Unit3

Distributed Mutex and Deadlock

Distributed mutual exclusion algorithms: Introduction - Preliminaries - Lamport's algorithm - Ricart Agrawala algorithm - Quorum-based mutual exclusion algorithms - Maekawa's algorithm - Token-based algorithms - Suzuki-Kasami's broadcast algorithm; Deadlock detection in distributed systems: Introduction - System model - Preliminaries - Knapp's classification of distributed deadlock detection algorithms - Mitchell and Merritt's algorithm for the single resource model - Chandy-Misra-Haas algorithm for the AND model - Chandy-Misra-Haas Algorithm for the OR model

* Distributed Mutual Exclusion (D-Mutex)

→ In distributed systems, state is not consistent at all times.

Mutual Exclusion : Concurrent access of process to a shared resource is executed in a mutually exclusive manner.

→ Only one process is allowed to enter the critical section 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.

→ Three basic approaches are used for distributed mutual exclusion:

1. Token-based approach
2. Non-token based approach
3. Quorum based approach

1. 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.

2. 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

3. 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 only one ~~requests~~ executes the CS at any time.

* 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.

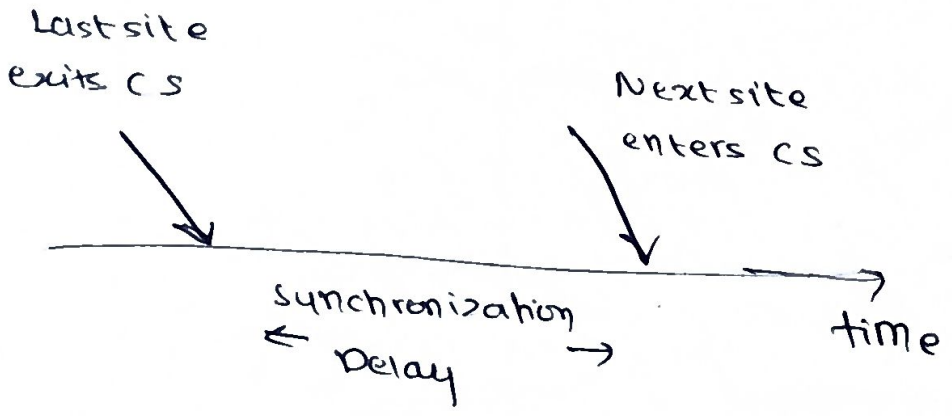
Fairness property generally means the CS execution requests are executed in the order of their arrival (time determined by a logical ~~system~~ clock) in the system.

* Performance Metrics

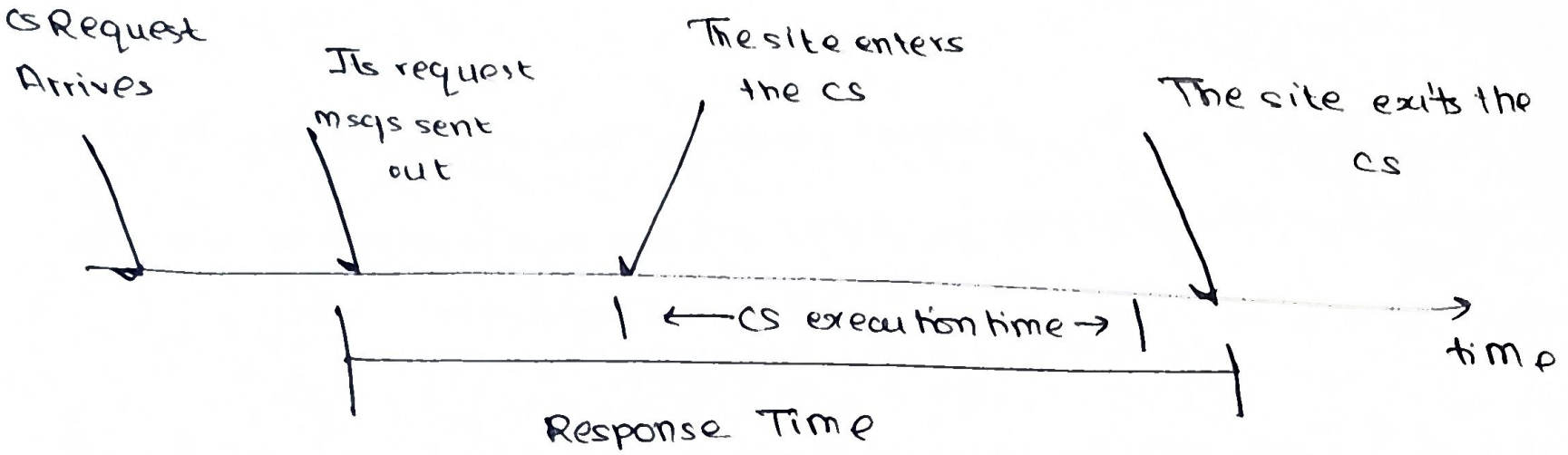
→ The performance is generally measured by 4 metrics:

1. Message complexity: The no. of messages required per CS execution by a site

2. Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS.



3. Response Time - The time interval a request waits for its CS execution to be over after its request messages have been sent out.



4. System Throughput - The rate at which the system executes requests for the CS

SD = synchronization delay

$$\text{system throughput} = \frac{1}{SD + E}$$

E = avg. critical section execution time.

* Non-Token-based Algorithms

A. Lamport's Algorithm for DMutex

1) uses distributed queues
2) broadcasts requests
3) uses Lamport's logical clock

Phases: Request, Reply, Release

Request: 1) Increment the sequence no
2) place req(s, pid)
3) broadcast to everyone

Reply : The receiver replies immediately when the receiver is not requesting the critical.

Release : When the execution of the critical section is over,
→ release (s, pid) is broadcast
→ causally, receiver deletes req(s, pid) on receiving this.

Example 1 P_1 wants to enter the CS

	P_1	P_2	P_3
0	s=0	s=0	s=0
1	req(1,1)	rec req(1,1)	rec req(1,1)
2		rep req(1,1)	rep req(1,1)
3.	rec rep(1,1) : P_2		
4.	rec rep(1,1) : P_3		
	enter CS exec CS		
	rel req(1,1)		
		rec rel(1,1) del req(1,1)	rec rel(1,1) del req(1,1)

Example 2 $S_1 \rightarrow S_2$

(can draw queue too)

P_1	P_2	P_3
~~rec~~ s = 0	s = 0	s = 0
req (1,1)	rec req (1,1)	rec req (1,1)
	req (2,2)	
rec req (2,2)		rep (1,1)
		rec req (2,2)
	P1 is first in the queue and has a smaller PID	~~rep (1,1)~~
		rep (2,2)
	rep (1,1)	
rec rep (1,1): P2	rec rep (2,2): P3	
rec rep (1,1): P3		
exec cs		
exit cs		
del req (1,1)		
	rec rel (1,1)	rec rel (1,1)
del req (1,1)	del req (1,1)	del req (1,1)
	rec rep (2,2) P1	
	exec cs	
	exit cs	
	rel req (2,2)	
rel rec (2,2)		rel rec (2,2)
del req (2,2)	del req (2,2)	del req (2,2)

Example 3 S1 || S2 (only 2 processes P_1 & P_2)

P_1	P_2	P_3
s = 0	s = 0	s = 0
req (1,1)	req (1,2)	
rec req (1,2)	rec req (1,1)	
DEADLOCK	SINCE BOTH MAKE SIMULTANEOUS REQUES	
	P_1 given prior because of lower PID	
	rep (1,1)	
rec rep (1,1) : P_2		
exec CS		
exit CS		
rel req (1,1)	rec rel (1,1)	
del req (1,1)	del req (1,1)	
rep (1,2)		
	rec rep (1,2) : P_1	
	exec CS	
	rel req (1,2)	
rec rel (1,2)	del req (1,2)	
del req (1,2)		

Example 4 $S_1 \rightarrow S_1 \rightarrow S_2$ (2 processes)

(delayed)

P_1	P_2
req (1,1)	
req (2,1)	
	rec req (2,1)
	rep (2,1)
rec rep (2,1) P2	
enter CS, exec CS	
	rec req (1,1) → just ignore this, it is late
rel req (2,1)	
	req (3,2)
rec req (3,2)	
	rel req (2,1)
del req (2,1)	
	del req (2,1)
rep (3,2)	
	rec rep (3,2) : P1
	enter CS, exec CS
	del rel req (3,2)
rec rel (3,2)	
del req (3,2)	
	del req (3,2)

DO NOT ENQUEUE OUTDATED REQUESTS

* Characteristics of Lamport's Algorithm

Message Complexity : $3(N-1)$

Synchronization Delay : T (round-trip delay)

System Throughput : $(T+E)^{-1}$

Fairness - All processes need to validate / approve before a process enters the critical section

usecase

If a process is under a DDOS attack, it cannot make another request, if it already has a pending request.

Liveness - Starvation occurs when 2 processes make concurrent requests. This is resolved by prioritising the process with lower PID.

Safety / Mutual Exclusion - Assume that 2 processes are in the critical section.

Then their sequence ids would have to be the same, but their process ids cannot be the same.

The processes would be sorted on the basis of PID.

Only one process can be on top

∴ Both cannot enter the CS at the same time.

B. Ricart - Agrawala Algorithm

example:

$S_1 \rightarrow (S_2 \| S_3) \rightarrow (S_4 \| S_5)$

→ an optimization of Lamport's algorithm

→ 2 rounds are used

Request → N-1 msgs

Reply → N-1 msgs

No release message

Example 1 $S_1 \| S_2$

P_1	P_2	P_3
$S_1 = 0$	$S_1 = 0$	$S = 0$
req (1,1) (s, p)	req (1,2)	rec req (1,1)
rec req (1,2)	rec req (1,1)	rec req (1,2)
	compare PIDs PID 1 < PID	rep (1,2) rep (1,1)
	rep(1,1)	
rec rep(1,1) : P2 rec rep (1,1) : P3		
exec CS		
del req (1,1)		
(automatically deleted in queue for all P's)		
rep(1,2)	rec rep(1,2) : P3 rec rep(1,2) : P1 enter CS, exec	
	del req (1,2)	

Message Complexity : 2(N-1)

Fairness - through ordering (seq id, pid)

- as well as through lamport's logical clock

Handling DOS - a process cannot raise more request if there are pending requests for that process.

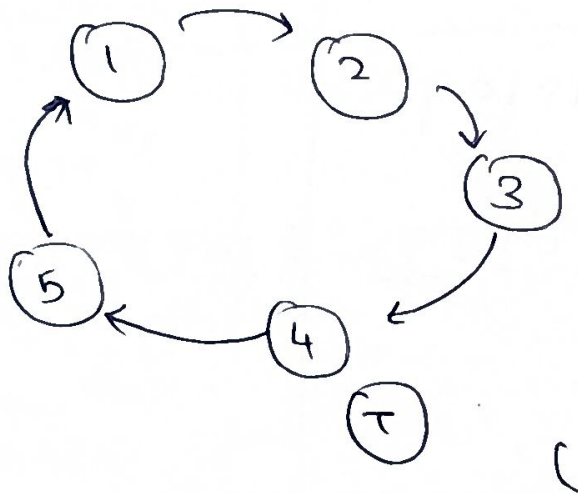
Mutual Exclusion - proof by contradiction - same as Lamport's algo

Liveness - same as lamport's algo.

* Token-based D-Mutex

→ Nodes are arranged in a ring topology

→ A token is present initially at one node - each node has a default neighbor that the token can be passed to



→ Here, the token is at 4

→ 4 can enter cs if it wants to

→ If it has no need of the token, pass it through the default edge

(can only pass to one node, otherwise mutual exclusion cannot be achieved)

Problem - In the worst case scenario, where the node that wants the token is all the way around the ring, Complexity would be O(n)

* Suzuki - Kasami Algorithm

→ There are 2 kinds of arrays used

Array R : 

Array T : 

→ Each process should make requests in its own R array

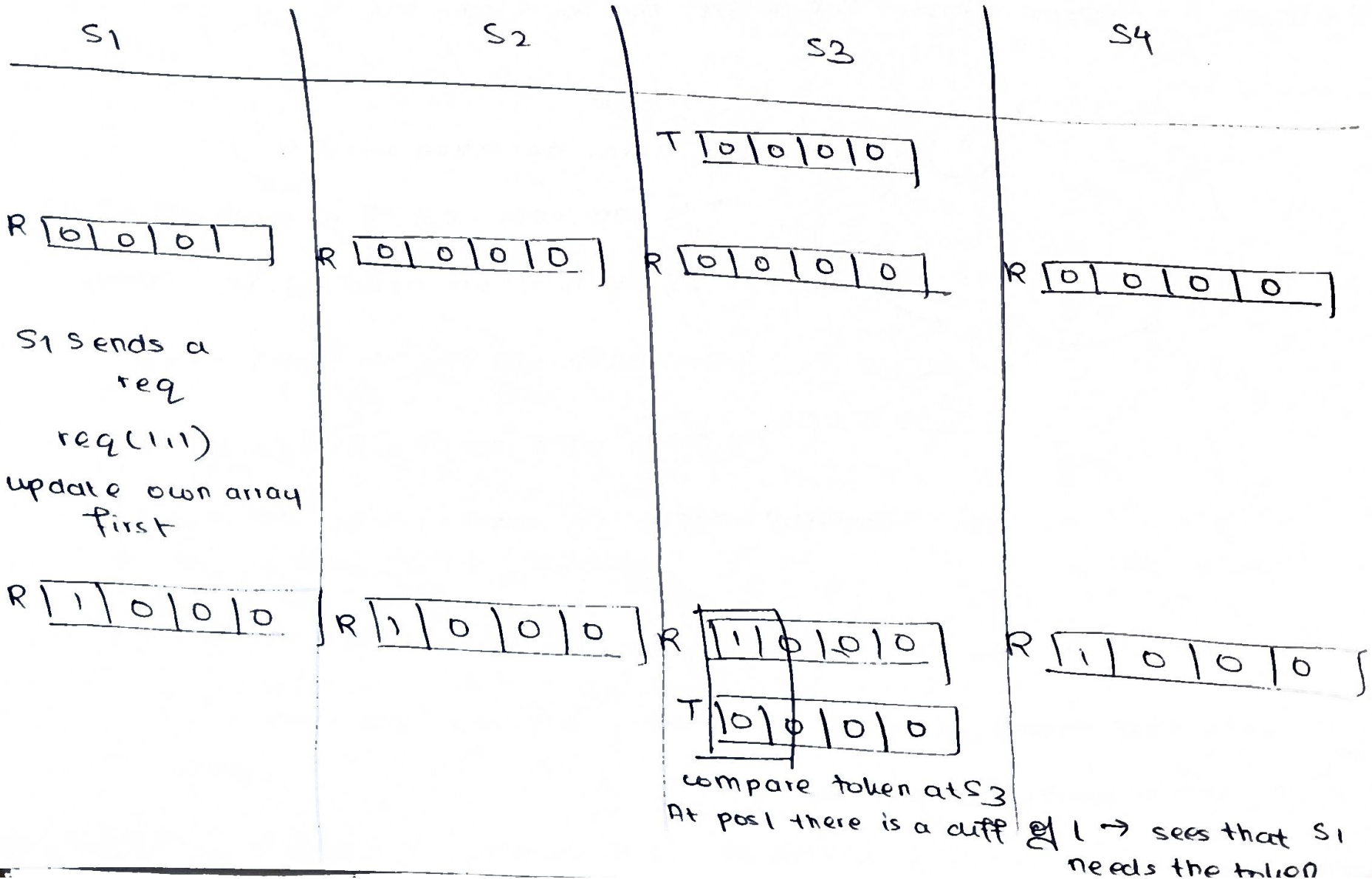
Each process has its own R array

→ Array T is present wherever the token is .

→ When the token is being passed around, it means array T is being moved around.

→ Both the arrays are populated with zeroes by default.

Example 1 $S_1 \rightarrow S_2 \rightarrow (S_3 \| S_4)$ Token is at S3



S_1	S_2	S_3	S_4
		passes token to S_1	
T [0|0|0|0]			
execute CS exit CS update token			
T [1|0|0|0]			
	S_2 makes a request req(2,2)		
	R [1|1|0|0]		
rec req(2,2)	~~S2~~	rec req(2,2)	rec req(2,2)
R [1|1|0|0] T [1|0|0|0]		R [1|1|0|0]	R [1|1|0|0]
check from last checked pos after			
diff = 1 ⇒ S_2 needs token			
	T [1|0|0|0]		
	enter CS, exec exit CS update token		
	T [1|1|0|0]		

S1	S2	S3	S4
(iii) S3 & S4 make concurrent requests		req (3,3)	req (3,4)
rec req (3,3) rec req (3,4)	rec req (3,3) rec req (3,4)	rec req (3,4)	rec req (3,3)
R 1 1 1 1	R 1 1 1 1 T 1 1 0 0 Compare at next pos from last comparison (break tie between S3 & S4)	R 1 1 1 1	R 1 1 1 1
		R 1 1 0 0	R 1 1 0 1
		T 1 1 0 0 enter cs exec cs exit cs update token	
		T 1 1 1 0	
		R 1 1 1 1 T 1 1 1 0	
			T 1 1 1 0 enter cs exec cs, exit update token
			T 1 1 1 1

Fairness - token difference is calculated from the last read position

- in case of DDOS, cannot make another request in case of pending requests.

Handling Outdated Requests (not able to enter CS within a certain time period)

~~increment the corresponding T to T+1~~

- Increment R value by 1, now there would be a difference of 2

- can no longer ascertain that the process is deserving.

- After passing/ skipping that operation, increment the token by 1, so that the difference once again becomes 1

Liveness - no starvation possible

scan in order of last made request position

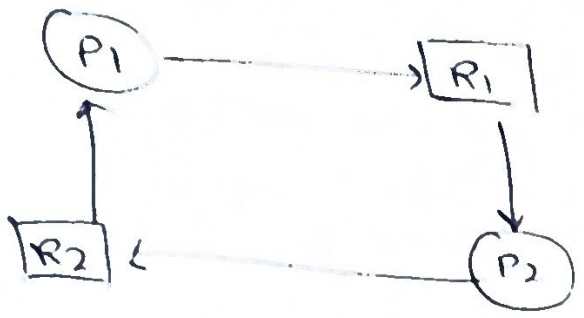
No 2 processes can acquire the resources at the same time.

Similarity to Lamport : (i) request, broadcast & reply to deserving candidates

(ii) implemented w/ distributed queues - L

implemented as arrays w/ seq id - Suzuki-Kasami

* Distributed Deadlocks



a deadlock

→ In distributed systems, there is no common clock, processor observer.

Phantom Observer : By the time, info about a deadlock is collected and declared, the deadlock no longer exists.

The opposite can also happen where deadlocks can arrive even if otherwise declared ⇒ ~~called~~ a transaction

→ Requests/ current possession of resources are denoted by transactions

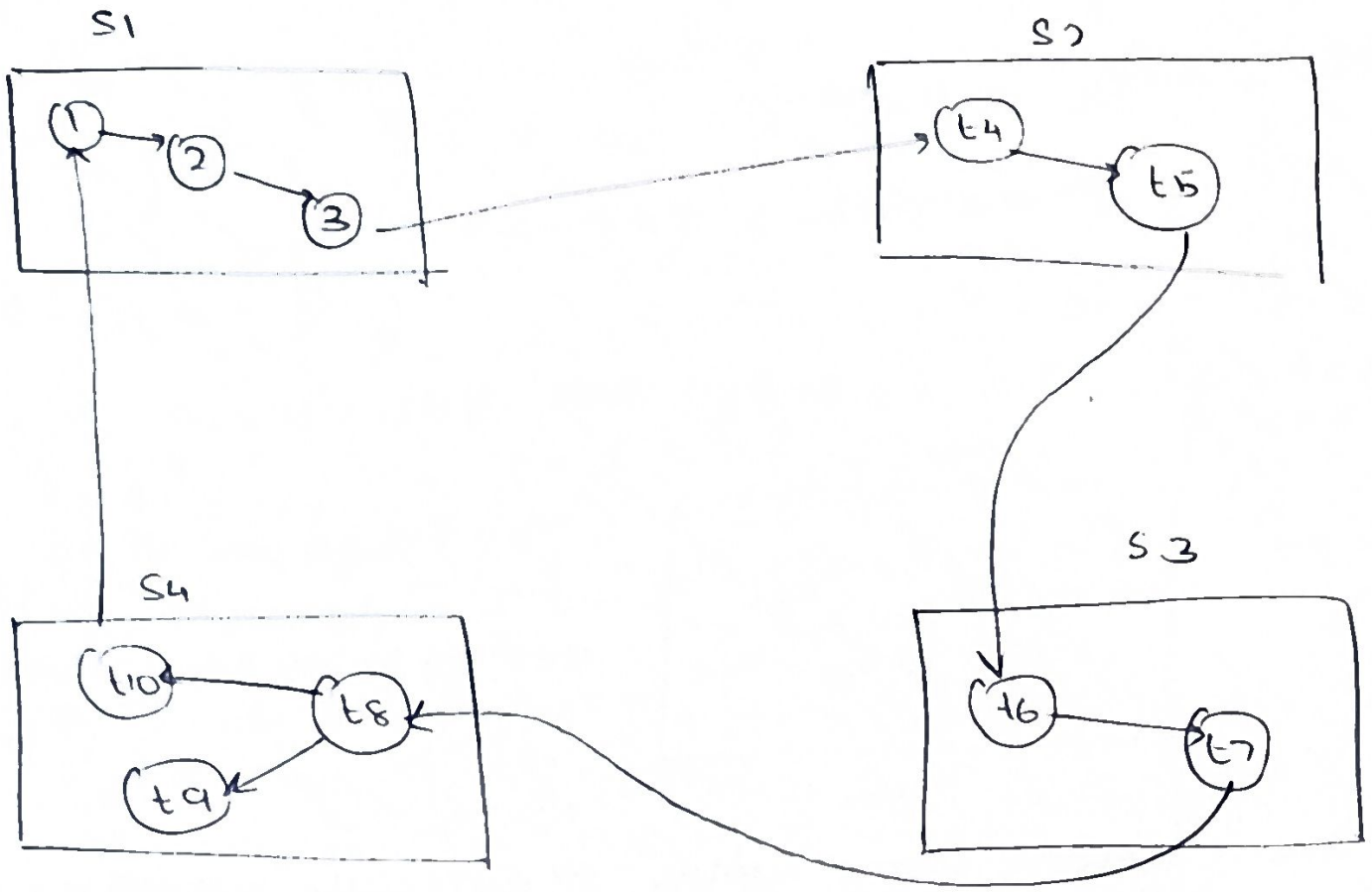
$P_1 \rightarrow R_1 \quad = t_1$

$R_1 \rightarrow P_2 \quad = t_2$

AND Model - A process needs more than one resource, and can only proceed if it procures all

* Path-Pushing Algorithm

Detect a deadlock for the following set-up using the path-pushing algorithm.



Step1 : S1 constructs a string

ext → t1 → t2 → t3 → ext

Step2 : S2 constructs its own path

ext → t4 → t5 → ext

Step3 : concatenate both strings

ext → t1 → t2 → t3 → ext → t4 → t5 → ext

Step4 : S3 creates its string & concatenates

ext → t6 → t7 → ext

concat: ext → t1 → t2 → t3 → ext → t4 → t5 → ext → t6 → t7 → ext

Step5 : S4 constructs its string and concatenates

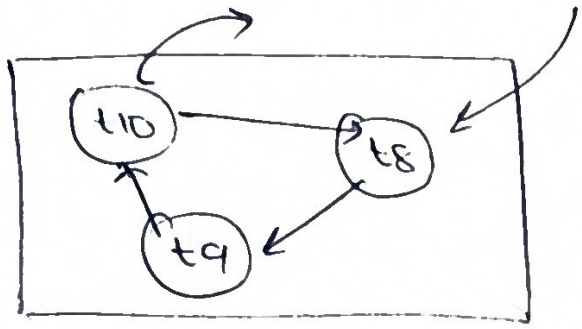
ext → t8 → t10

ext → t1 → t2 → t3 → ext → t4 → t5 → ext → t6 → t7 → ext → t8 → t10 → ext

→ When S_1 checks the concatenated string, t_i is already present. This creates a cycle

⇒ declares a deadlock

Key Considerations : In a situation like this



The string would be

ext → t8 → t9 → t10 → t8 → ext

must detect local cycles and declare a deadlock.

Drawbacks of path-pushing algorithm

1. In the overhead / processing time taken for concatenation & processing the string - the states may have become free of deadlock

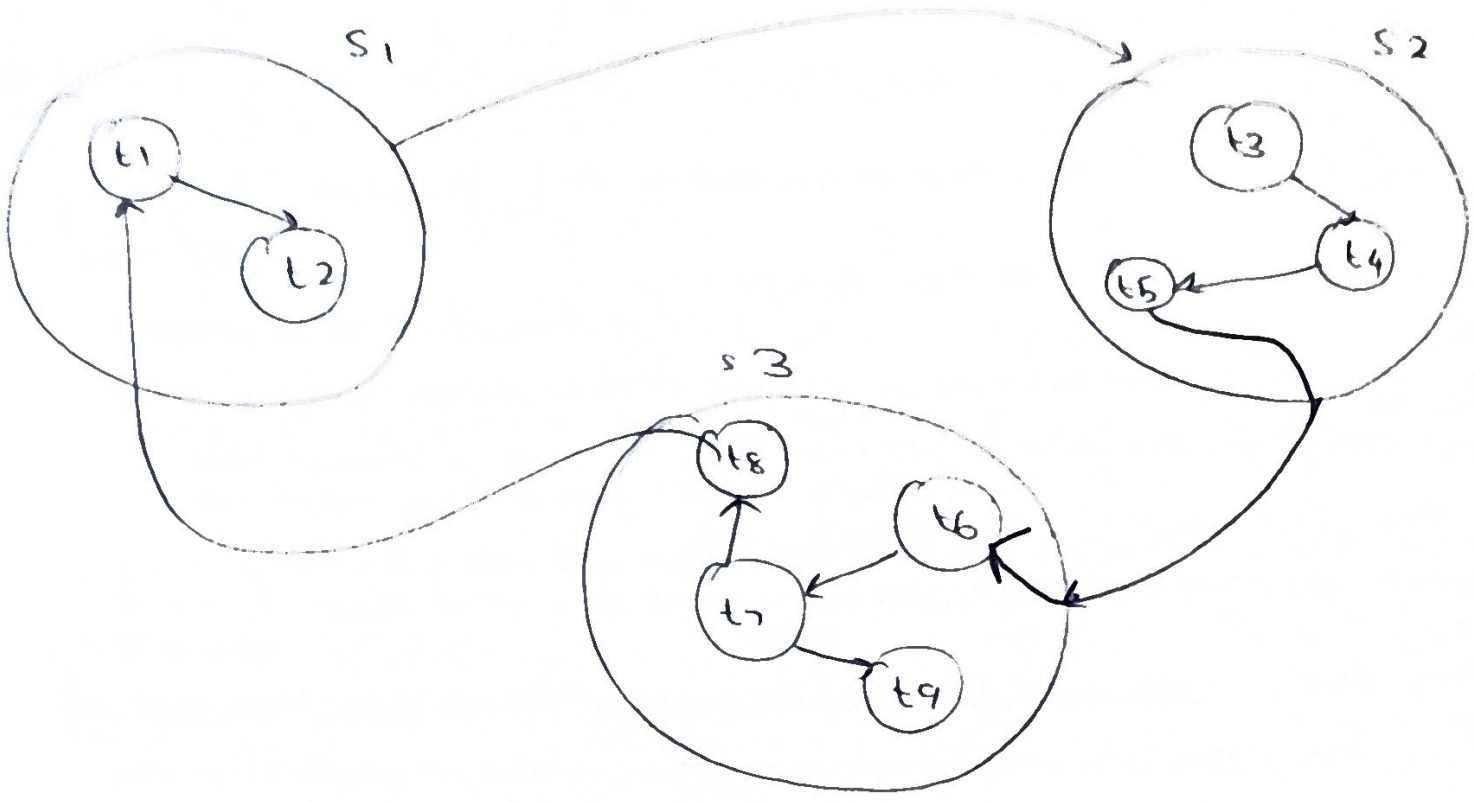
i.e the states are obselete (Phantom deadlocks)

soln - come up with a solution that transfers only a little bit of information, rather than the whole string

- Chandy - Misra - Haas Algorithm

* Edge - Chasing Algorithm - Chandy - Misra - Haas

Working and Algorithm



Algorithm : S_1 sends a probe to S_2

probe : (i, j, k)

i = initiator

j = transaction waits on external node

k = external node part of transaction

i.e it would send (1, 2, 3)

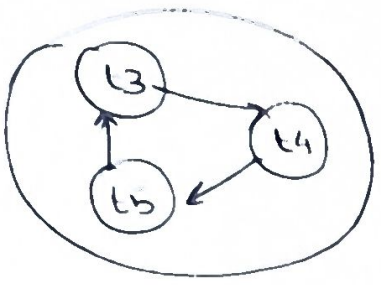
when S_2 receives it, it changes the j and k bits as (1, 5, 6)

when S_3 receives it, it changes the j & k bits to (1, 8, 1)

both belong to S_1 → hence a deadlock is declared.

Note : even if the tuple was (1, 8, 2) it would be a deadlock, since both belong to S_1

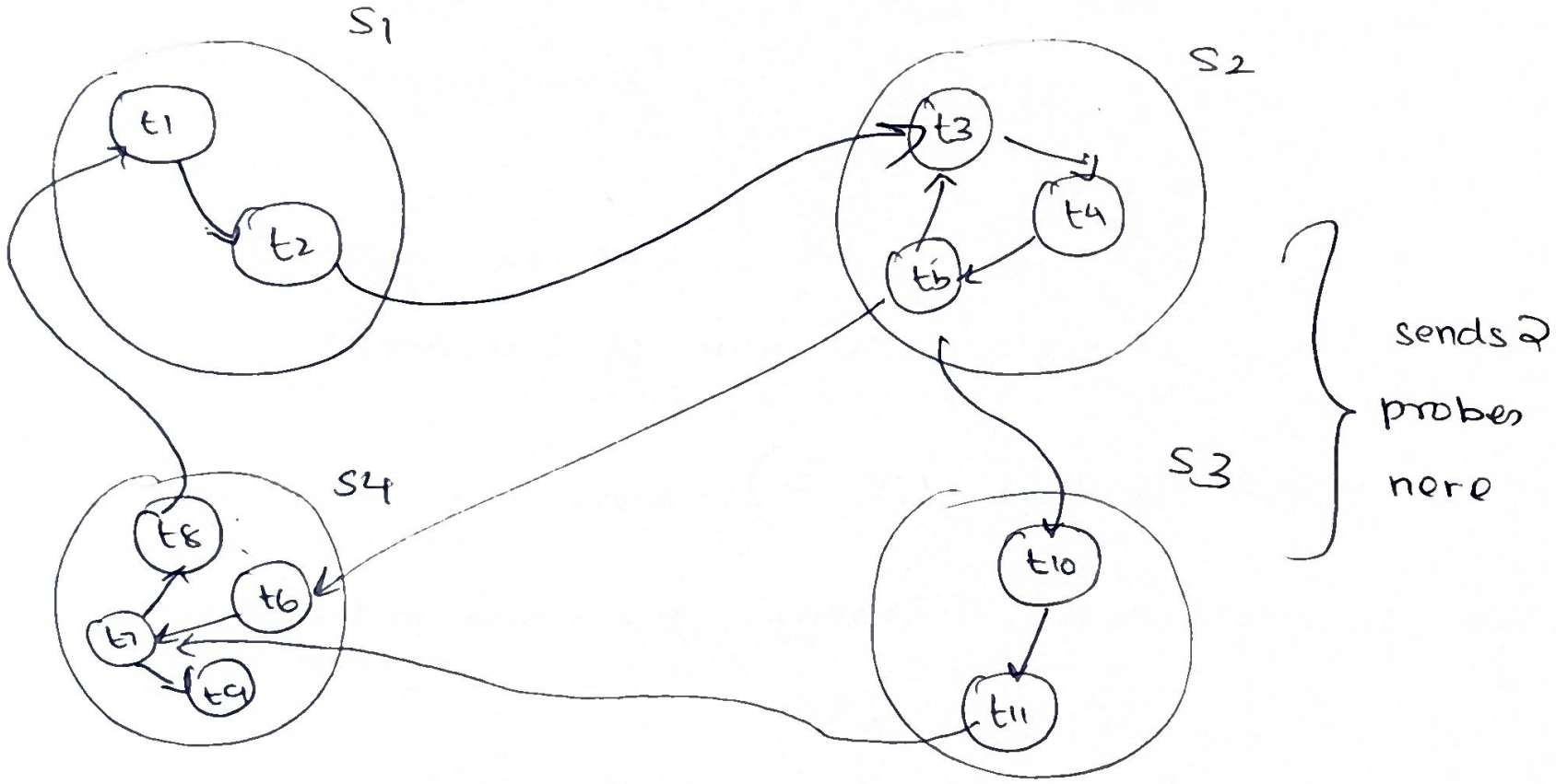
* How to detect local deadlocks?



On receiving probe, check for local deadlock using simple operating system-based time comparisons.

All systems in S_2 share a clock and memory & are aware of the ordering of processes.

* Counting the number of deadlocks → count the no. of probe messages received.



S1 would understand that there are 2 deadlocks since there is a cycle both through t6 & t11

* Proof of Detecting Deadlocks Correctly?

To prove: The iniator always detects the presence of deadlocks & the state will not change in between

→ would need to prove that the time taken for the change of edges (change of state) is greater than the time for the probe to receive the iniator - cannot prove directly - use probability based methods

UCS2701 Distributed Systems

Unit 3

Requirements for Distributed Mutex

1 Safety Property: At any instant, only one process can execute the critical section.

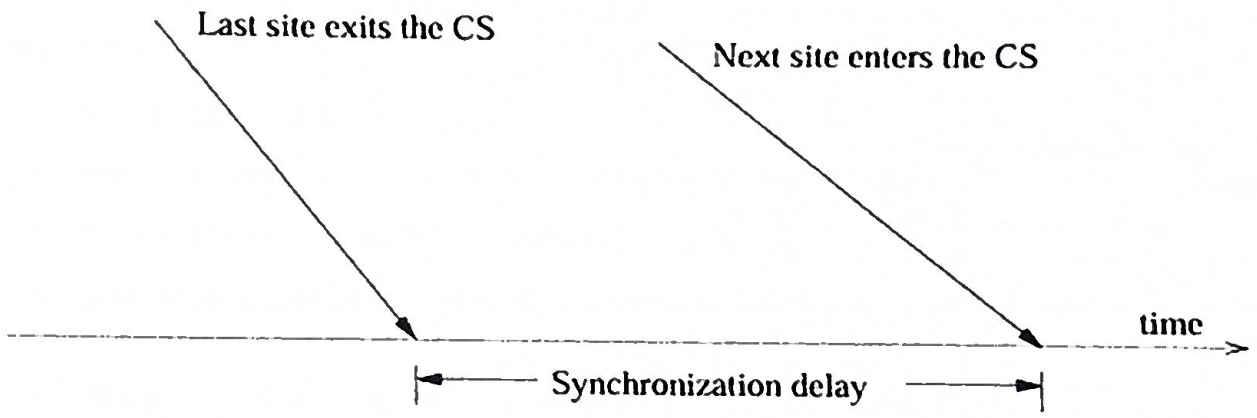
2 Liveness Property: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.

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

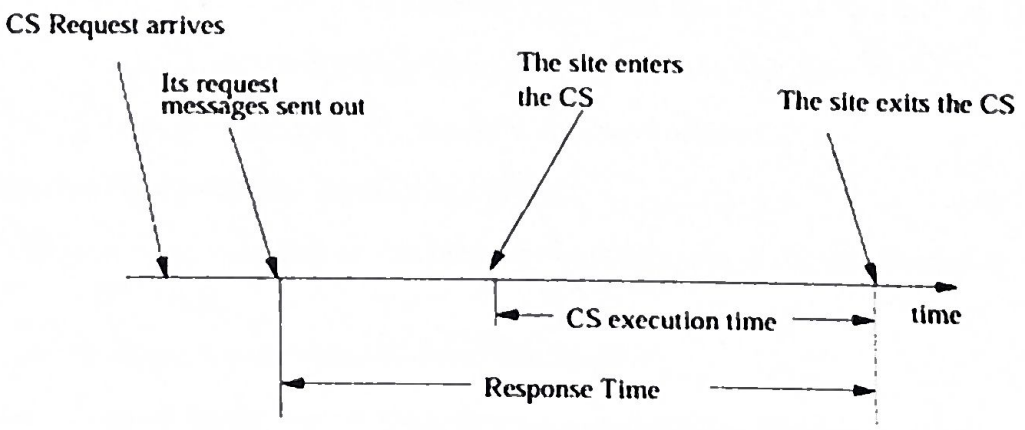
Performance Metrics by which Distributed Mutex Algorithms are Analyzed

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



Response time: The time interval a request waits for its CS execution to be over after its request messages have been sent out



System throughput: The rate at which the system executes requests for the CS system. throughput=1/(SD+E) where SD is the synchronization delay and E is the average critical section execution time.

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.

Knapp's Classification of Distributed Deadlock Detection Algorithms

Distributed deadlock detection algorithms can be divided into four classes:

path-pushing

edge-chasing

diffusion computation

global state detection

Path-Pushing Algorithms

- Detect deadlocks by building a global **Wait-For Graph (WFG).**
- Each site sends its local WFG to neighboring sites.
- Sites combine and update WFGs until a complete global view is formed to detect or confirm no deadlock.
- The term "path-pushing" comes from sending WFG paths across sites.

Edge-Chasing Algorithms

- Detect cycles (deadlocks) by sending special **probe messages** along graph edges.
- Probes differ from regular request/reply messages and are sent only by blocked processes.
- A cycle is detected if a site receives a probe it previously sent.
- Probes are short and efficient for detecting deadlocks.

Diffusing Computations Based Algorithms

- Deadlock detection is distributed by sending query messages through the **Wait-For Graph (WFG).**
- Uses **echo algorithms** to detect deadlocks.
- Process sends query messages along all outgoing edges.
- Queries propagate until all processes are queried.

- A process sends a reply after receiving replies for all its sent queries.
- Deadlock is detected when the initiator receives replies for all its sent queries.

Global State Detection Based Algorithms

- Deadlocks are detected by taking a **snapshot** of the distributed system.
- Key points:
    1. A consistent snapshot can be captured without pausing the system.
    2. If a deadlock exists before the snapshot, it will still exist in the snapshot.
- Deadlock detection involves analyzing the snapshot for cycles or waiting dependencies.

* Quorum-based Algorithm - Maekawa's Algorithm

→ Quorum-based mutual exclusion algorithms have the following key features:

1. A site does not request permission from all other sites, but only from a subset of sites called a quorum.

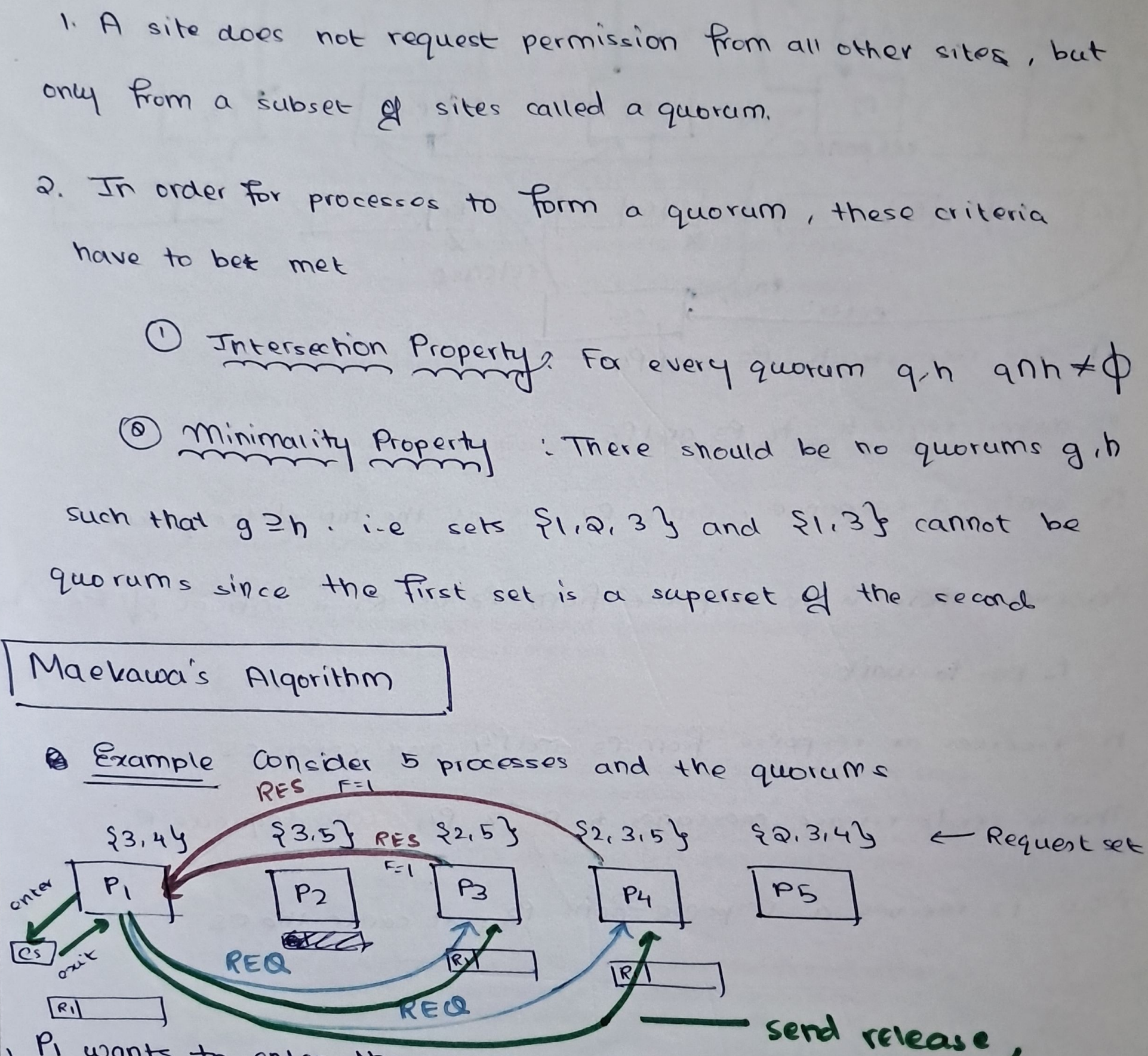
2. In order for processes to form a quorum, these criteria have to bet met

   ① Intersection Property: For every quorum q, h $q \cap h \neq \phi$

   ② Minimality Property: There should be no quorums g, h such that $g \supseteq h$. i.e sets $\{1,2,3\}$ and $\{1,3\}$ cannot be quorums since the first set is a superset of the second

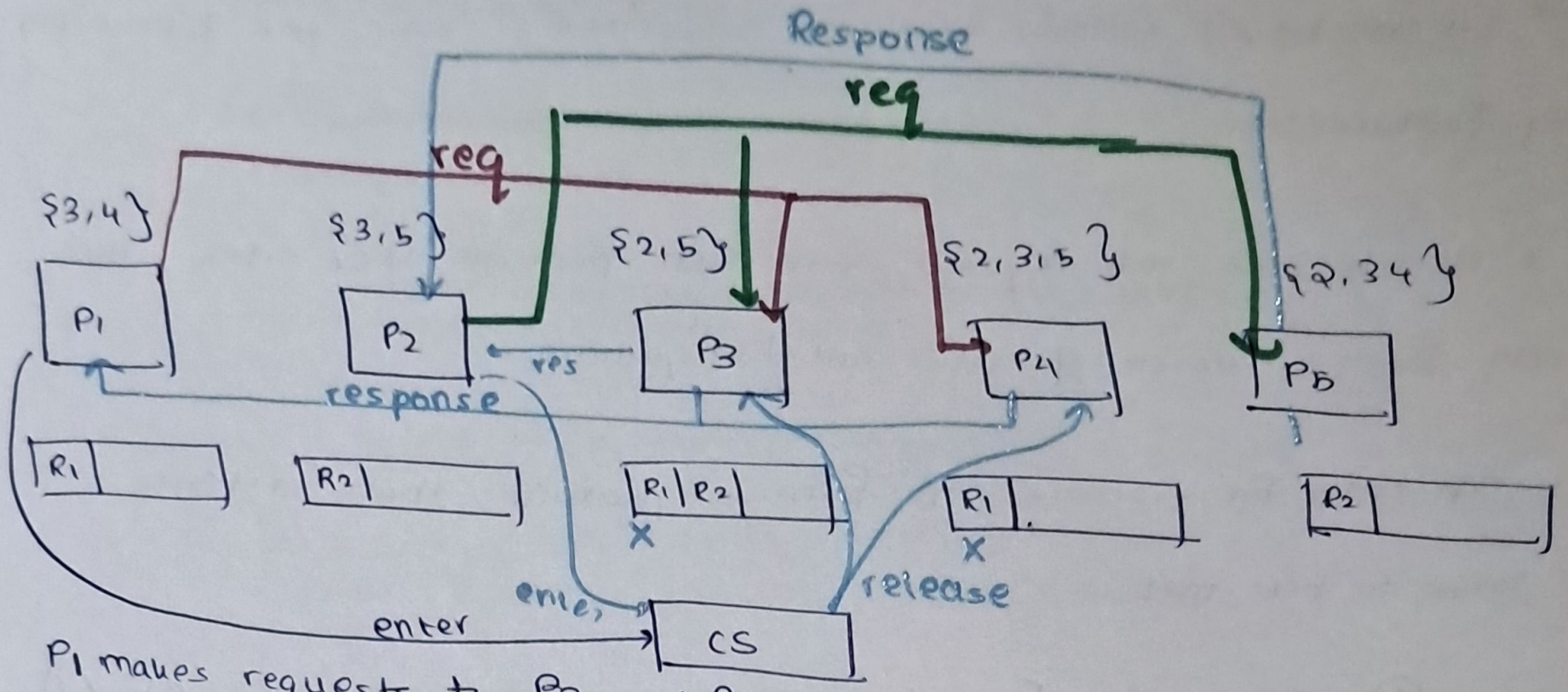
Maekawa's Algorithm

- Example Consider 5 processes and the quorums



1. P_1 wants to enter the critical section
2. sends a request to P_3 & P_4 P_1
3. P_3 and P_4 enqueue the request onto their queue
4. P_3 and P_4 don't have any other process waiting so the send a reply
5. P_1 enters CS, sends release msgs

Now consider a case where P_1 and P_2 want to enter CS concurrently



P_1 makes requests to P_3 and P_4

P_2 makes requests to P_3 and P_5

P_5 sends a response, since P_2 is at the top of the queue

However, there is no response from P_3 as R_1 is infront of P_2

$\therefore$ P_2 has to wait

P_1 receives a response from P_3 and P_4 and enters CS

Then it sends release messages to P_3 & P_4, who dequeue R_1

Now P_2 receives a response from P_3, it can te the CS

Quorum-based Distributed Mutex Algorithm- Maekawa's Algorithm

A site Si executes the following steps to enter, execute, and release the critical section (CS):

1. Requesting the Critical Section

1. Step (a):
    - Site Si requests access to the critical section by sending REQUEST(i) messages to all sites in its request set Ri.
2. Step (b):
    - When a site Sj receives a REQUEST(i) message:
        - It sends a REPLY(j) message to Si if it has not already sent a REPLY message to another site since the last RELEASE message.
        - Otherwise, it queues the REQUEST(i) for later processing.

2. Executing the Critical Section

3. Step (c):
    - Site Si executes the critical section only after receiving a REPLY message from every site in Ri.

3. Releasing the Critical Section

4. Step (d):
    - After completing the execution of the critical section, site Si sends a RELEASE(i) message to every site in Ri.
5. Step (e):
    - When a site Sj receives a RELEASE(i) message from Si:
        - It sends a REPLY message to the next site in its queue (if there is one) and deletes that entry from the queue.
        - If the queue is empty, Sj updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

Mitchell-Meritt Algorithm for the Single Resource Model Mitchell and Merritt's algorithm is an **edge-chasing algorithm** designed for deadlock detection in distributed systems with a **single resource**. It works by sending **probes** along the edges of the wait-for graph (WFG) to identify cycles, which signify deadlock.

1. **Wait-For Graph (WFG)**:
    - A graph where nodes represent processes.
    - Directed edges represent "waiting" relationships (e.g., if process A is waiting for process B, there is an edge from A to B).
2. **Labels**:
    - Each node (process) maintains two labels:
        - **Private Label**: Unique to the node, changes with each new request.
        - **Public Label**: Visible to other processes and used in probes.
    - Initially, the private and public labels are the same for all processes.
3. **Blocking and Transmitting**:
    - **Blocking**: Creates an edge in the WFG.
    - **Transmit**: Sends a **probe** in the opposite direction of the WFG edges to propagate labels.

Steps of the Algorithm

1. Initiating Deadlock Detection

- When a process suspects deadlock (e.g., it has been waiting too long), it initiates the detection process by sending a **probe** with its public label

2. Processing Probes

- A process that receives a probe acts as follows:
    - If the probe's label is **less than** its own public label, it ignores the probe.
    - If the probe's label is **greater than or equal to** its public label, it propagates the probe to the processes it is waiting for.

3. Detecting Deadlock

- If a probe initiated by a process **returns to the same process** with its private label, it indicates a **deadlock cycle**.
- Only one process in the cycle detects the deadlock, simplifying the resolution.

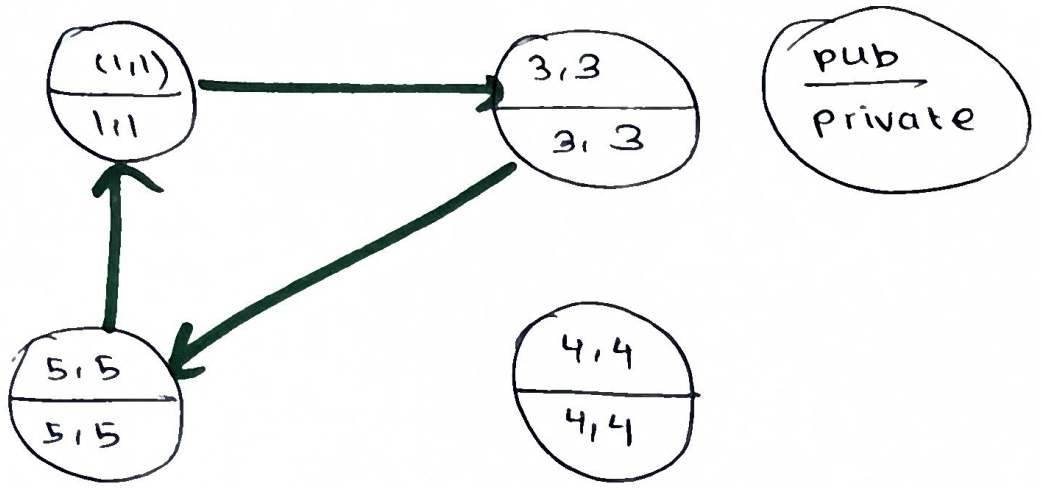
4. Resolving Deadlock

- The process that detects the deadlock declares it and resolves it by:
    - Aborting itself or one of the processes in the cycle.
    - This breaks the cycle and restores the system to a functional state.

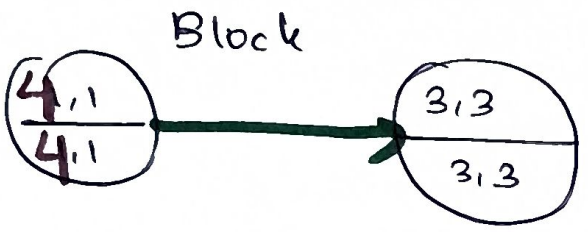
Mitchell Merilt Algorithm

→ each label is (count, pid)

→ Assume that the 4 processes are, and the edges between them are in green.

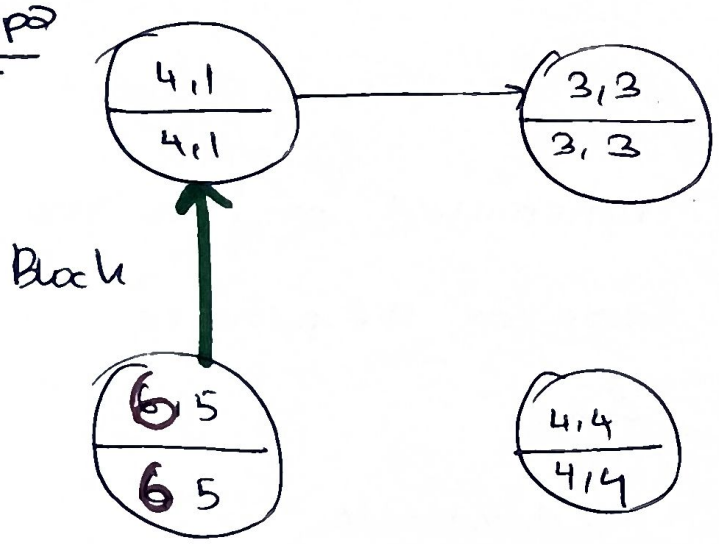


Step1

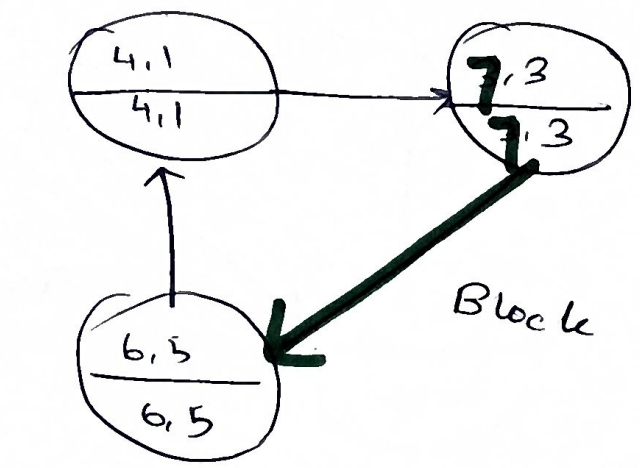


(1,1) sends a request to (3,3) make the count of the sender greater than that of the receiver

Step2



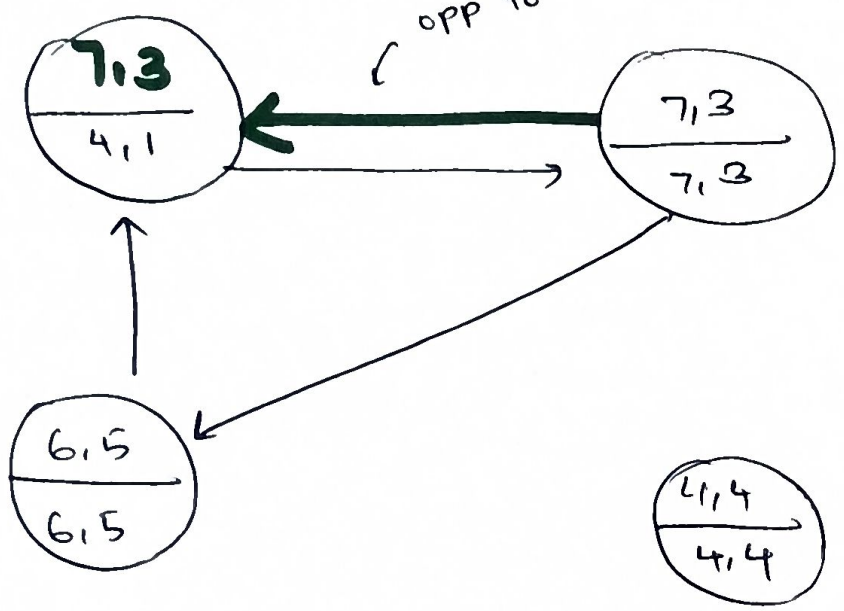
Step3



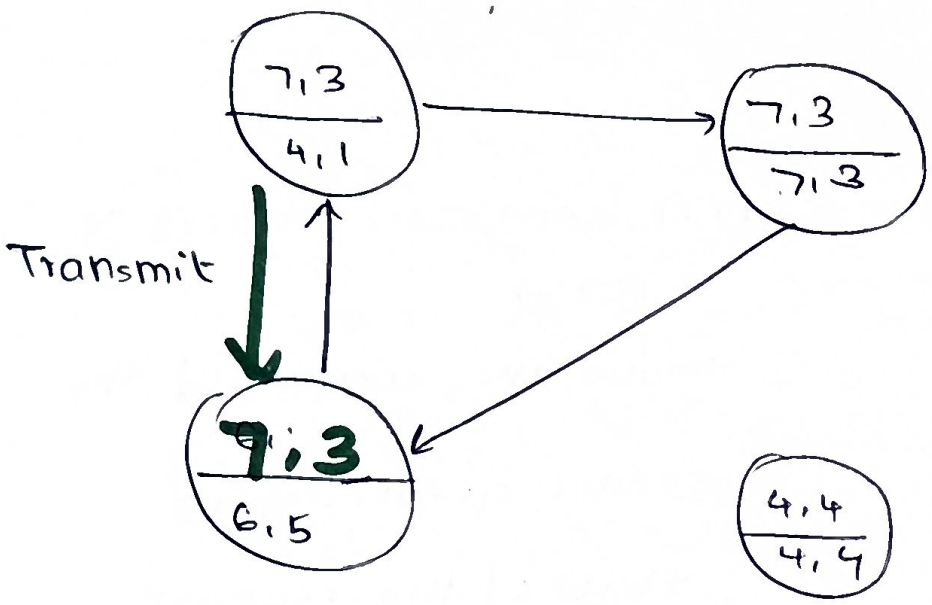
Step 4

Transmit

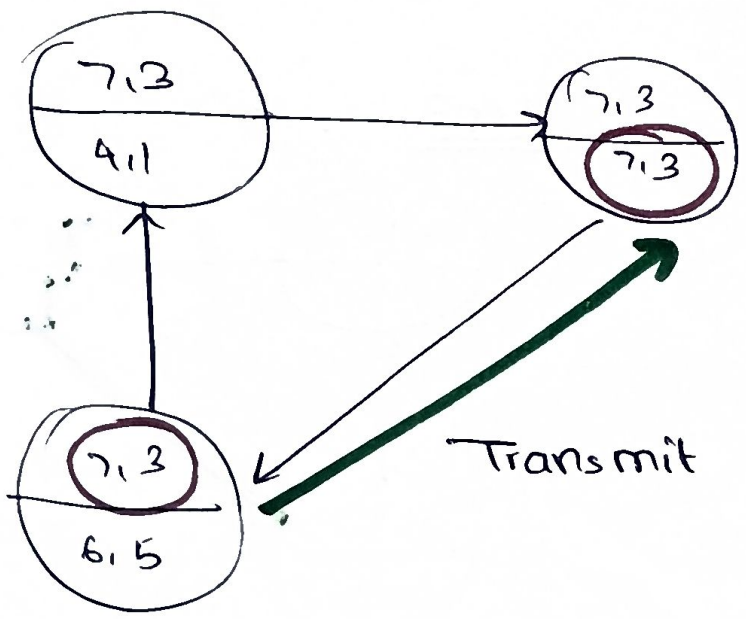
opp to direction of WFG



Step 5



Step 6

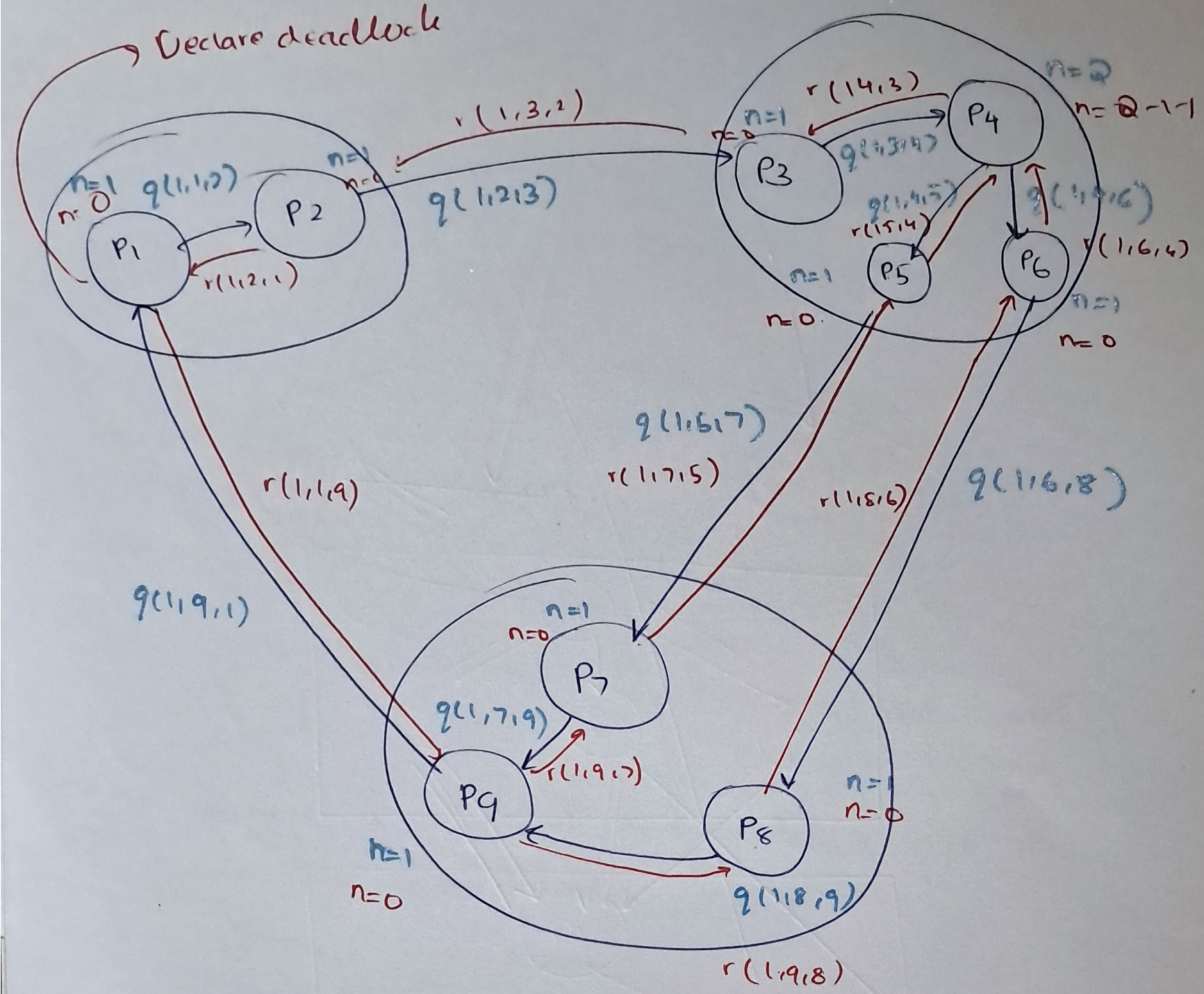


Transmitted (c,pid) is the same as the private label

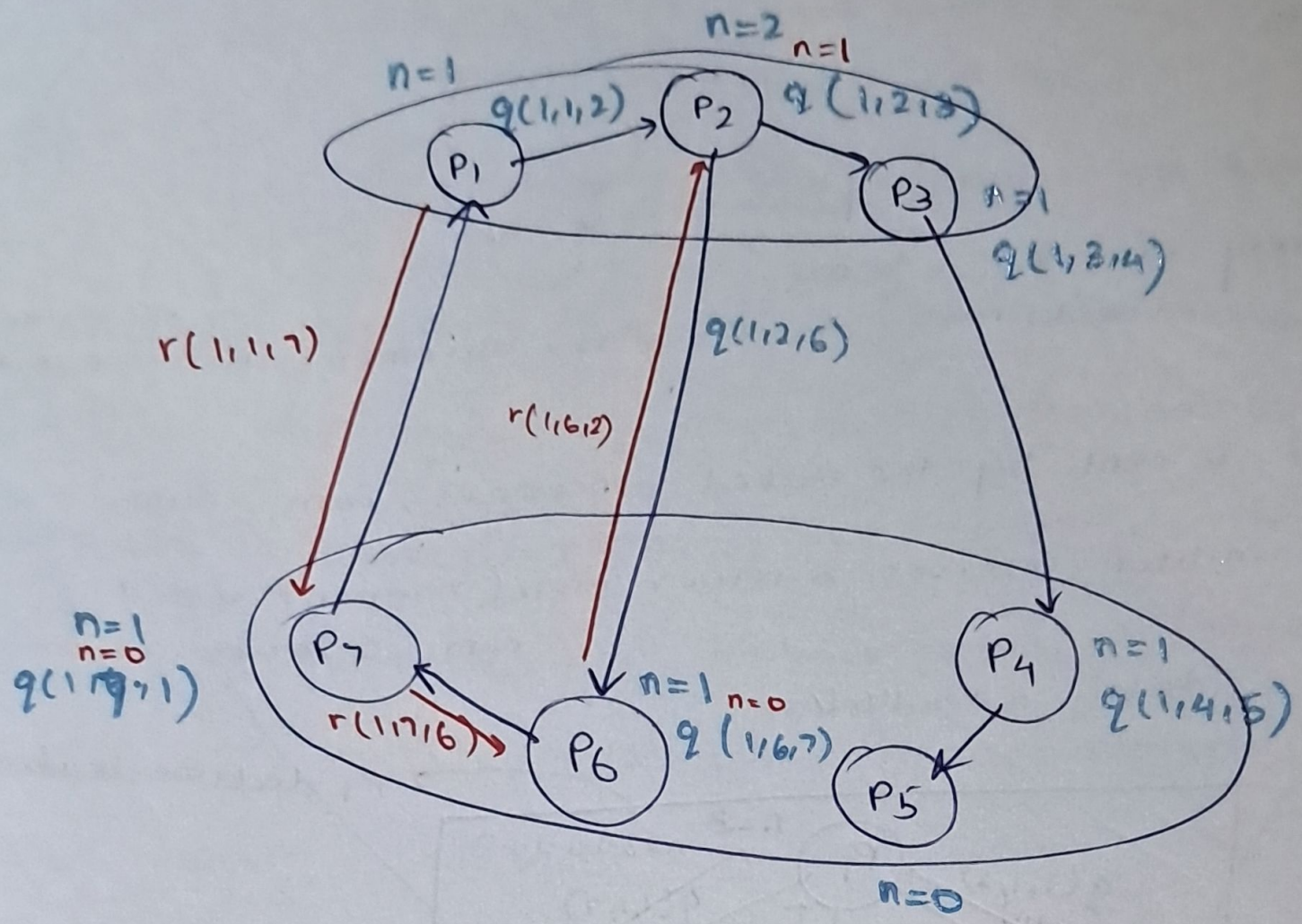
⇒ deadlock

Example1

CHANDY MISRA HAAS OR MODEL



Example 2



P5 does not send a reply as it is not blocked

There is never a reply from $P_5 \rightarrow P_4 \rightarrow P_3 \rightarrow P_2 \rightarrow P_1$

=> There is no knot

=> There is no deadlock.