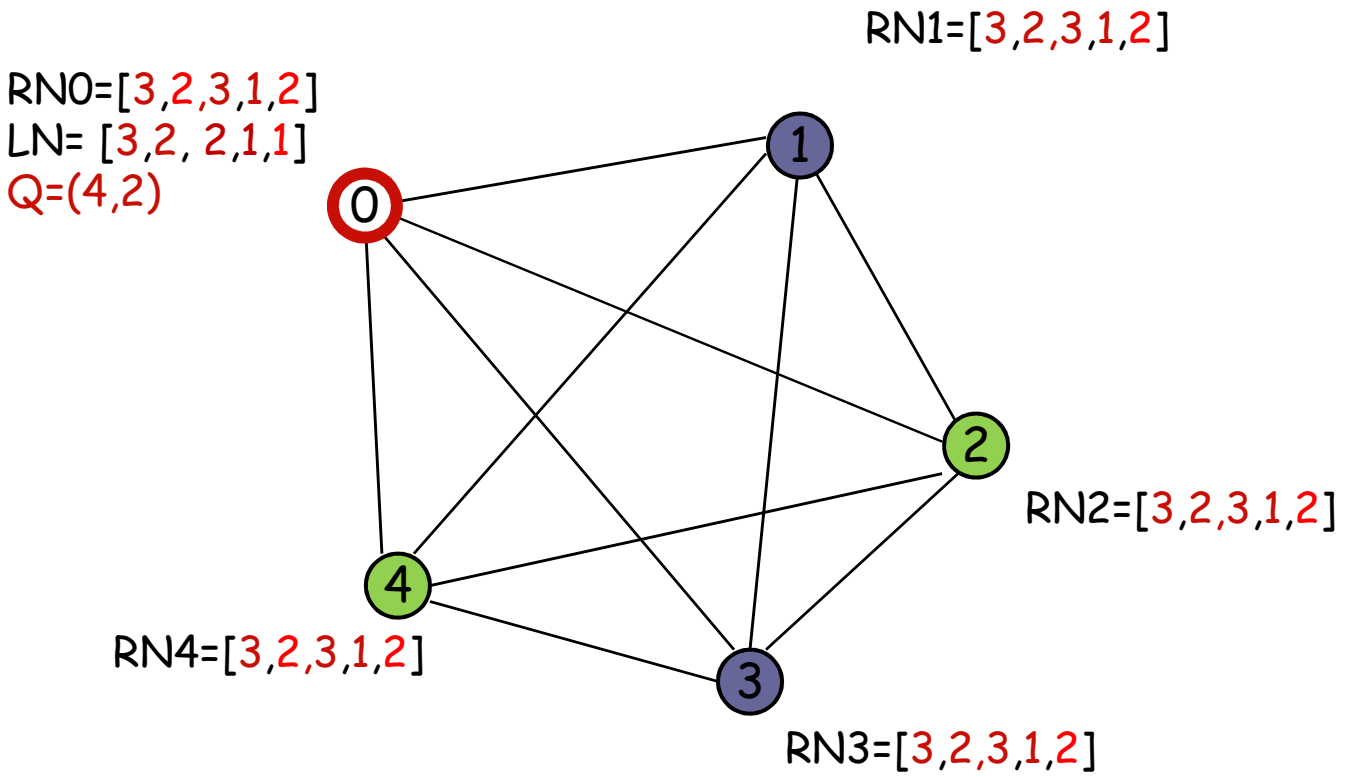
Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

0 gets the token, 0 executes in CS
While process 0 exiting from CS, it sets
LN[0] = RN0[0] i.e.LN= [3,2, 2,1,1]

Suzuki Kasami's Distributed Mutual Exclusion Algorithm



Requesting CS
RNi [i] = RNi [i] + 1
Send **REQUEST(i, sn)**
RNj [i] = max(RNj [i], sn).

Executing in CS
When process gets token, it uses shared variable.

Exiting from CS
Remove process from Token queue
Set **LN[i] = RNi [i].**
Give token to process which satisfies
RNi [j]=LN[j]+1.

4 & 2 are making request for CS

Thank You

Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms are different in the following two ways:

1. A site does not request permission from all other sites, but only from a subset of the sites. The request set of sites are chosen such that $\forall i\ \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \Phi$. Consequently, every pair of sites has a site which mediates conflicts between that pair.
2. A site can send out only one REPLY message at any time. A site can send a REPLY message only after it has received a RELEASE message for the previous REPLY message.

continuation..

Since these algorithms are based on the notion of 'Coteries' and 'Quorums', we next describe the idea of coteries and quorums.
A coterie C is defined as a set of sets, where each set $g \in C$ is called a quorum.
The following properties hold for quorums in a coterie:

- **Intersection property:** For every quorum g, h $\in$ C, g $\cap$ h$\neq \emptyset$.
  For example, sets $\{1,2,3\}$, $\{2,5,7\}$ and $\{5,7,9\}$ cannot be quorums in a coterie because the first and third sets do not have a common element.
- **Minimality property:** There should be no quorums g, h in coterie C such that g $\supseteq$ h. For example, sets $\{1,2,3\}$ and $\{1,3\}$ cannot be quorums in a coterie because the first set is a superset of the second.

continuation..

Coteries and quorums can be used to develop algorithms to ensure mutual exclusion in a distributed environment. A simple protocol works as follows:

- Let 'a' is a site in quorum 'A'. If 'a' wants to invoke mutual exclusion, it requests permission from all sites in its quorum 'A'.
- Every site does the same to invoke mutual exclusion. Due to the Intersection Property, quorum 'A' contains at least one site that is common to the quorum of every other site.
- These common sites send permission to only one site at any time. Thus, mutual exclusion is guaranteed.

Note that the Minimality property ensures efficiency rather than correctness.

Maekawa's Algorithm

Maekawa's algorithm was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$

M2: $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$

M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K)$

M4: Any site S_j is contained in K number of R_is, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that $N = K(K-1)+1$. This relation gives $|R_i| = \sqrt{N}$.

continuation..

- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm.
- Condition M3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion.
- Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have "equal responsibility" in granting permission to other sites.

The Algorithm

A site S_i executes the following steps to execute the CS.

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.

The Algorithm

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.

Correctness

Theorem: *Maekawa's algorithm achieves mutual exclusion.*
Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are concurrently executing the CS.
- This means site S_i received a REPLY message from all sites in R_i and concurrently site S_j was able to receive a REPLY message from all sites in R_j.
- If $R_i \cap R_j = \{S_k\}$, then site S_k must have sent REPLY messages to both S_i and S_j concurrently, which is a contradiction. $\square$

Performance

- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution.
- Synchronization delay in this algorithm is $2T$. This is because after a site S_i exits the CS, it first releases all the sites in R_i and then one of those sites sends a REPLY message to the next site that executes the CS.

Problem of Deadlocks

- Maekawa's algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.
- Assume three sites S_i, S_j, and S_k simultaneously invoke mutual exclusion.
- Suppose $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, and $R_k \cap R_i = \{S_{ki}\}$.
- Consider the following scenario:
  - S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}).
  - S_{jk} has been locked by S_j(forcing S_k to wait at S_{jk}).
  - S_{ki} has been locked by S_k(forcing S_i to wait at S_{ki}).
- This state represents a deadlock involving sites S_i, S_j, and S_k.

Handling Deadlocks

- Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock.
- A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.

Deadlock handling requires three types of messages:

FAILED: A FAILED message from site S_i to site S_j indicates that S_i can not grant S_j's request because it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from S_i to S_j indicates that S_i would like to find out from S_j if it has succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site S_i to S_j indicates that S_i is returning the permission to S_j (to yield to a higher priority request at S_j).

Handling Deadlocks

Maekawa's algorithm handles deadlocks as follows:

- When a REQUEST(ts, i) from site S_i blocks at site S_j because S_j has currently granted permission to site S_k, then S_j sends a FAILED(j) message to S_i if S_i's request has lower priority. Otherwise, S_j sends an INQUIRE(j) message to site S_k.
- In response to an INQUIRE(j) message from site S_j, site S_k sends a YIELD(k) message to S_j provided S_k has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new GRANT from it.
- In response to a YIELD(k) message from site S_k, site S_j assumes as if it has been released by S_k, places the request of S_k at appropriate location in the request queue, and sends a GRANT(j) to the top request's site in the queue. Maekawa's algorithm requires extra messages to handle deadlocks
- Maximum number of messages required per CS execution in this case is $5\sqrt{N}$.

Maekawa's Distributed Mutex Algorithm

Reference: 1. Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems,

2. George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012

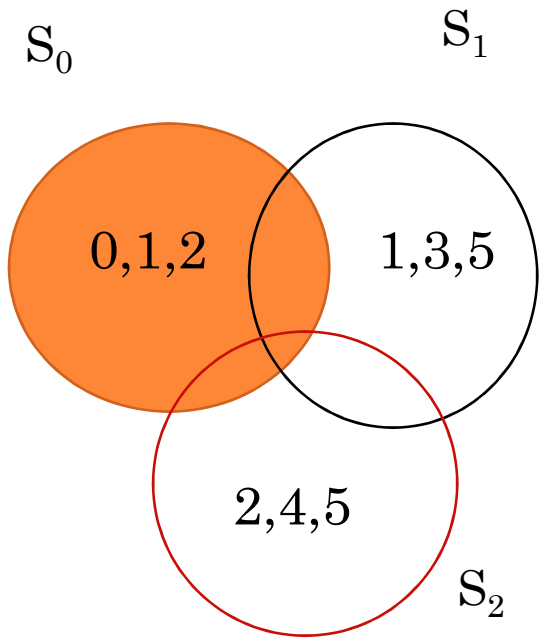
MAEKAWA'S ALGORITHM

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

  $\mathbf{i \in S_i}$

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

- **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

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

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

Version 1 {Life of process I}

1. Send timestamped ***request*** to each process in S_i.
2. Request received → send ***reply*** 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 *reply* 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 reply 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-VERSION 1

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

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

$S_i \cap S_j \neq \emptyset$ implies there is a process $k \in S_i \cap S_j$

Process **k** will **never** send reply 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-VERSION 1

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

From S_0= **{0,1,2}**, **0,2** send *reply* to **0**, but **1** sends *reply* to **1**;

From S_1= **{1,3,5}**, **1,3** send *reply* to **1**, but **5** sends *reply* to **2**;

From S_2= **{2,4,5}**, **4,5** send *reply* to **2**, but **2** sends *reply* 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!

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-VERSION 2

Avoiding deadlock

If processes always receive messages **in increasing order of timestamp**, then deadlock "could be" avoided. But this is too strong an assumption.

Version 2 uses three ***additional*** messages:

- ***failed***
- ***inquire***
- ***yield***

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-VERSION 2

New features in version 2

- Send ***reply*** and set **lock** as usual.
- If **lock is set** and a request with a larger timestamp arrives, send ***failed*** *(you have no chance)*. If the incoming request has a lower timestamp, then send ***inquire*** *(are you in CS?) to the locked process.*
- Receive ***inquire*** and at least one ***failed*** message → send ***yield***. The recipient resets the lock.

$S_0 =$ {0, 1, 2}

$S_1 =$ {1, 3, 5}

$S_2 =$ {2, 4, 5}

$S_3 =$ {0, 3, 4}

$S_4 =$ {1, 4, 6}

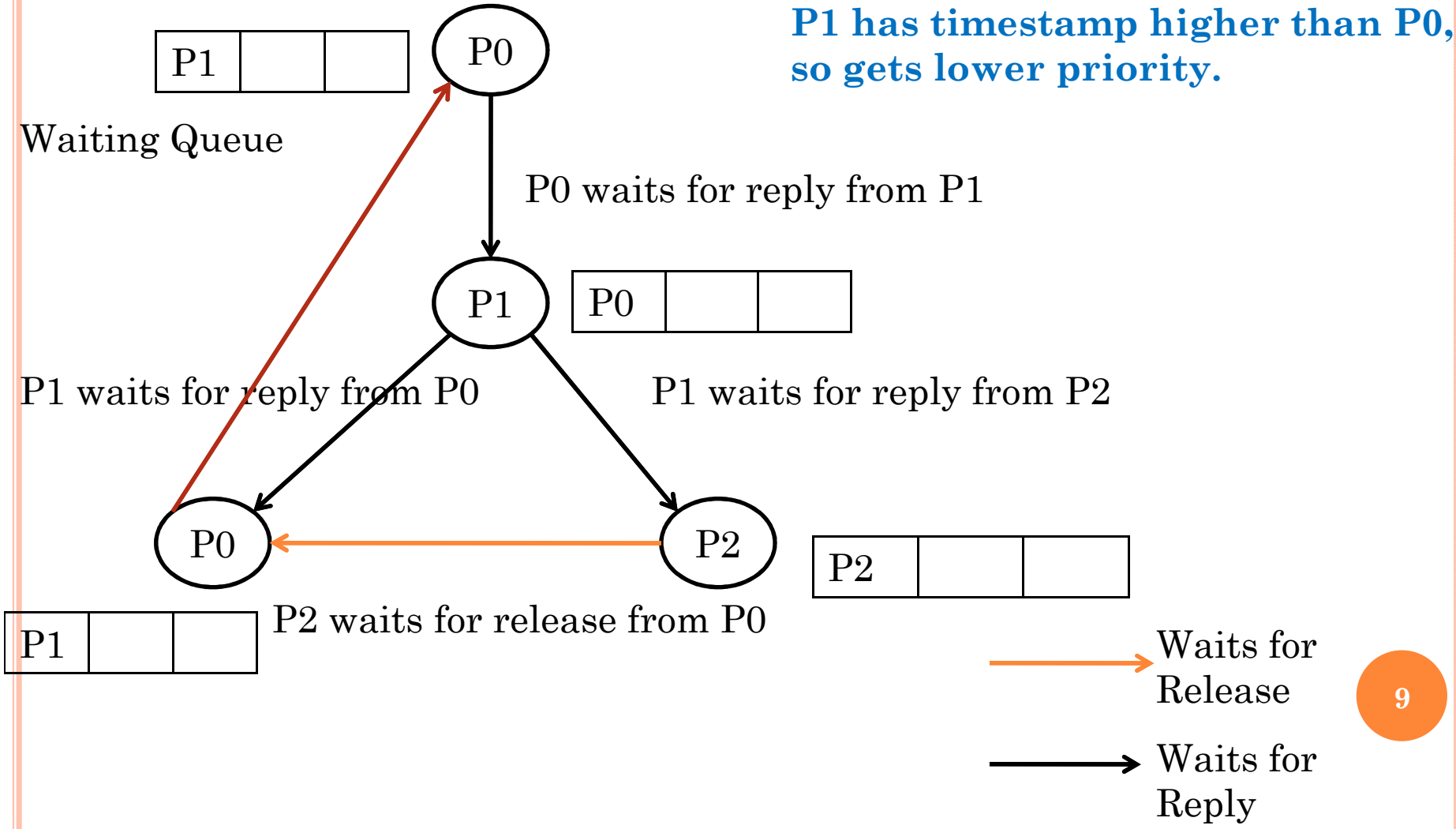
$S_5 =$ {0, 5, 6}

$S_6 =$ {2, 3, 6}

Resolving Deadlock

P0 has lower timestamp so gets higher priority.

P1 has timestamp higher than P0, so gets lower priority.



MAEKAWA'S ALGORITHM-VERSION 1

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

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!

low priority **high priority**

i **j** **k**

P1 blocks → P2 blocks→ P0 (P0 locked by P1)

P2 sends failed to P1 ←

P2 sends Inquire to P0 →

P0 sends yield to P2 ←

P2 sends Grant to P0 →

(P0 after receiving grant from P2, P0 replies to its waiting set of process with high priority, say here it is to P1)

S_0= **{0, 1, 2}**

S_1= **{1, 3, 5}**

S_2= **{2, 4, 5}**

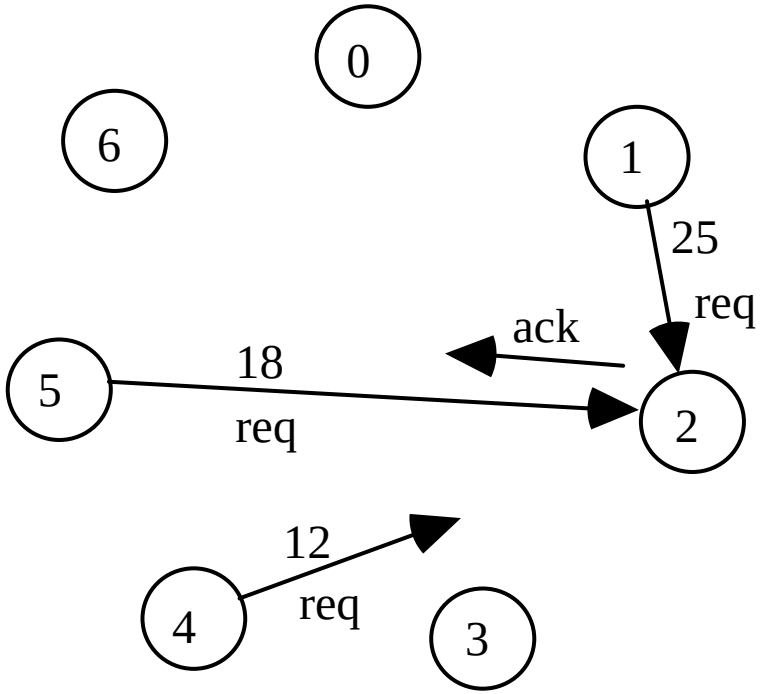
S_3= **{0, 3, 4}**

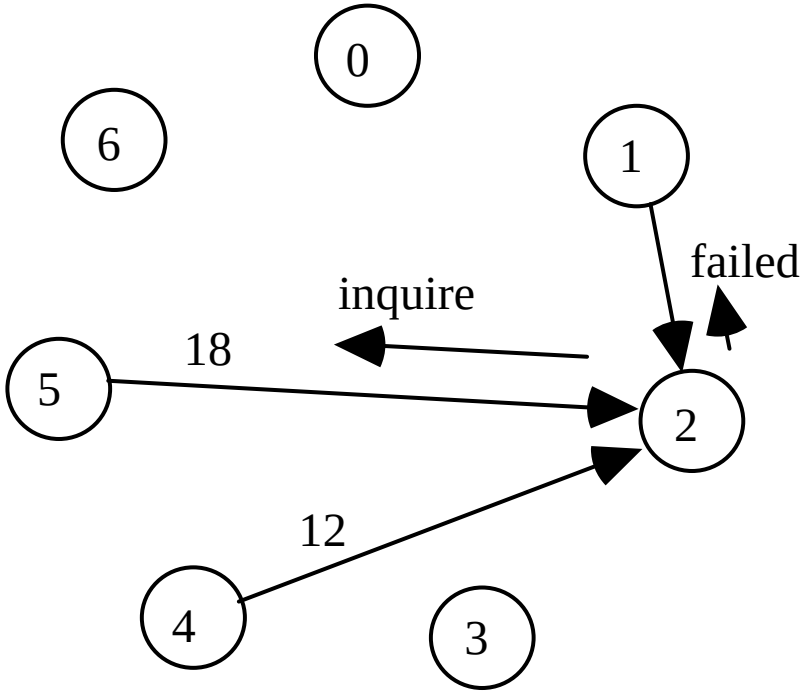
S_4= **{1, 4, 6}**

S_5= **{0, 5, 6}**

S_6= **{2, 3, 6}**

MAEKAWA'S ALGORITHM-VERSION 2





MAEKAWA'S ALGORITHM

- state = Released, voted = false
- enter() at process P*i*:
  - state = Wanted
  - Multicast Request message to all processes in V*i*
  - Wait for Reply (vote) messages from all processes in V*i* (including vote from self)
  - state = Held
- exit() at process P*i*:
  - state = Released
  - Multicast Release to all processes in V*i*

MAEKAWA'S ALGORITHM

- When P*i* receives a Request from P*j*:

if (state == Held OR voted = true)
    queue Request
else
    send Reply to P*j* and set voted = true

- When P*i* receives a Release from P*j*:

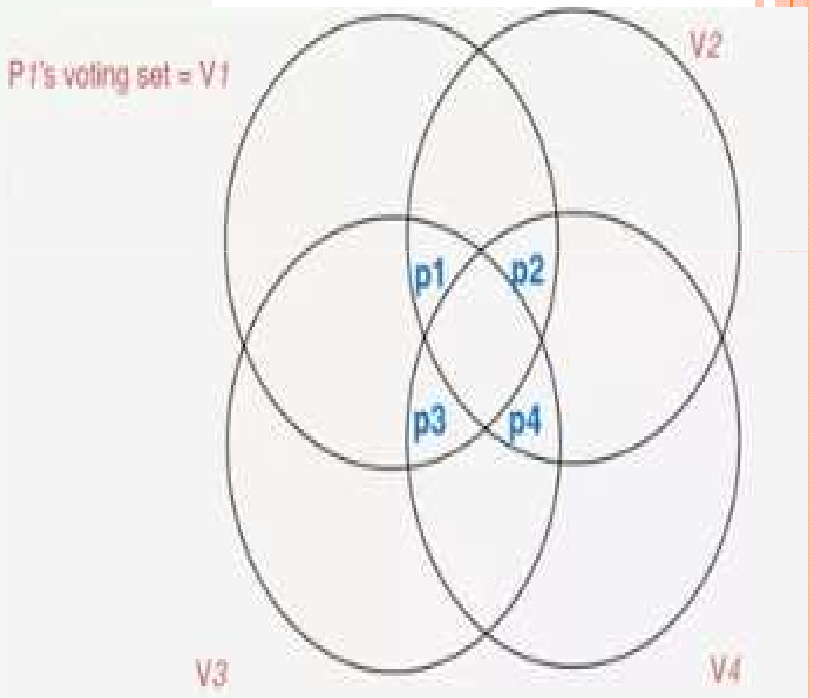
if (queue empty)
    voted = false
else
    dequeue head of queue, say P*k*
    Send Reply *only* to P*k*
    voted = true

SAFETY

- When a process P*i* receives replies from all its voting set V*i* members, no other process P*j* could have received replies from all its voting set members V*j*
  - V*i* and V*j* intersect in at least one process say P*k*
  - But P*k* sends only one Reply (vote) at a time, so it could not have voted for both P*i* and P*j*

LIVENESS

- A process needs to wait for at most (N-1) other processes to finish CS
- But does not guarantee liveness
- Since can have a *deadlock*
- Example: all 4 processes need access
  - P1 is waiting for P3
  - P3 is waiting for P4
  - P4 is waiting for P2
  - P2 is waiting for P1
  - No progress in the system!
- There are deadlock-free versions



Performance of Maekawa's Algorithm

- Bandwidth
  - $2\sqrt{N}$ messages per enter()
  - $\sqrt{N}$ messages per exit()
  - Better than Ricart and Agrawala's ($2*(N-1)$ and $N-1$ messages)
  - $\sqrt{N}$ quite small. N ~ 1 million => $\sqrt{N}$ = 1K
- Client delay: One round trip time
- Synchronization delay: 2 message transmission times

Deadlock Detection in Distributed Systems

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Chapter 10

Introduction

- Deadlocks is a fundamental problem in distributed systems.
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.

System Model

- A distributed program is composed of a set of n asynchronous processes p_1, p_2, . . . , p_i, . . . , p_n that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.

- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.
- We make the following assumptions:
  - The systems have only reusable resources.
  - Processes are allowed to make only exclusive access to resources.
  - There is only one copy of each resource.

- A process can be in two states: *running* or *blocked*.
- In the running state (also called *active* state), a process has all the needed resources and is either executing or is ready for execution.
- In the blocked state, a process is waiting to acquire some resource.

Wait-For-Graph (WFG)

- The state of the system can be modeled by directed graph, called a *wait for graph* (WFG).
- In a WFG , nodes are processes and there is a directed edge from node P_1 to mode P_2 if P_1 is blocked and is waiting for P_2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

- Figure 1 shows a WFG, where process P_{11} of site 1 has an edge to process P_{21} of site 1 and P_{32} of site 2 is waiting for a resource which is currently held by process P_{21}.
- At the same time process P_{32} is waiting on process P_{33} to release a resource.
- If P_{21} is waiting on process P_{11}, then processes P_{11}, P_{32} and P_{21} form a cycle and all the four processes are involved in a deadlock depending upon the request model.

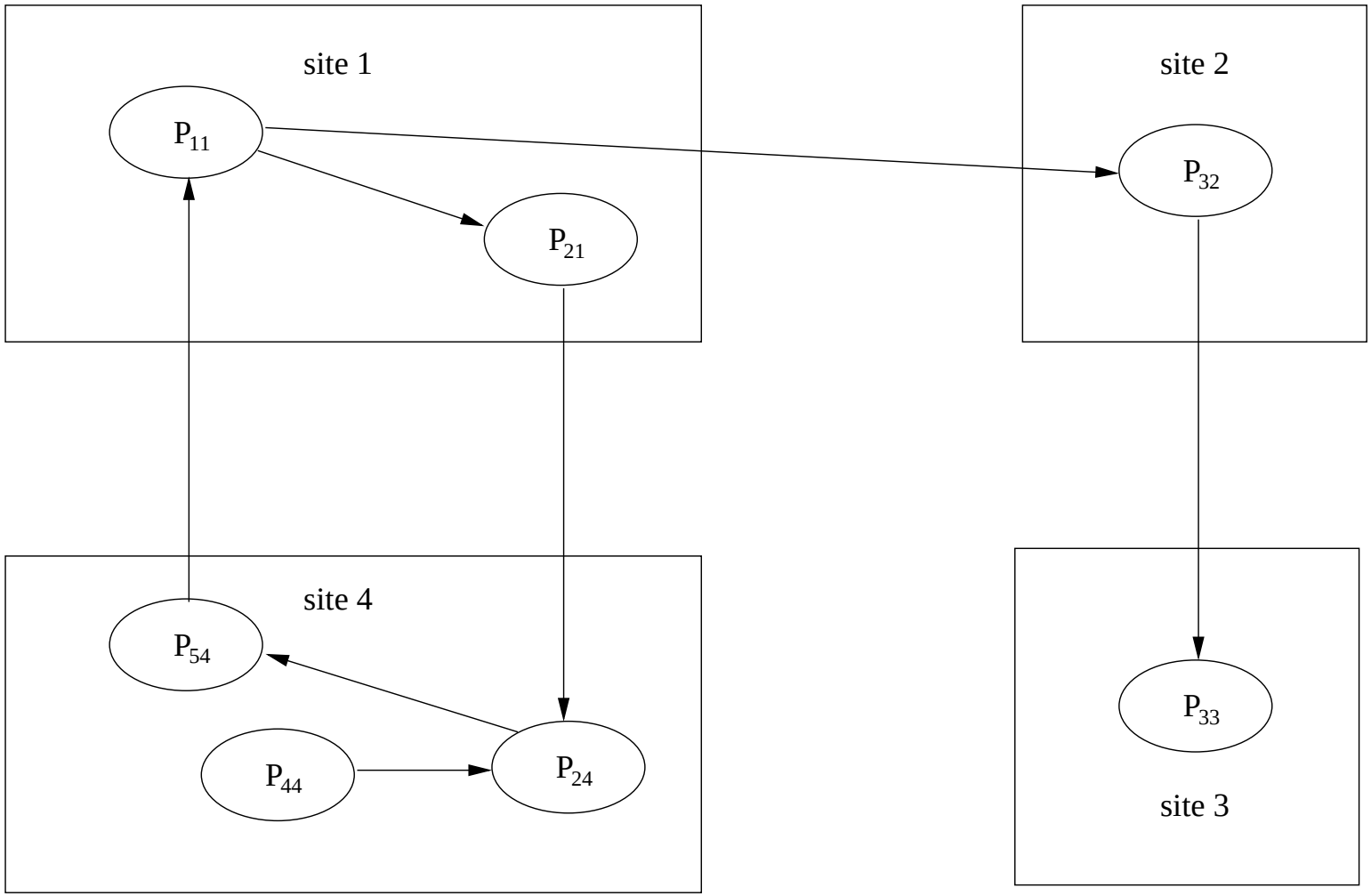


Figure 1: An Example of a WFG

Deadlock Handling Strategies

- There are three strategies for handling deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection.
- Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay.
- Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.
- This approach is highly inefficient and impractical in distributed systems.

- In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system).
- However, due to several problems, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires examination of the status of process-resource interactions for presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

Issues in Deadlock Detection

- Deadlock handling using the approach of deadlock detection entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.
- Detection of deadlocks involves addressing two issues: Maintenance of the WFG and searching of the WFG for the presence of cycles (or knots).

Correctness Criteria: A deadlock detection algorithm must satisfy the following two conditions:

(i) Progress (No undetected deadlocks):

- The algorithm must detect all existing deadlocks in finite time.
- In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

(ii) Safety (No false deadlocks):

- The algorithm should not report deadlocks which do not exist (called *phantom or false* deadlocks).

Resolution of a Detected Deadlock

- Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock.
- It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution.

Models of Deadlocks

Distributed systems allow several kinds of resource requests.

The Single Resource Model

- In the single resource model, a process can have at most one outstanding request for only one unit of a resource.
- Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

The AND Model

- In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

- Consider the example WFG described in the Figure 1.
- P_{11} has two outstanding resource requests. In case of the AND model, P_{11}shall become active from idle state only after both the resources are granted.
- There is a cycle P_{11}->P_{21}->P_{24}->P_{54}->P_{11} which corresponds to a deadlock situation.
- That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process P_{44} in Figure 1.
- It is not a part of any cycle but is still deadlocked as it is dependent on P_{24}which is deadlocked.

The OR Model

- In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- Consider example in Figure 1: If all nodes are OR nodes, then process P_{11} is not deadlocked because once process P_{33} releases its resources, P_{32} shall become active as one of its requests is satisfied.
- After P_{32} finishes execution and releases its resources, process P_{11} can continue with its processing.
- In the OR model, the presence of a knot indicates a deadlock.

The AND-OR Model

- A generalization of the previous two models (OR model and AND model) is the AND-OR model.
- In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request.
- For example, in the AND-OR model, a request for multiple resources can be of the form x *and* (y *or* z).
- To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.
- Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

The $\binom{p}{q}$ Model

- The $\binom{p}{q}$ model (called the P-out-of-Q model) allows a request to obtain any k available resources from a pool of n resources.
- It has the same in expressive power as the AND-OR model.
- However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request.
- Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as $\binom{p}{p}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

Unrestricted Model

- In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests.
- Only one assumption that the deadlock is stable is made and hence it is the most general model.
- This model helps separate concerns: Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).

Knapp's Classification

Distributed deadlock detection algorithms can be divided into four classes:

- path-pushing
- edge-chasing
- diffusion computation
- global state detection.

Path-Pushing Algorithms

- In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG.
- The basic idea is to build a global WFG for each site of the distributed system.
- In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites.
- After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.
- This feature of sending around the paths of global WFG has led to the term path-pushing algorithms.

Edge-Chasing Algorithms

- In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is be verified by propagating special messages called probes, along the edges of the graph.
- These probe messages are different than the request and reply messages.
- The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
- Whenever a process that is executing receives a probe message, it discards this message and continues.
- Only blocked processes propagate probe messages along their outgoing edges.
- Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.

Diffusing Computations Based Algorithms

- In *diffusion computation* based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system.
- These algorithms make use of echo algorithms to detect deadlocks.
- This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock.
- To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
- These queries are successively propagated (i.e., diffused) through the edges of the WFG.

- When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent.
- For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message.
- The initiator of a deadlock detection detects a deadlock when it receives reply for every query it had sent out.

Global State Detection Based Algorithms

- Global state detection based deadlock detection algorithms exploit the following facts:
  1. A consistent snapshot of a distributed system can be obtained without freezing the underlying computation and
  2. If a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.
- Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock.

Mitchell and Merritt's Algorithm for the Single-Resource Model

- Belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG.
- When a probe initiated by a process comes back to it, the process declares deadlock.
- Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock.

- Each node of the WFG has two local variables, called labels:
  1. a private label, which is unique to the node at all times, though it is not constant, and
  2. a public label, which can be read by other processes and which may not be unique.
- Each process is represented as u/v where u and u are the public and private labels, respectively.
- Initially, private and public labels are equal for each process.
- A global WFG is maintained and it defines the entire state of the system.

- The algorithm is defined by the four state transitions shown in Figure 2, where z = inc(u, v), and inc(u, v) yields a unique label greater than both u and v labels that are not shown do not change.
- Block creates an edge in the WFG.
- Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for.
- Activate denotes that a process has acquired the resource from the process it was waiting for.
- Transmit propagates larger labels in the opposite direction of the edges by sending a probe message.

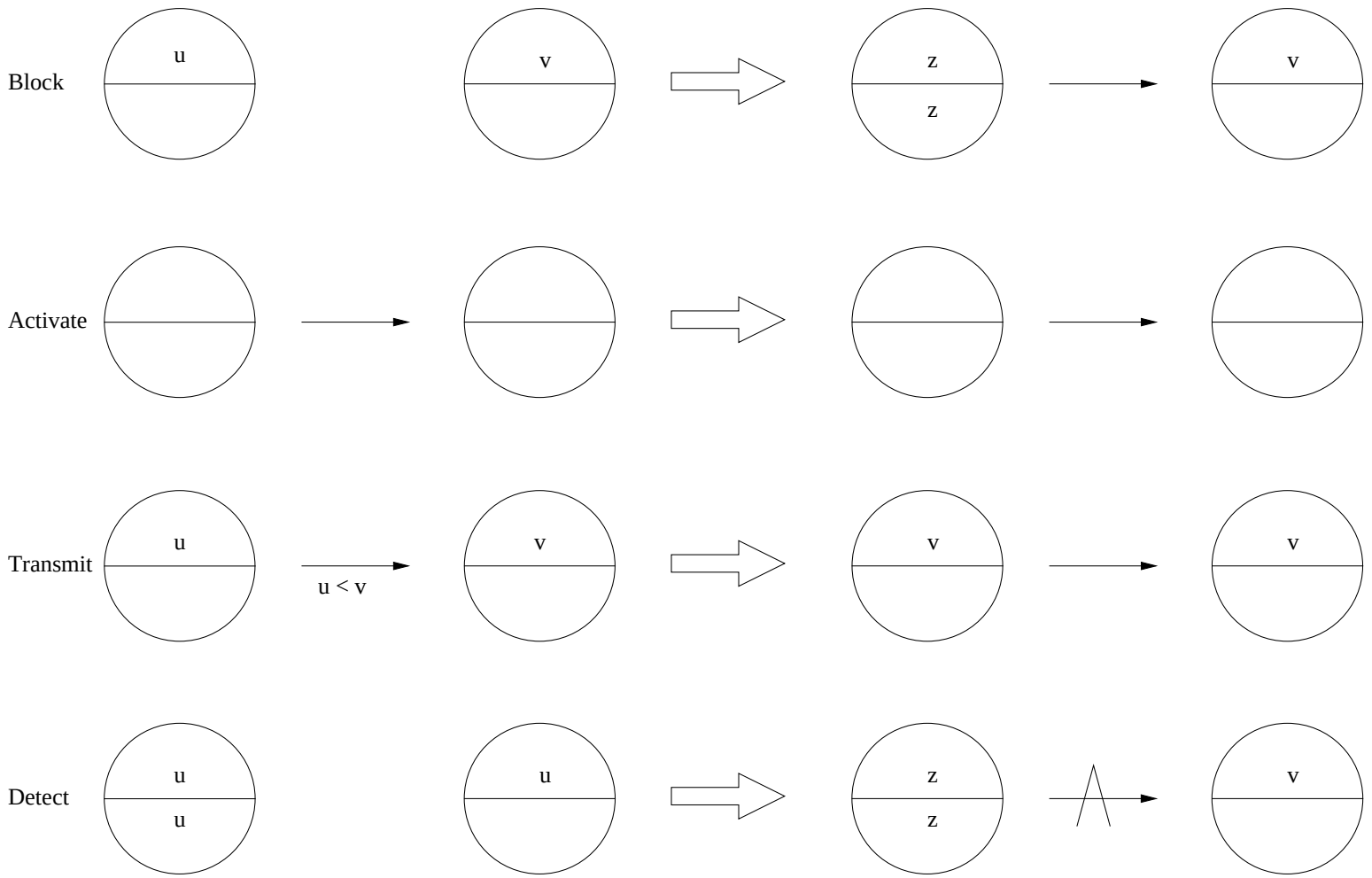


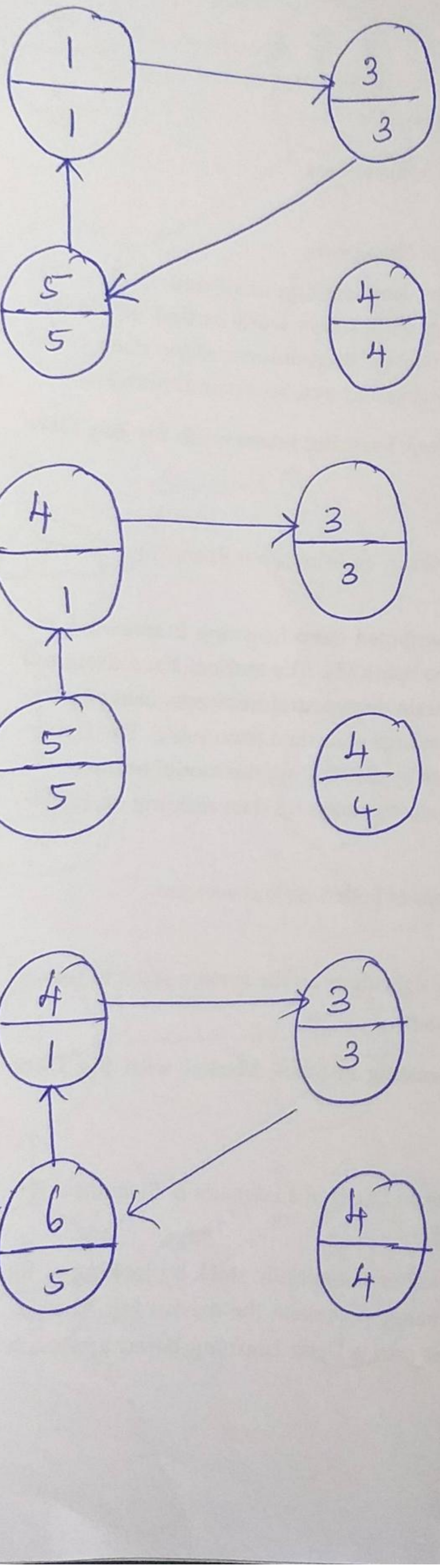
Figure 2: The four possible state transitions

- Whenever a process receives a probe which is less then its public label, then it simply ignores that probe.
- Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
- The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted.

Message Complexity:

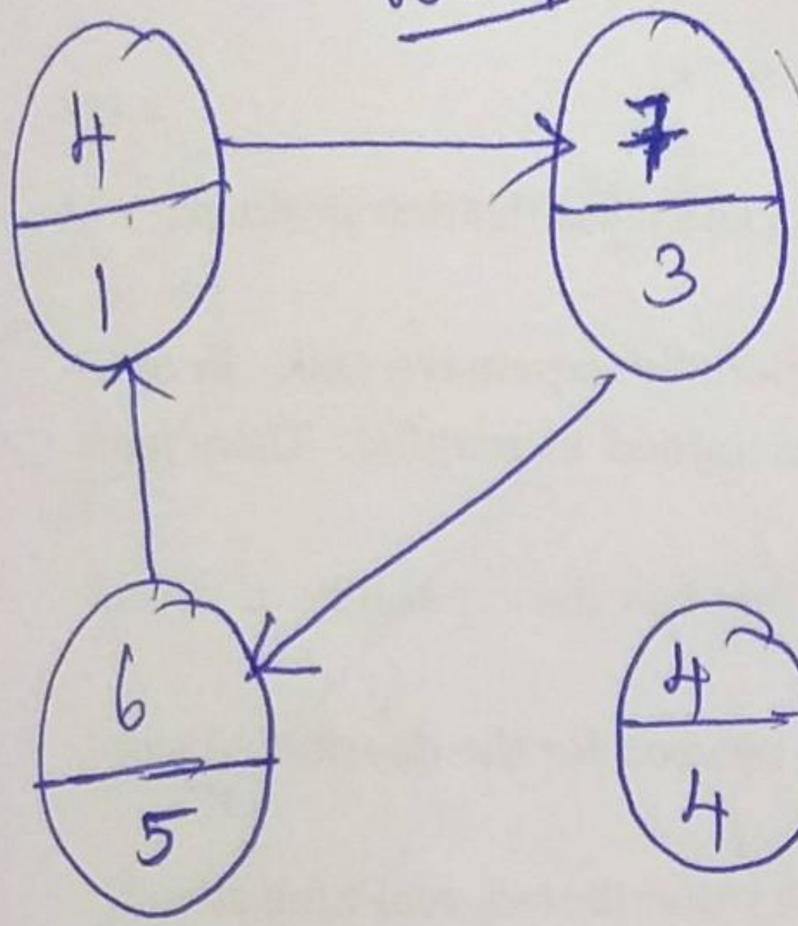
If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is s(s - 1)/2 Transmit steps, where s is the number of processes in the cycle.

Mitchell & Merritt Algo. for Single Resource Model.



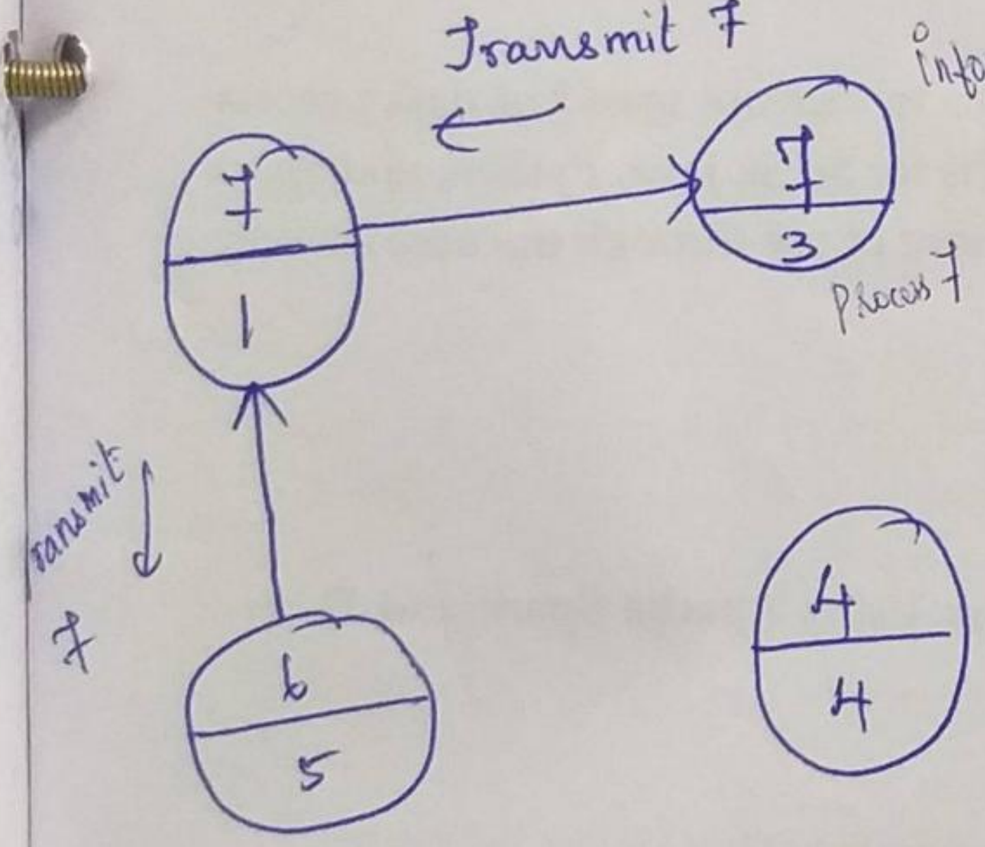
Public 'U'
Private 'V'

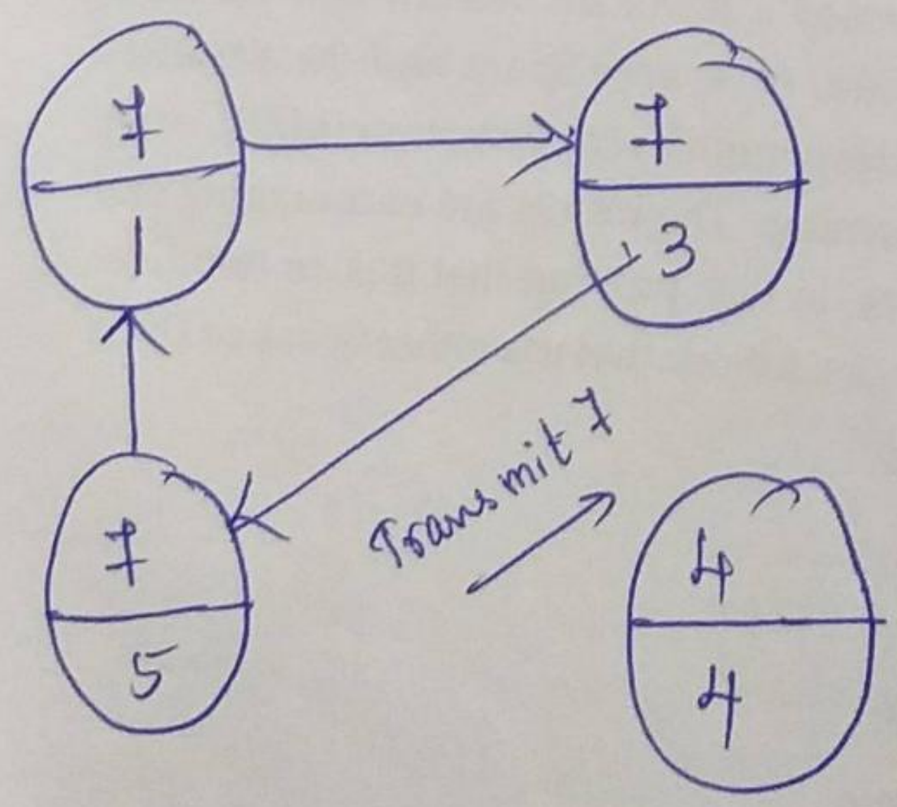
WFG

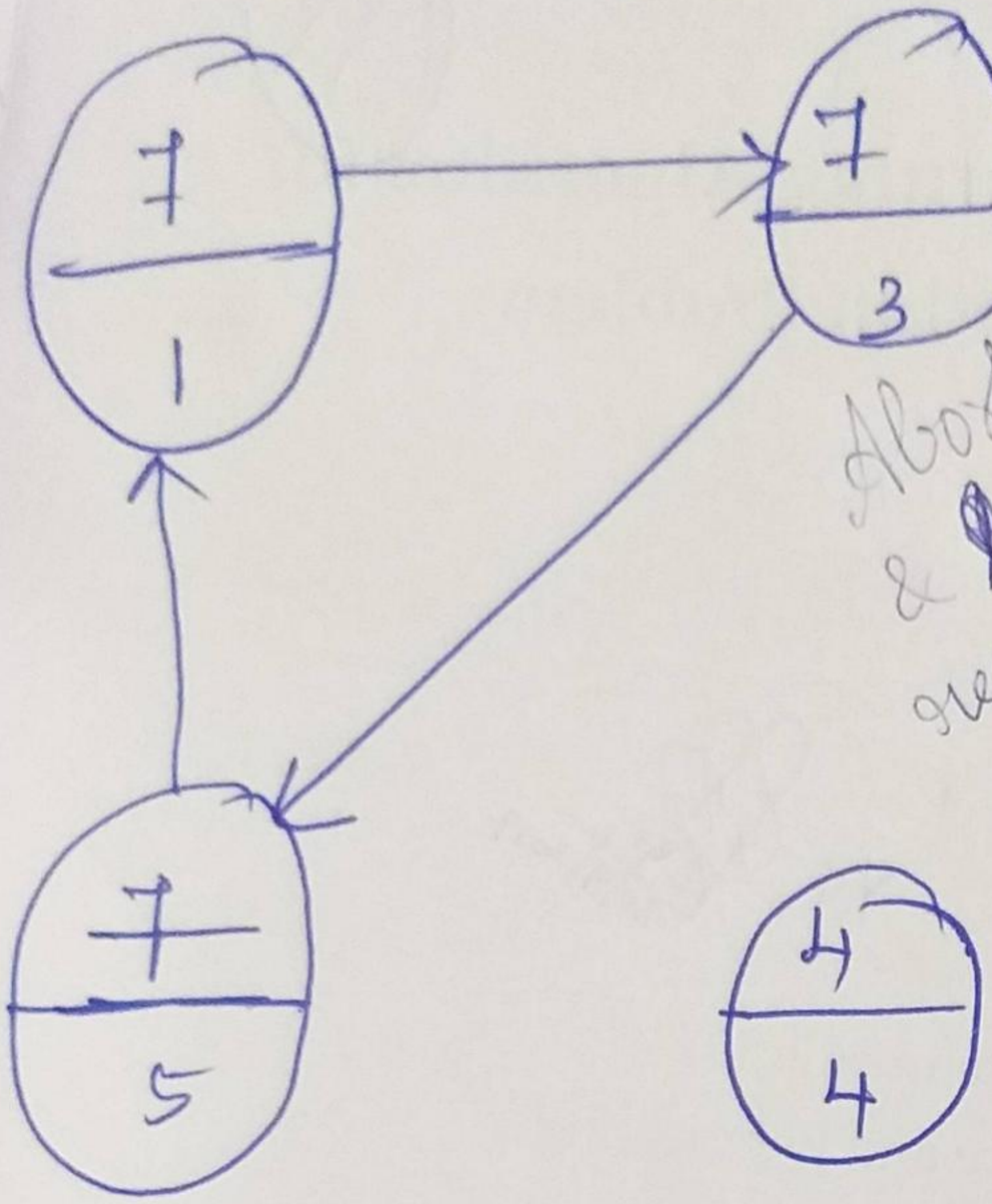


Transmit back to the blocked process to inform public label of process it is waiting for.

Process 7 initates deadlock detection algo.







Abort & Send resources to blocked process

Abort & set U = V.

Chandy-Misra-Haas Algorithm for the AND Model

- Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model is based on edge-chasing.
- The algorithm uses a special message called *probe*, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process P_i and it is being sent by the home site of process P_j to the home site of process P_k.
- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.

- A process P_j is said to be *dependent* on another process P_k if there exists a sequence of processes P_j, P_{i1}, P_{i2}, ..., P_{im}, P_k such that each process except P_k in the sequence is blocked and each process, except the P_j, holds a resource for which the previous process in the sequence is waiting.
- Process P_j is said to be *locally dependent* upon process P_k if P_j is dependent upon P_k and both the processes are on the same site.

Data Structures

- Each process P_i maintains a boolean array, $dependent_i$, where $dependent_i$(j) is true only if P_i knows that P_j is dependent on it.
- Initially, $dependent_i$(j) is false for all i and j.

Algorithm

The following algorithm determines if a blocked process is deadlocked:

- if P_i is locally dependent on itself then declare a deadlock else for all P_j and P_k such that
  1. P_i is locally dependent upon P_j, and
  2. P_j is waiting on P_k, and
  3. P_j and P_k are on different sites,send a probe (i, j, k) to the home site of P_k
- On the receipt of a probe (i, j, k), the site takes the following actions: if
  1. P_k is blocked, and
  2. $dependent_k$(i) is false, and
  3. P_k has not replied to all requests P_j,

then

begin

$dependent_k$(i) = true;

if k=i

then declare that P_i is deadlocked

else for all P_m and P_n such that

(a') P_k is locally dependent upon P_m, and

(b') P_m is waiting on P_n, and

(c') P_m and P_n are on different sites,

send a probe (i, m, n) to the home site of P_n

end.

- A probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

Performance Analysis

- One probe message (per deadlock detection initiation) is sent on every edge of the WFG which that two sites.
- Thus, the algorithm exchanges at most $m(n-1)/2$ messages to detect a deadlock that involves m processes and that spans over n sites.
- The size of messages is fixed and is very small (only 3 integer words).
- Delay in detecting a deadlock is O(n).

Chandy-Misra-Haas Algorithm for the OR Model

Chandy-Misra-Haas distributed deadlock detection algorithm for OR model is based on the approach of diffusion-computation.

- A blocked process determines if it is deadlocked by initiating a diffusion computation.
- Two types of messages are used in a diffusion computation:
- query(i, j, k) and reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process P_i and are being sent from process P_j to process P_k.

- A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set.
- If an active process receives a query or reply message, it discards it.
- When a blocked process P_k receives a query(i, j, k) message, it takes the following actions:
  1. If this is the first query message received by P_k for the deadlock detection initiated by P_i (called the *engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable num_k(i) to the number of query messages sent.
  2. If this is not the engaging query, then P_k returns a reply message to it immediately provided P_k has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

- Process P_k maintains a boolean variable $wait_k$(i) that denotes the fact that it has been continuously blocked since it received the last engaging query from process P_i.
- When a blocked process P_k receives a reply(i, j, k) message, it decrements num_k(i) only if $wait_k$(i) holds.
- A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query.
- The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.

Algorithm

The algorithm works as follows:

Initiate a diffusion computation for a blocked process P_i:

send query(i, i, j) to all processes P_j in the dependent set DS_i of P_i;

num_i(i):= $|DS_i|$; $wait_i$(i):= true;

When a blocked process P_k receives a query(i, j, k):

if this is the engaging query for process P_i

then send query(i, k, m) to all P_m in its dependent set DS_k;

num_k(i): = $|DS_k|$; $wait_k$(i):= true

else if $wait_k$(i) then send a *reply*(i, k, j) to P_j.

When a process P_k receives a reply(i, j, k):

if $wait_k$(i)

then begin

num_k(i):= num_k(i) $-$ 1;

if num_k(i)= 0

then if i=k then **declare a deadlock**

else send reply(i, k, m) to the process P_m which sent the engaging query.

- In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process.
- However, messages for outdated diffusion computations may still be in transit.
- The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

Performance Analysis

For every deadlock detection, the algorithm exchanges e query messages and e reply messages, where e=n(n-1) is the number of edges.

Distributed Deadlock Detection

Y. V. Lokeswari
AP/ CSE

Reference: **Advanced Concepts in OS**
by
Mukesh Singhal & N.G. Shivaratri

Distributed Deadlock Detection

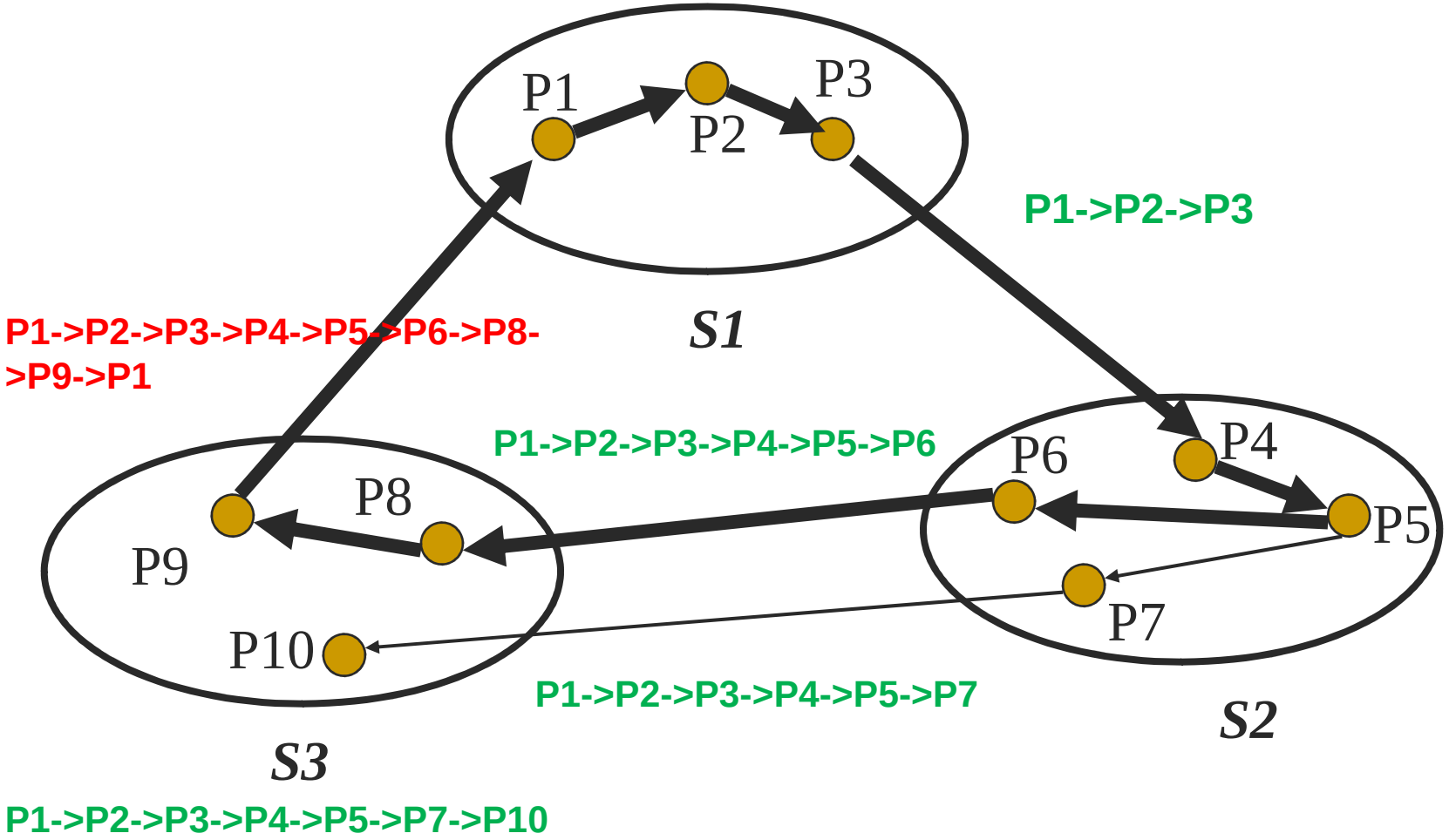
Obermarck's Algorithm

&

Chandy, Misra and Haas Algorithm

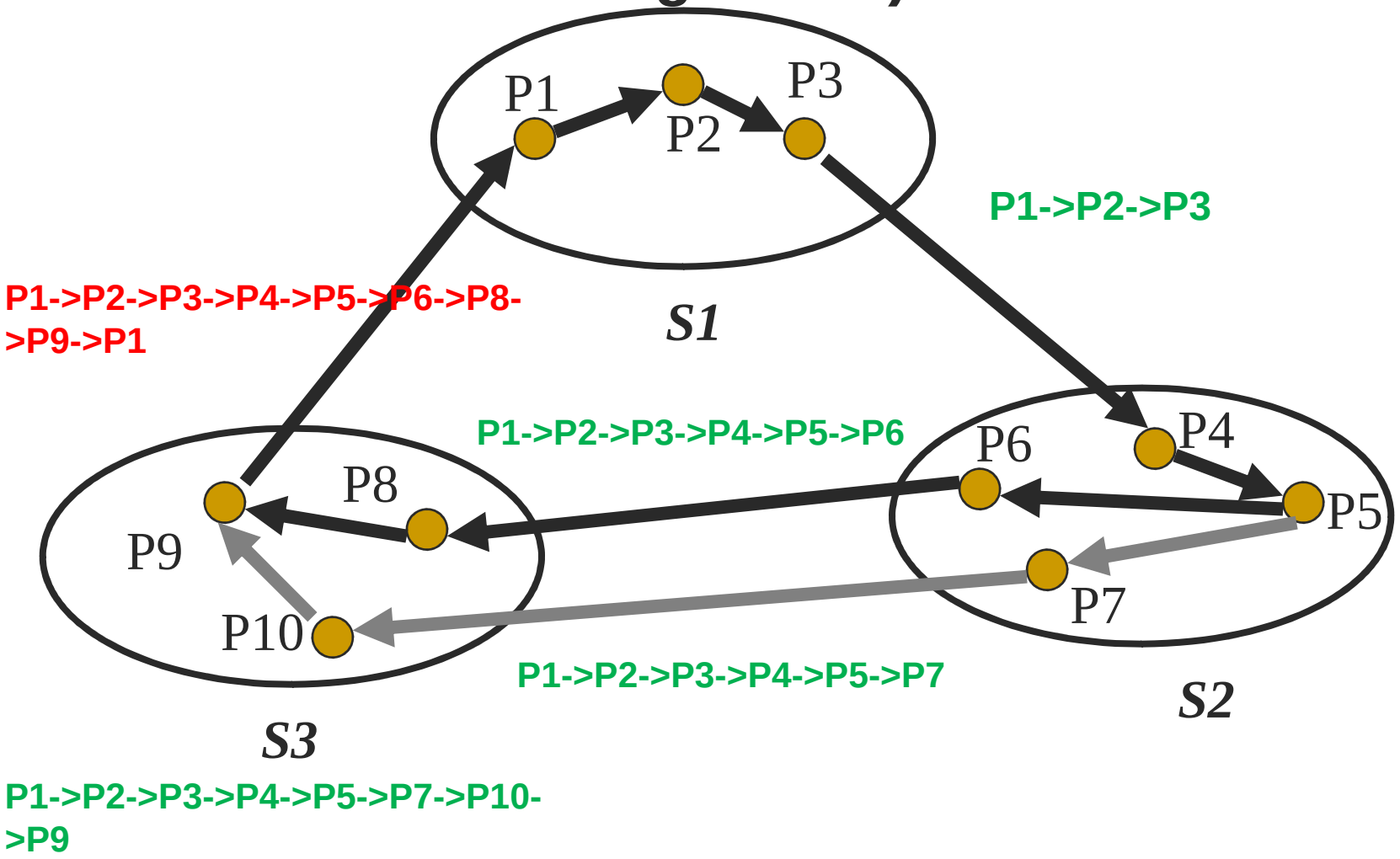
Deadlock in the **AND** model; there is a **cycle** but no knot **No Deadlock** in the **OR** model

Path-Pushing Algorithm (Obermarck's Algorithm)



Deadlock in both the **AND** model and the **OR** model; there are **cycles** and a **knot**

Path-Pushing Algorithm (Obermarck's Algorithm)



Drawbacks of Path-Pushing Algorithm

- The algorithm **detects Phantom deadlocks**
- The algorithm sends ***n(n-1)/2*** messages to detect deadlock involving ***n*** sites.
- Size of the message is ***O(n)***
- Delay in detecting deadlock is ***O(n)***
- Overhead in sending WFG information to each site via network.
- By the time deadlock is declared, deadlock would have been resolved. (**Phantom deadlocks).**

Chandy Misra Haas's Algorithm

Edge-Chasing Algorithm

- Instead of sending the WFG from each site, this method sends a Probe message which has a fixed size.
- Probe(i,j,k)

Chandy Misra Haas's Algorithm

Edge-Chasing Algorithm

Controller sending a probe

```
if Pj is locally dependent on itself
      then declare deadlock
else for all Pj, Pk  such that
      (i) Pi is locally dependent on Pj,
      (ii) Pj is waiting for 'Pk and
      (iii) Pj, Pk are on different controllers.
send probe(i, j, k). to home site of Pk
```

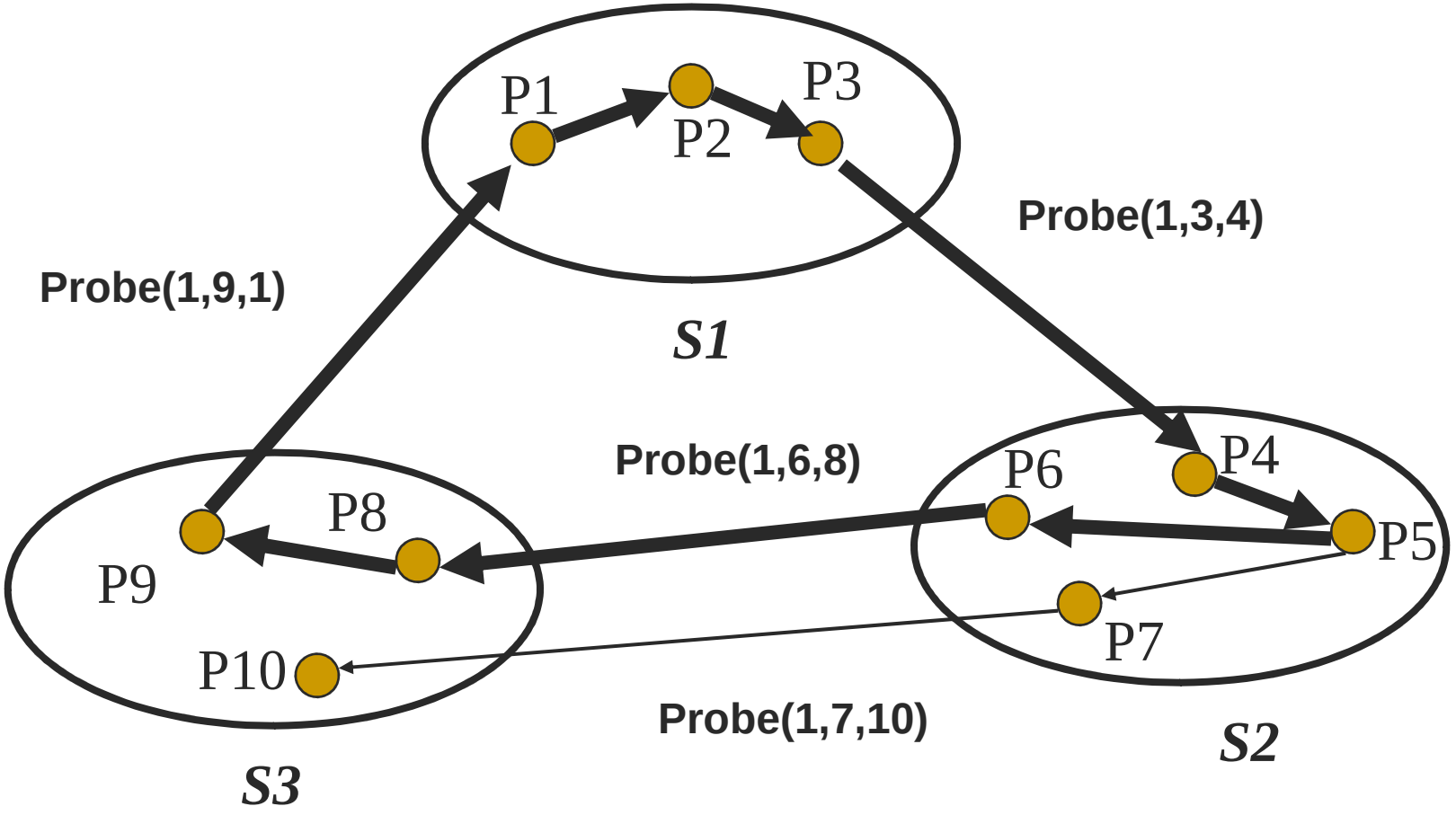
Controller receiving a probe

```
if
      (i) Pk is idle / blocked
      (ii) dependent_k(i) = false, and
      (iii) Pk has not replied to all requests of to Pj
then begin
      "dependents"_k"(i) = true;
      if k == i
      then declare that Pi  is deadlocked
      else for all Pa, Pb  such that
                (i) Pk is locally dependent on Pa,
                (ii) Pa is waiting for 'Pb and
                (iii) Pa, Pb are on different controllers.
      send probe(i, a, b). to home site of Pb
end
```

Deadlock in the **AND** model; there is a **cycle** but no knot

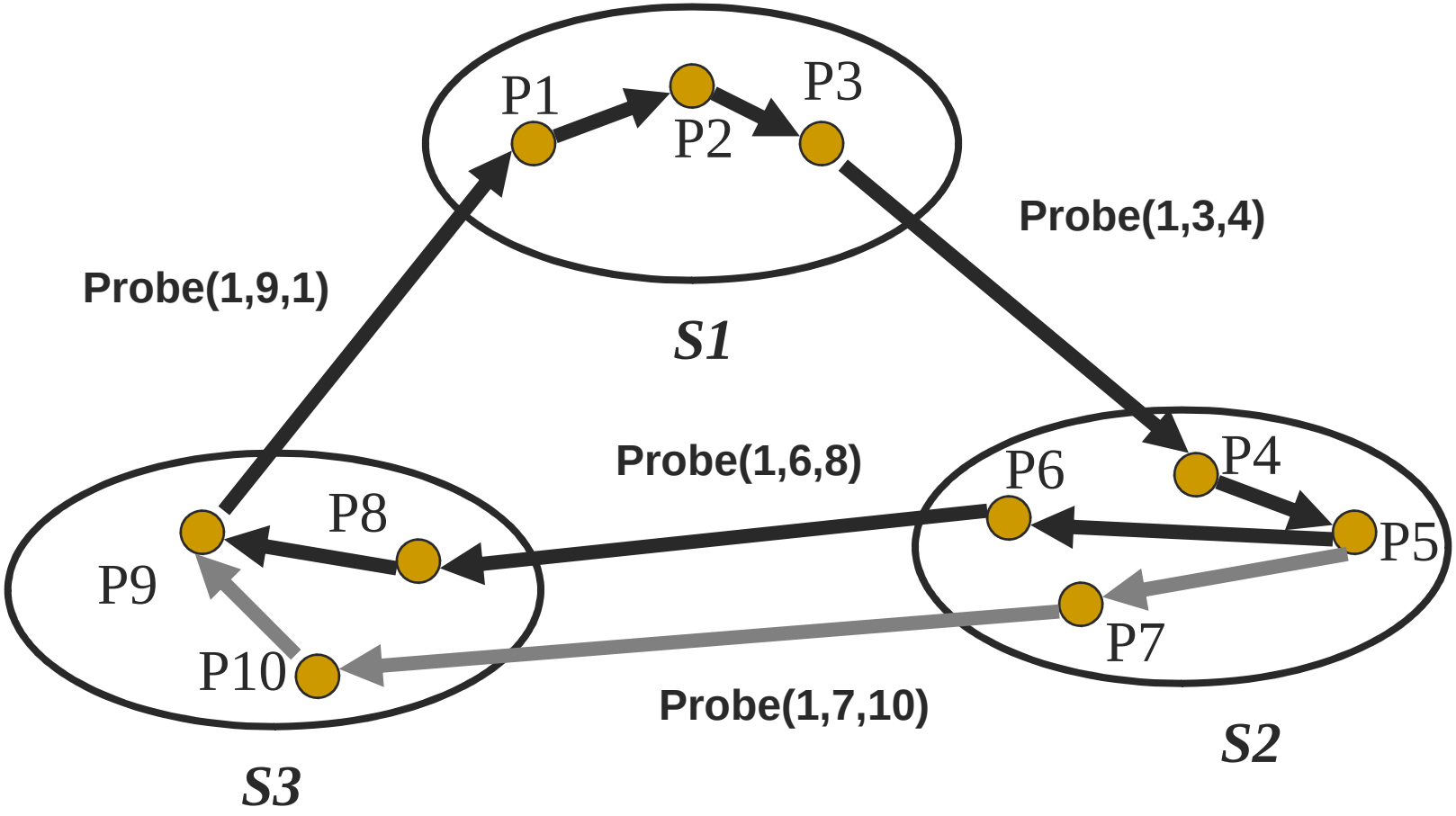
No Deadlock in the **OR** model

Edge-Chasing Algorithm



Deadlock in both the **AND** model and the **OR** model; there are **cycles** and a **knot**

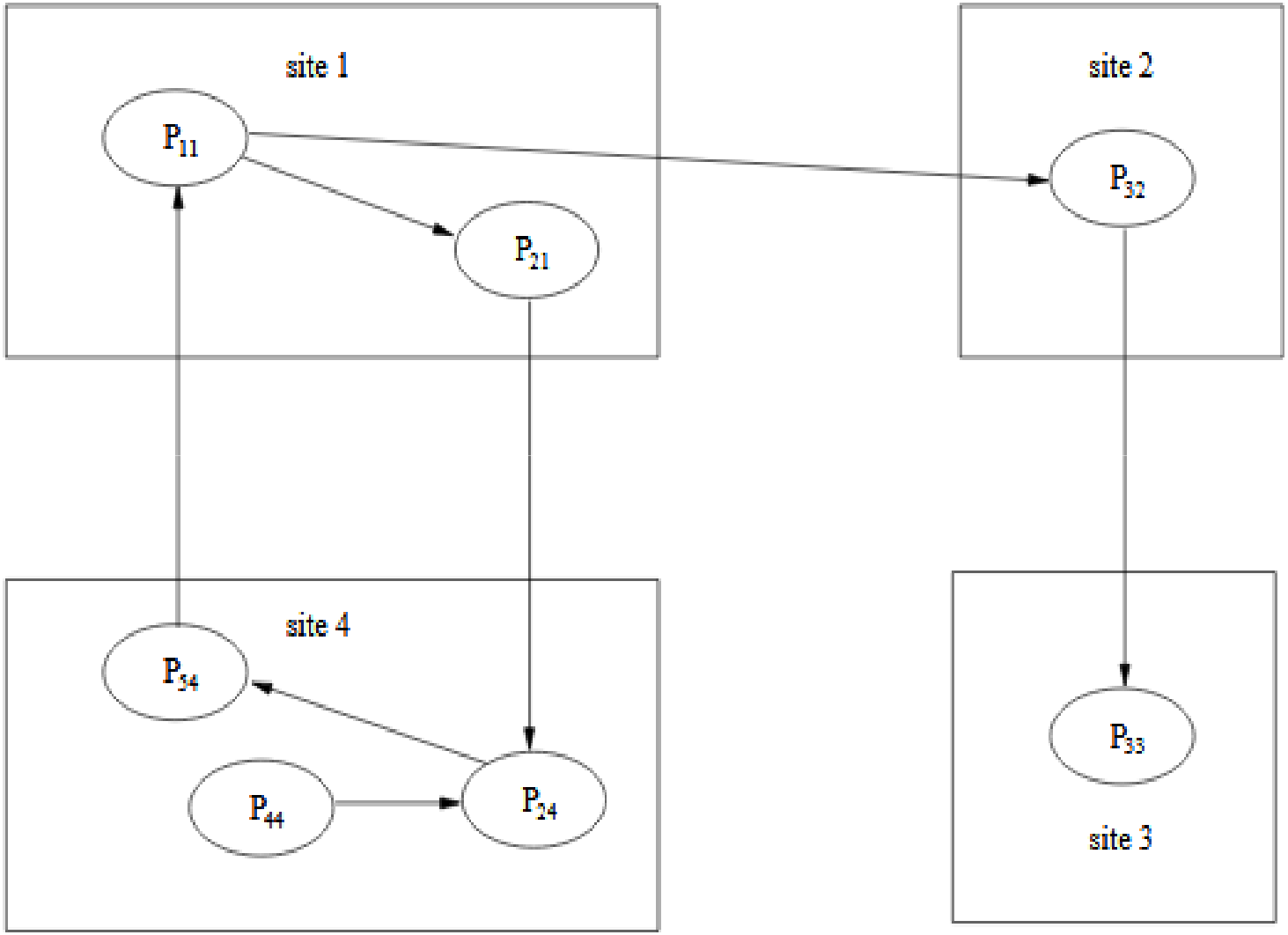
Edge-Chasing Algorithm



Edge-Chasing Algorithm

- The algorithm at most exchanges ***m(n-1)/2*** messages to detect deadlock involving ***m*** processes and spans over ***n*** sites.
- Size of the message is fixed. Only 3 integer values.
- Delay in detecting deadlock is ***O(n)***

Workout Example



Other Deadlock detection Algorithms

- Diffusion Computation Algorithm
- Global State Detection based Algorithm

Thank You