Global states.

Clock ⇒ ordering very important
Stakes ⇒ consistency very important

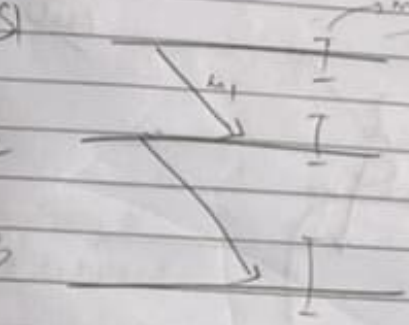
ATM

Bank Server 10,000

Msg1: Withdrawn and then deducted
Msg2: Deduct and then withdrawn

$GS = \bigcup_{i=1}^{n} LS_i$

Global state is the union of all local states
↳ send and receive events



LS_1 = {send(m1)}

LS_2 = {send(m2), rec(m1)}

LS_3 = {rec(m2)}

Is this system consistent?

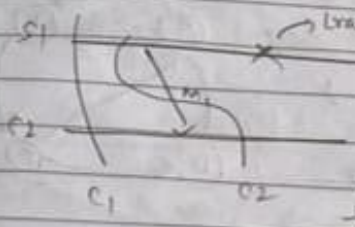
If every send has its receive then it
Called ⇒ strongly consistent

To record this
each time one captures the Snapshot
others should freeze
⇓
But this will affect the performance

Not possible practically

∴ We only consider consistent global state.

Case: Inconsistency with Crash:



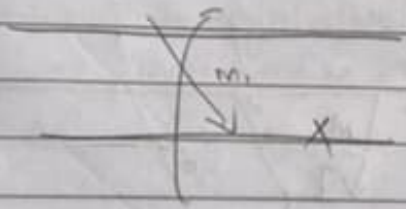
If there is a crash, there is no account of this send m, from S_1. But receive alone has happened

thus problem

So only receive recorded and send not recorded {inconsistent}

Motivation to have Consistency (ideal is strong Consistency)

Case 5: Consistency and crash.



If not received within stipulated time, retransmission of packets take place

∴ consistency is not disturbed.

should be proved but practically possible

To Prove: state change time is less than the time taken for the info to travel back to S_i.

Single request model ⇒ hold and wait. → only for this theory works.

OR request model ⇒ simultaneously req for 2, atleast one is required.

Another also is required

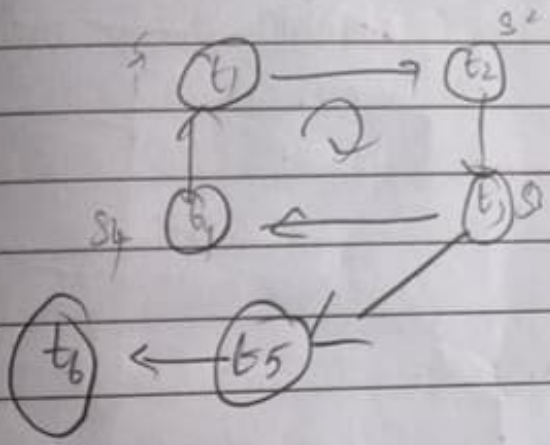
Knapp's classification: ① Path pushing: path info strings.

② Edge chasing: Instead of sending whole string, small probe messages are sent (AND model).

③ Diffusion computation: used for OR model.

④ Global state based: GSR algorithm to find deadlock.

OR based model: Knots



In AND model, a cycle indicates deadlock ⇒ declare deadlock

OR request model:- No matter how many req made, atleast one free.

Distributed mutual Exclusion

Expected
- ↳ Fairness
- ↳ Liveliness
- ↳ Consistent

how to ensure fairness

- ↳ Non Token Based Algorithms (Distributed Queues)
- ↳ Token Based Algorithms (Raj hpdgs similar) — Suzuki Kasami

Distributed Queues:
⇒ Each process has its own queue and will be broadcasted through Lamport logical clock.

⇒ order in which request has been made.

Lampart logical — Ricart

Lamport's algorithm:

3 phases:
1) Request
2) Reply
3) Release

Request: when a node wants to enter critical section
→ request/sequence number
→ Process ID
(current process num + 1.)

It will be enqueued in its own queue.
It is then broadcast to everyone else.
↳ to maintain consistency.

Critical section:
A piece of code that handles code to shared resource.

Ricart - Agarwala's Algorithm (Non-Token).

Reduce $3(N-1)$ messages $\Rightarrow$ to $2(N-1)$ messages

No release phase.

P1	P2	P3
(1,1) (2,1)	(1,1) (2,2)	(1,1) (2,2)
Request (1,1)	rec req (1,1) request (2,2) rep (1,1)	rec req (1,1) rep (1,1) rec req (2,2) rep (2,2)
rec req (2,2) rec rep [1,1] from P3 rec rep (1,1) from P2 exec CS rep (2,2)	rec rep (2,2) from P3 rec req (2,2) from P1 exec CS	
(1,1) (2,1)	(1,2) (2,2)	(1,1) (2,2)
request (1,1) rec req (1,2)	request (1,2) rec req (1,1) rep req (1,1)	rec req (1,1) rec req (1,2) rep (1,1) rep (1,2)
rec rep (1,1) from P2 rec rep (1,1) from P3 exec CS rep (1,2)	rec rep (1,2) from P3 rec rep (1,2) from P1 exec CS	

Concurrent request

Ensure fairness: When a req is pendg, code should be written to prevent more requests.

If the same proc has taken and keeps asking for request

P1

R1 | 0 | 0 | 0 |

T | 0 | 0 |

R[i] = Seq + 1

Acquire (Seq, P=0)

R1 | 1 | 1 | 0 |

T | 0 | 0 |

exec CS

T | 1 | 0 | 0 |

T[P] = R1[2]

R1 | 2 | 1 | 0 |

R1 | 2 | 1 | 1 |

P2

R2 | 0 | 0 | 0 |

R2 | 1 | 0 | 0 |

R2 | 1 | 1 | 0 |

T | 1 | 0 | 0 |

exec CS

T | 1 | 1 | 0 |

R2 | 1 | 1 | 0 |

R2 | 2 | 1 | 1 |

P3

R3 | 0 | 0 | 0 |

T | 0 | 0 | 0 |

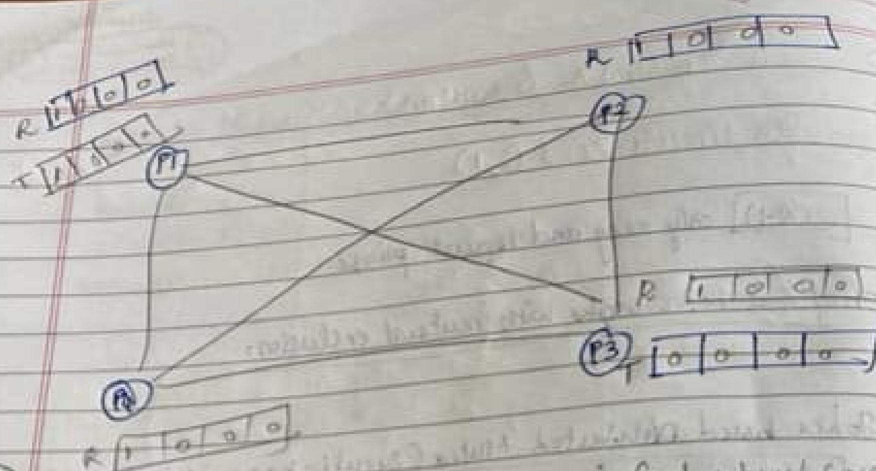
R3 | 1 | 0 | 0 |

T[x] = R3[i] - 1

R3 | 1 | 1 | 1 |

R3 | 1 | 1 | 1 |

R3 | 2 | 1 | 1 |



Idle token: node possessing token but no request ⟹ Here P3 has token but does not need it, T[1] = 0

Why check for maximum? (and not just store?)

outdated request problem ⟹ If after some time, if no reply, again request is sent ⟹ if old req reaches again, that should not be overwritten

So token is passed from P3 to P1 as $T[1] == R_3[1] - 1$

This won't lead to starvation

P1	P2	P3	P4
R [0|0|0|0]	R [0|0|0|0]	R [0|0|0|0] T [0|0|0|0]	R [0|0|0|0]
R [1|0|0|0] passes (1,1)	R [1|0|0|0]	R [1|0|0|0] $R_3[1] = T[1]+1$	R [1|0|0|0]
T = [0|0|0|0]			R [1|0|0|1]
Exec CS T [1|0|0|0] R_1 [1|0|0|1] $R_1[4] = T[4]+1$	R [1|0|0|1]	R [1|0|0|1]	
			T [1|0|0|0] Exec CS T [1|0|0|1]

Why mutual Ex? ⟹ only the node which has token can enter.

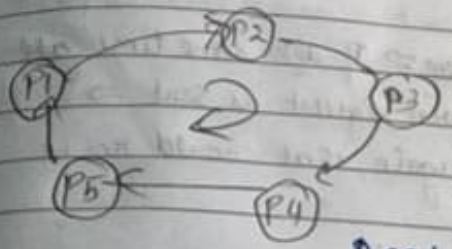
(ii) There is a problem in DOS attack

After timeout send (2, 1)

$[2(N-1)]$ only reply and request phase

Proof that both works with mutual exclusion.

Token based Distributed Mutex (Suzuki-kasami's Algorithm).



The node which posses the token does not know which node requires the token.

Disadvantage: Starvation.

Data structures used: R - request vector } of size n (every node has an instance)
T - Token vector
↳ 1 instance

Request:

Incr $R_i[i]$ and broadcast

Response:

on receiving broadcast message (every node)

$R_j[i] = \max(R_j[i], R_i[i])$

If a node has token, for all k

if $T[k] == R_i[i] - 1$

pass the token.

rec rel req (1,2)	rel req (1,2)	rec rel req (1,2)

If DDOS attack on Process 2; other process could not enter.

If a process is already in queue, no more new req to the process is allowed } FAIRNESS

If seq number rec < prev rec seq no reject

3(n-1) msgs communicated ⇒ Reduce this in the next algo.

Case2: Suppose P1's request more.

P1	P2	P3
req (1,1)	rec req (1,1) req (2,2) rep req (1,1)	rec req (1,1) rep req (1,1) rec req (2,2) rep req (2,2)
rec req (2,2) rec rep (1,1) from P3 rec rep (1,1) from P2 exec CS rel req (1,1) req (2,2)	rec rep (2,2) from P3	
	rec rel req (1,1) rec rep (2,2) from P1 exec CS rel req (2,2)	rec rel req (1,1)
rec rel req (1,2)		rec rel req (2,2)

Case3:

P1	P2	P3
(1,1)	(1,2)	
req (1,1) rec req (1,2)	req (1,2) rec req (1,1)	rec req (1,1) rec req (1,2) rep req (1,1) rep req (1,2)
rec rep (1,1) from P3	rec rep (1,2) from P3 rep req (1,1)	
rec rep (1,1) from P2 exec CS rel req (1,1) rep req (1,2)	rec rel req (1,1) rec rep (1,2) from P1 exec CS	rec rel req (1,1)

(1,1) is given preference based on FCFS

Starvation

dropping

Case3:

rec rel req (1,2)

If DDOS co

If n no mo allo

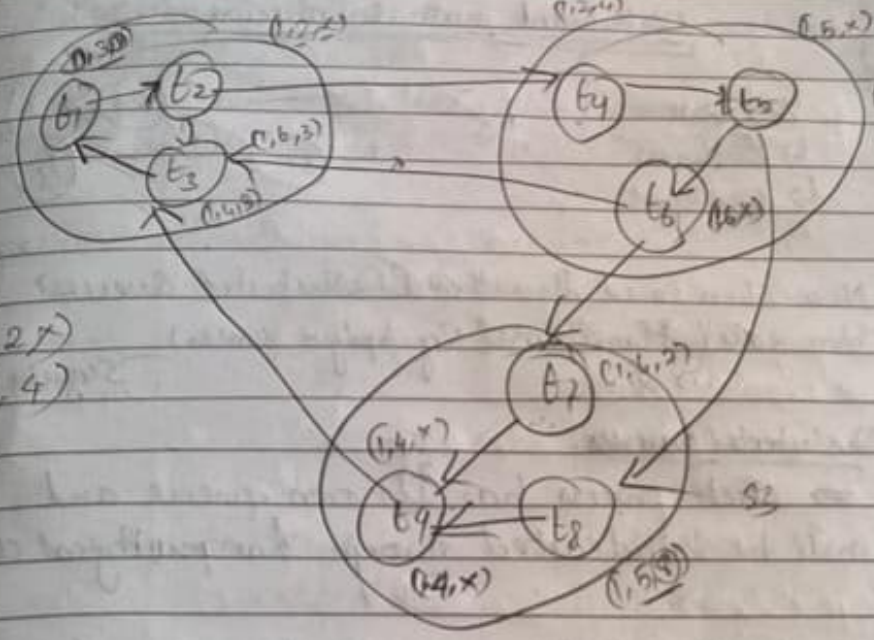
If M

3(n-1)

Reply: • All the other systems send reply in the order to which they received.
• Reply is given only if the system doesn't want critical section

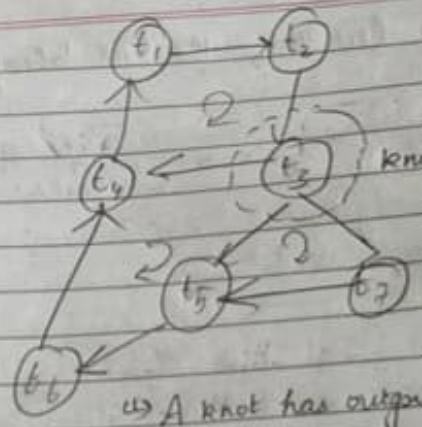
Release: Broadcast to release the resource.

P1	P2	P3
Req (1,1)		
	Rec req (1,1)	Rec req (1,1)
	Rep req (1,1)	Rep req (1,1)
Rec rep (1,1) P2		
Rec rep (1,1) P3		
Execute CS.		
Release req (1,1)		
	Rec release req (1,1)	rec release req (1,1)
	del req (1,1)	del req (1,1)



(1,2,x)
(1,2,4)

(1,x)
|



Multiple cycles present

Query and reply will be both each

All knots have cycles, but all cycles don't have knots

↳ A knot has outgoing edges, when we go through the all outgoing edge, everything will lead to its own edge

↳ Does not have any loose edge

With the presence of knots, deadlock is detected.

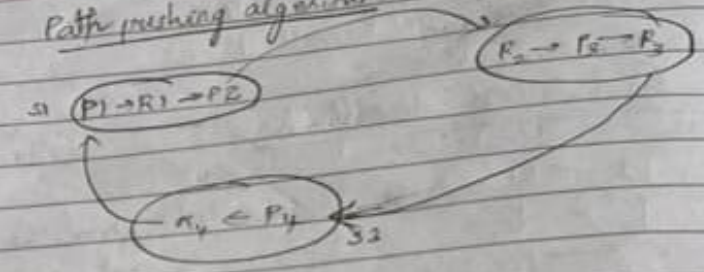
Algorithm: (i) Send queries from initiator to all outgoing edges

(ii) Send reply only if the node is waiting for already visited

(iii) Send reply from a node only if it receives replies for all of the queries

Assumption: Time taken for transmission is less than time for sending query messages.

Path pushing algorithm:



Fairness should be maintained

construct a string:

$t_1 \to ext \to t_2 \to ext \to t_3 \to ext \to t_1$

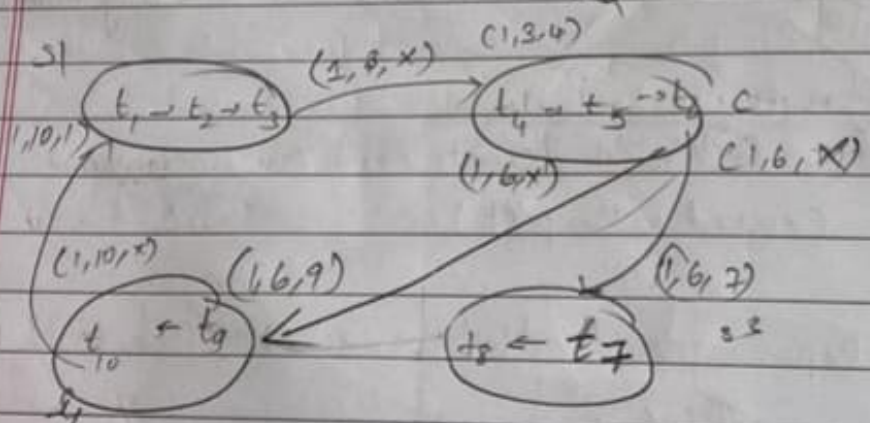
level / phantom deadlock

taught in distributed mutual exclusion

↳ when delay in communication channel, deadlock is not correctly identified.

↳ This is computationally intensive
↳ more time constructing strings / concatenation etc.

Better solution? Edge chasing algorithm:

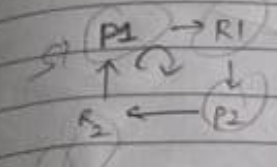


(i) Check for local deadlock ↝ treated same as global deadlock
If there is local deadlock ⇒ declare

(ii) When there is a waiting edge
If i and j are same then deadlock is detected.

Application of Recording global state is finding deadlock.

Distributed deadlocks:



In distributed system, the problem is that each of the processes are in separate systems

∴ Delay in req.
∴ Losing Consistency

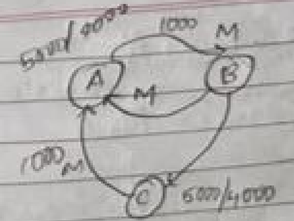
Traditional: Centralized system

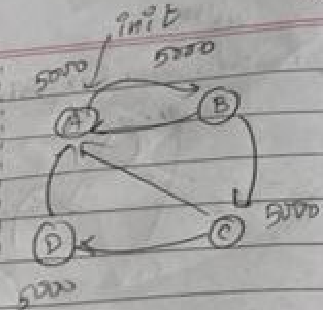
Disadv: → one dedicated server for deadlock
→ Incoming traffic huge channel
→ There cannot be a Centralized Godly
If any one can invade any task

Phantom deadlock

a deadlock which doesn't exist (or might have existed in the past).

Reports no deadlock, but deadlock could be there.





At t_1, A → B

At t_2, Initiation (init) happens

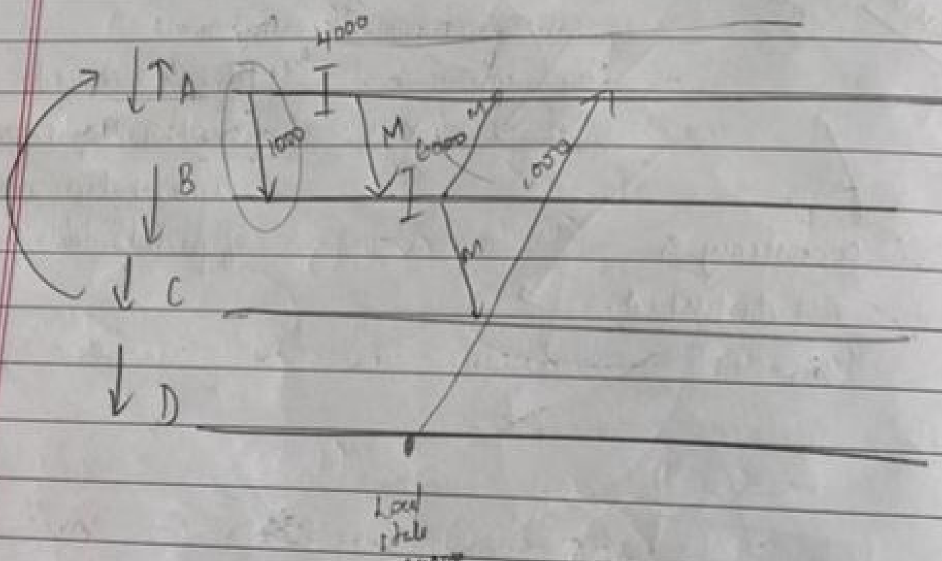
t_3 : D → A, M from B

t_4 : B → C, C → D, M from C

t_5 : C → A.

Sender

Recv: on recv Mark,
if any msg precedes
marker, integrate

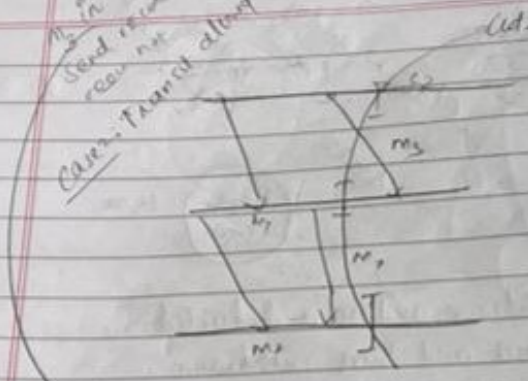


A → B

stop!
message
preceded
by
marker

m_2 in transit
Send recorded
recv not recorded

Case: Transit allowed



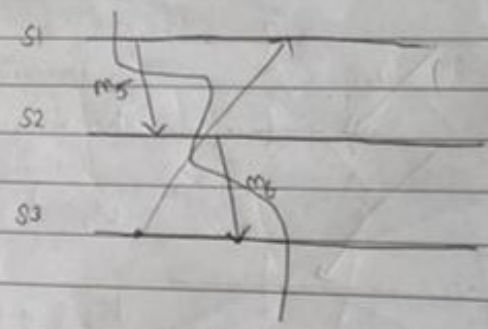
Consistent
⇓
If every receive should have corresponding send

$LS_1 = \{$ Send(m_3) $\}$

$LS_2 = \{$ Send(m_1) $\}$

$LS_3 = \{$ recv(m_1) $\}$

Case 3: Inconsistency



Consistent partially but is still inconsistent

$LS_1 = \{$ $\}$

Inconsistent global state

$LS_2 = \{$ receive(m_5) $\}$

$LS_3 = \{$ Send(m_4), receive(m_5) $\}$

22/9/2023

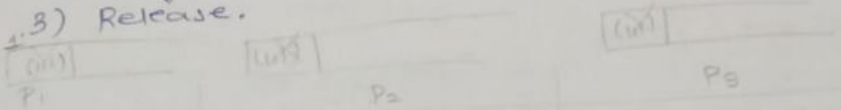
Distributed Mutual Exclusion :-

1. Non token based (Distributed Queues) — lamport, Ricart
2. Token based algorithm.

lamport's Algorithm :-

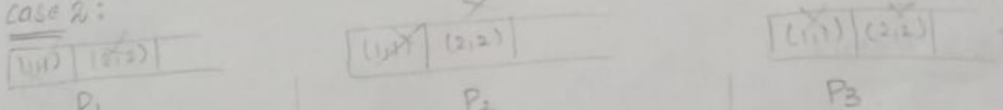
1) Request. Req (Sn , Pid)
2) Reply.
3) Release.

Case 1:



P_1	P_2	P_3
1. Req (1,1)	Rec req (1,1)	Rec req (1,1)
2.	Rep req (1,1)	Rep req (1,1)
3.		
4. rec rep (1,1) : P3		
5. rec rep (1,1) : P3		
6. exec CS		
7. Rel req (1,1)	rec rel (1,1)	rec rel (1,1)
8.	del req (1,1)	del req (1,1)

Case 2:



P_1	P_2	P_3
req (1,1)		
	rec req (1,1)	rec req (1,1)
	req (2,2)	rep req (1,1)
rec req (2,2)	rep req (1,1)	rec req (2,2)
rec rep (1,1) : P3		rep req (2,2)
rec rep (1,1) : P2	exec rep (2,2) : P3	
exec CS		

rel req(1,1)	rec rel req(1,1)	rec rel req(1,1)
	del req(1,1)	del req(1,1)
rep req(2,2)	rec req(2,2) : P1	
	exec CS	
	rel req(2,2)	rec rel req(2,2)
rec rel req(2,2)		del req(2,2)
del req(2,2)		

Case : 3

P1	P2	P3
req (1,1)	req (1,2)	
rec req (1,2)	rec req (1,1)	rec req (1,1)
		rec req (1,2)
rec rep(1,1) : P3		rep rep (1,1)
	rec rep (1,2) : P3	rep req (1,2)
rec rep (1,1) : P2	rep req (1,1)	
exec CS		
rel req(1,1)		
	rec rel req(1,1)	rec rel req (1,1)
rep req (1,2)	del req(1,1)	del req (1,1)
	rec rep(1,2): P1	
	exec CS	
rec rel req(1,2)	rel req (1,2)	
del req (1,2)		rec rel req (1,2)
		del req (1,2)

DOS Attack

request will be floodad continuously.

In case 3 :

Both try to enter critical section then two processid come matter. P1 will be enter into CS first

n system

n-1 request, reply, token

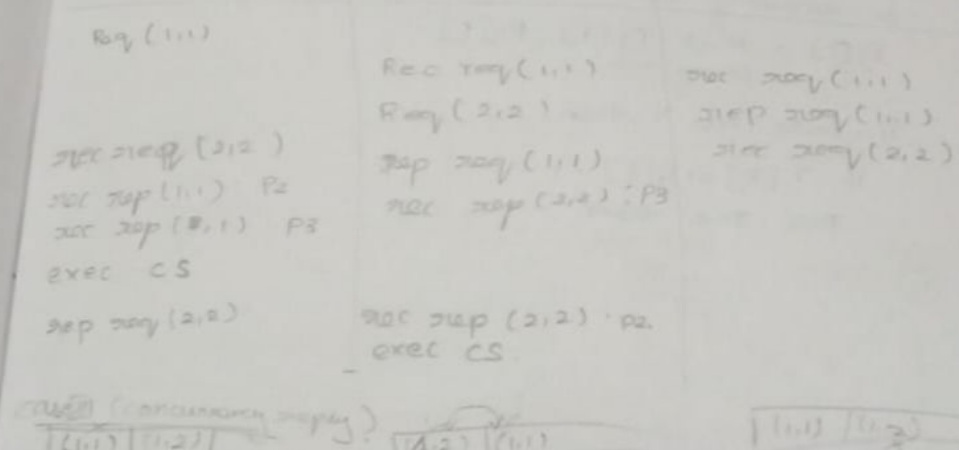
n x n-1 message are required.

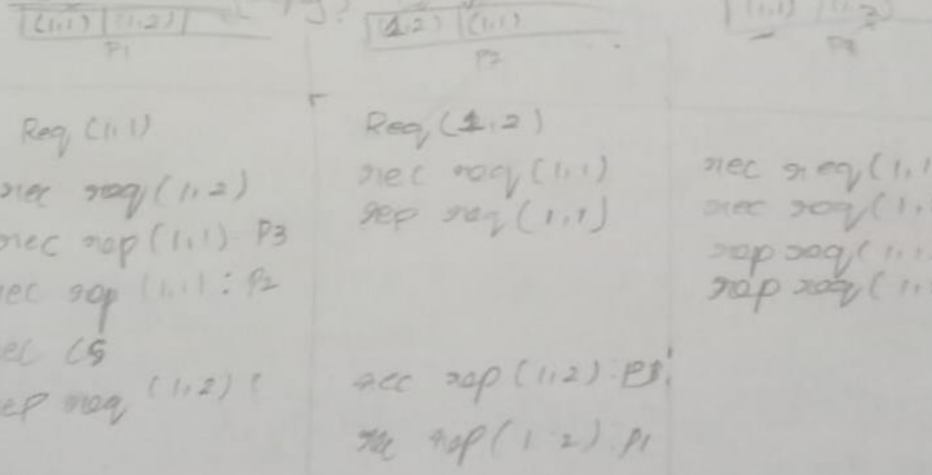
25/09/2023

Ricart - Agarwala's Algorithm (Di. Mutex)

→ It won't wait any reply message. It delete the msg after send reply.

3(N-1) => 2(N-1)





26/09/2023

Token based D-muter (Suzuki- kasami's Algorithm)

Data Structures used

1. R - Request vector (array)
2. T - Token vector (Long one integers)

Size - n

Request:

Inc $R_i[i]$ & broadcast

Response:

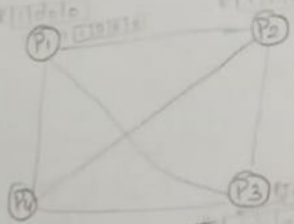
on receiving broadcast msg

$R_j[i] = \max(R_j[i], R_i[i])$ (every node)

if a node has T, then

if $T[k] == R_j[i] - 1$

pass the token



P_1	P_2	P_3	P_4
R_1 0 0 0 0	R_2 0 0 0 0	R_3 0 0 0 0	R_4 0 0 0 0
		T 0 0 0 0	
R 1 0 0 0	R 1 0 0 0	R 1 0 0 0	R 1 0 0 0
		$R_3[i] == T[i]+1$	
T 0 0 0 0			R 1 0 0 1
exec CS	R 1 0 0 1	R 1 0 0 1	
T 1 0 0 0			T 1 0 0 0
R 1 0 0 1			exec CS
$R_1[4] == T[4]+1$			T 1 0 0 1

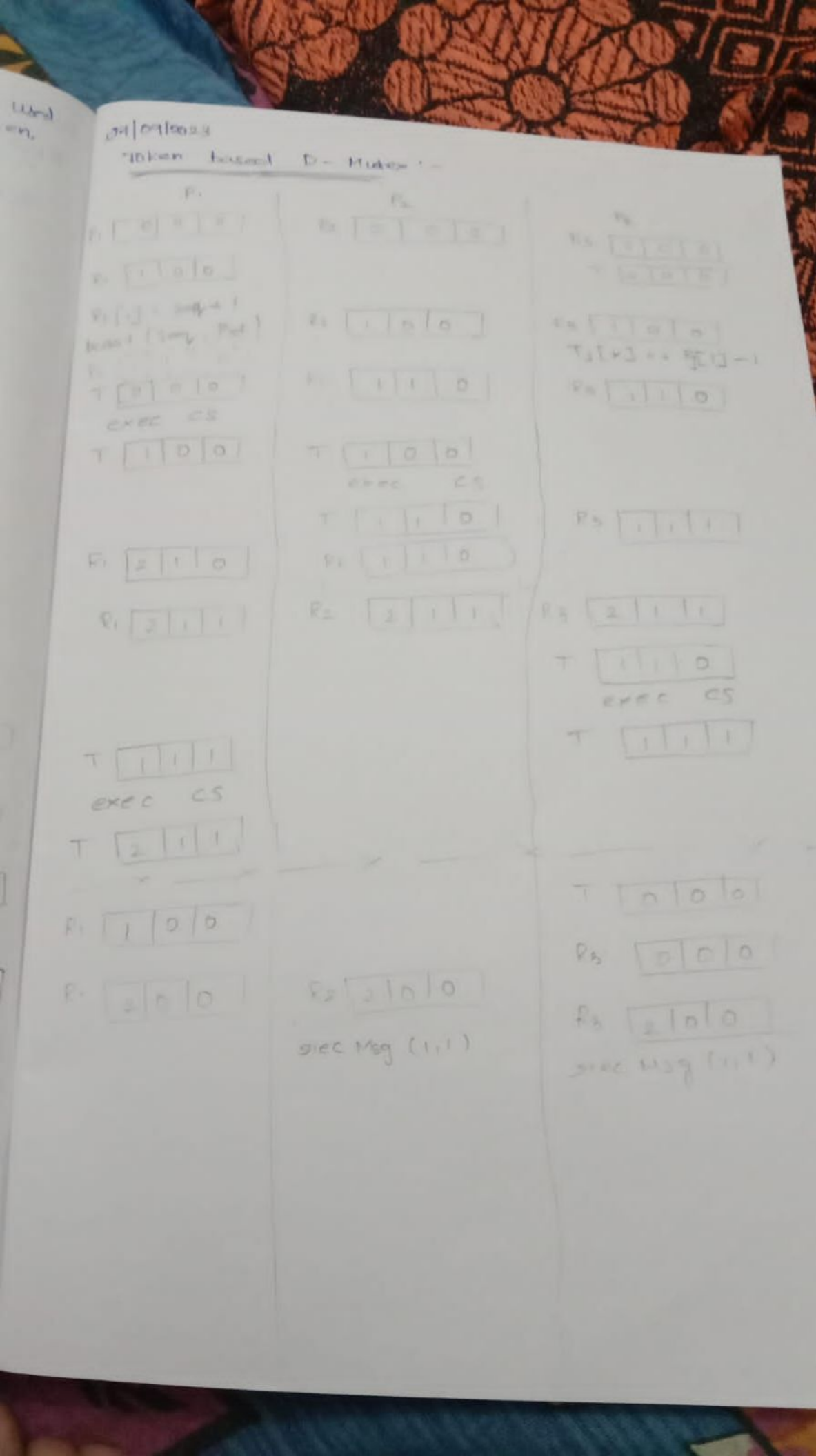
ideal token: not used token.

07/09/2023

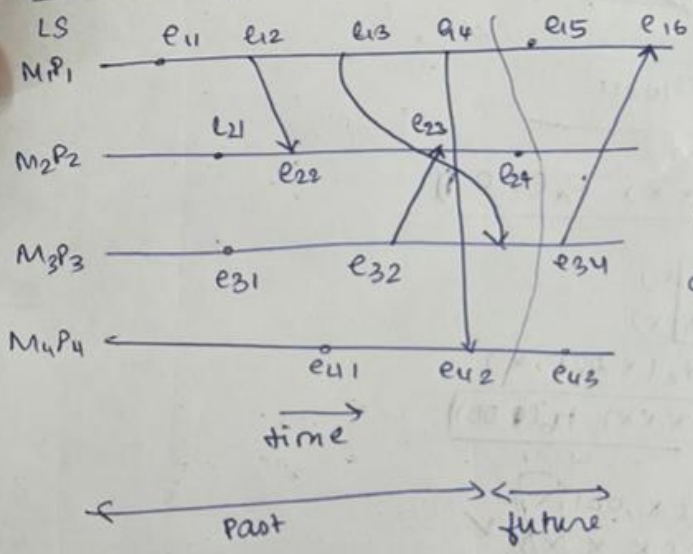
token

04/09/2024

Token based D- Mutex :-



Local and global states of dist systems



LS_i Local state of P_i

$LS\ C_{ij}$

$GS = \Sigma LS_i \cup \Sigma LS\ C_i$

↓

Global state

Local state of every process

& of msgs in channel.

- Local state will record local event before that state and the message that has been sent

→ Cut is used to 'to tell if system is in

CGS - consistent (o:udy recorded recently not)

SCGS - strongly consistent (both recorded)

mCGS - inconsistent.

Chandy - Lamport Algorithm

Marker msg - seperate msgs that recorded & not recorded

2nd marker msg - local snapshot.

Distributed Mutex Algorithms

Non-token based approach: — send req to all proc. & once they say yes, go into critic section

Lamport's DMutex Algo.

Ricart Agrawala's DMutex Algo.

Token level DMutex Approach — Unique token needed to access critical section

Suzuki - Kasami's DMutex Algo

Quorum-based DMutex Approach – no need permission from all proc only from a group-quorum (subset of proc.)

Maekawa's DMutex Algo.

Lamport's DMutex Algo req, rep, release - msgs.

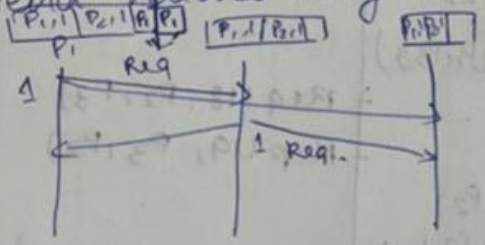
Request: ◦ send REQUEST(ts_i, i) ⇒ timestamp & pid
S_i place in request-queue;
S_j ◦ Receive, place in req queue; and send REPLY. order in queue by inc. timestamp.
send reply if own timestamp is > recvd ts.
else it won't send reply.

Critical section:
L1: S_i rcvd ts larger than its own from all other
L2: S_i's req. front of the queue

Release:
S_i: Remove req. from queue. Broadcast RELEASE +ts msg
S_j: Remove req from queue after reching msg.
If own req comes to top - enable CS. to enter

- Safety, Liveness, Fairness – Mutex Property. ✓
- 3(N-1) message complexity.
- T- Sych. delay.

-Sometime when proc removes its req from queue, its own msg comes to front of the queue - no need to send release msg and start req all over again.



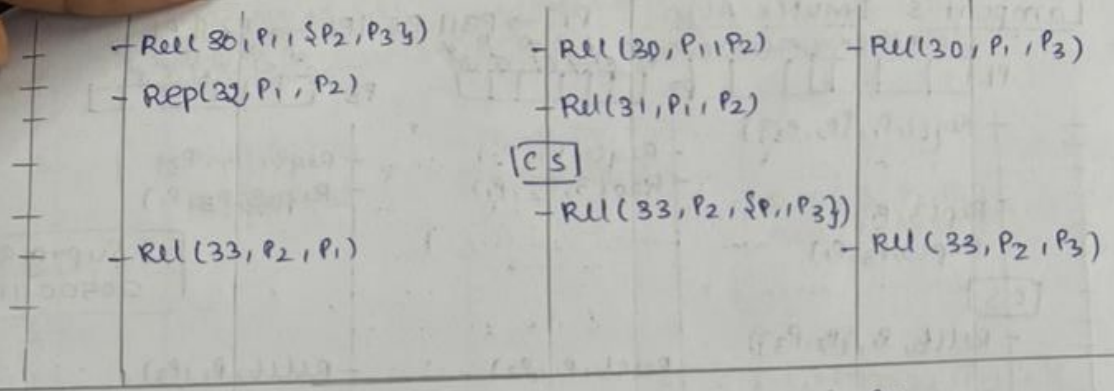
follow total order if same timestamp

Example:

$P_3 \longrightarrow P_1 \,||\, P_2 \rightarrow P_2$

Each proc maintains a FIFO queue

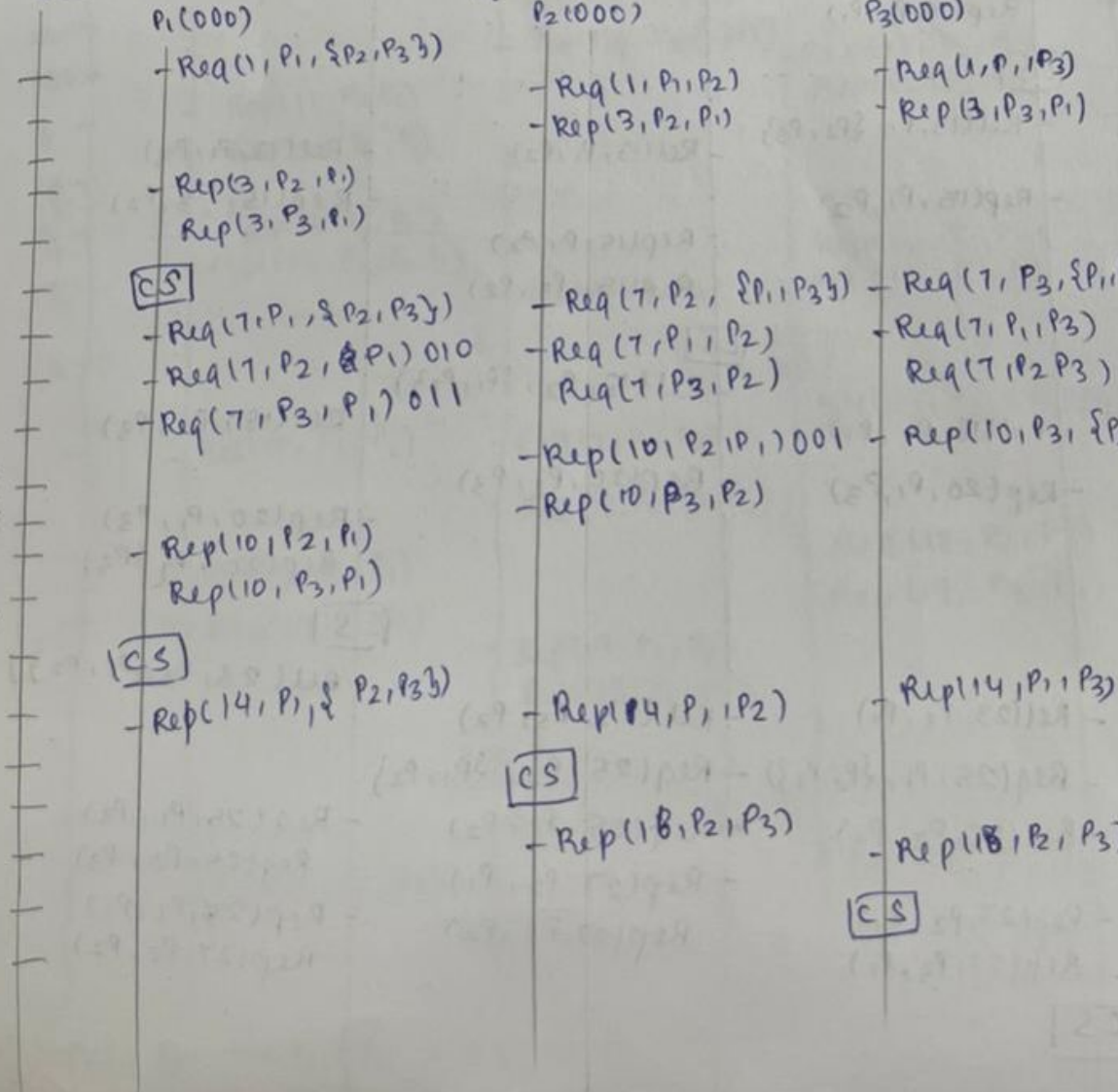
Messages: Req (timestamp, S, D)
Rep
Rel

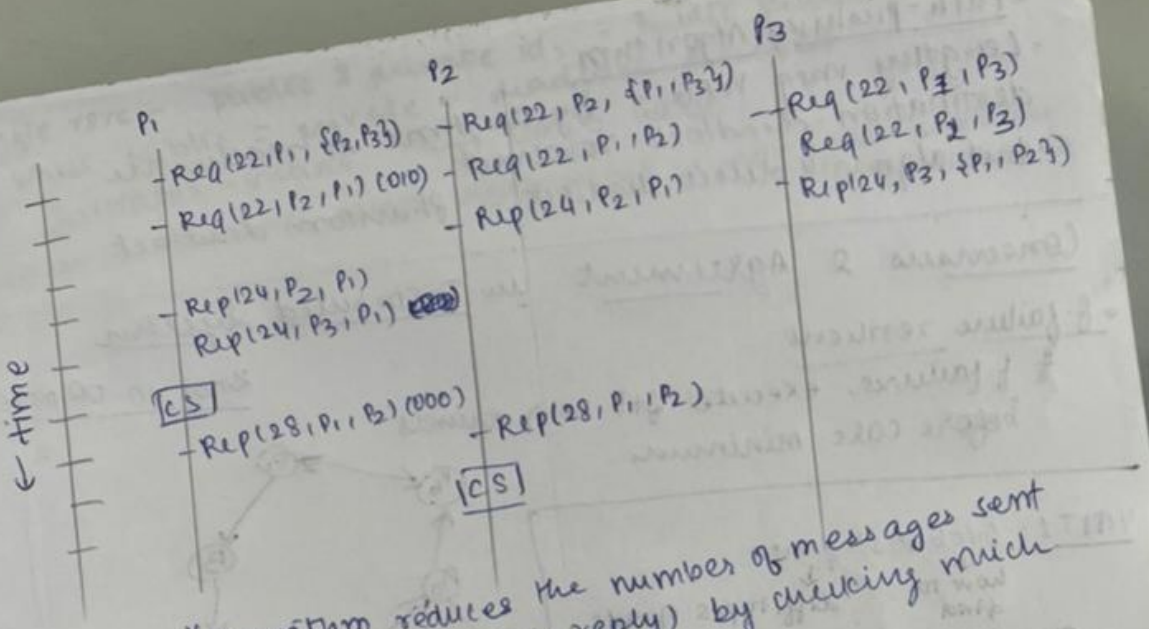


Lamport's algorithm achieves mutual exclusion by sending req, rep and rel msgs accordingly.

RICART AGRAWALA ALGORITHM:

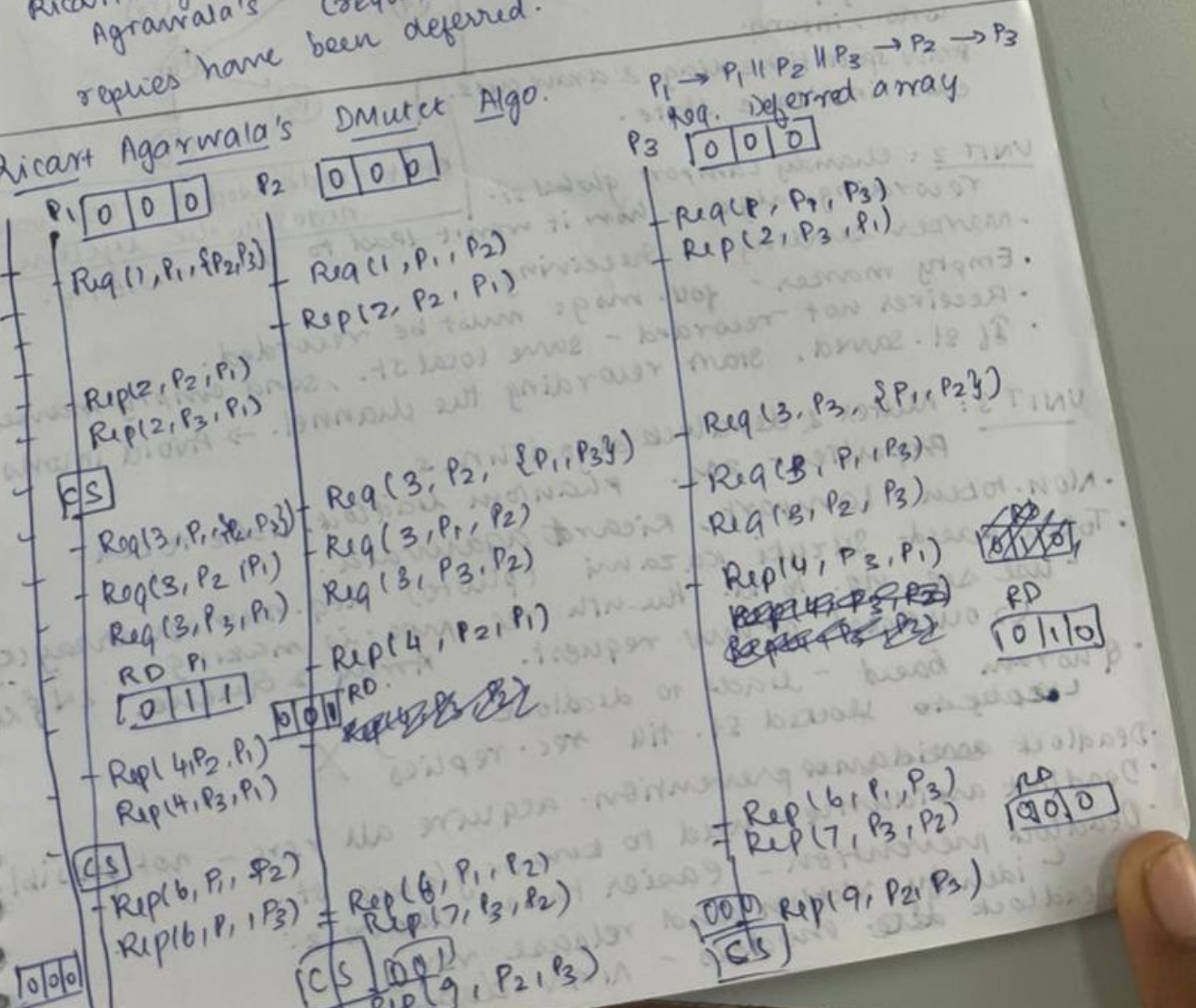
$P_1 \rightarrow P_3 \,||\, P_2 \,||\, P_1 \rightarrow P_2 \,||\, P_1$ RD → 000





Ricart Agrawala's Algorithm reduces the number of messages sent (request, reply) by queuing which replies have been deferred.

Ricart Agarwala's DMutex Algo.



- Path-pushing Algorithm:
- Lengthy msg WFG - by the time it reaches the destination - deadlock resolved.
- But algo will detect deadlock - Phantom deadlock.

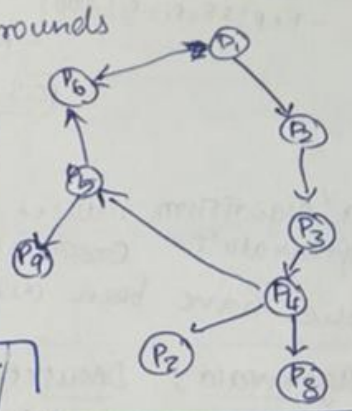
Consensus & Agreement in Distributed System

- f-failure resilient:
  If f failures, execute f+1 no. of rounds before calc. minimum.

UNIT 1: Global st. & cuts
- how to find
- diff. types of cuts.

Strongly cons., incons.
Draw space time diag. & draw cuts.
cuts indicate state.

Knot in OR Model



Knot - node used to reach every node in the system.

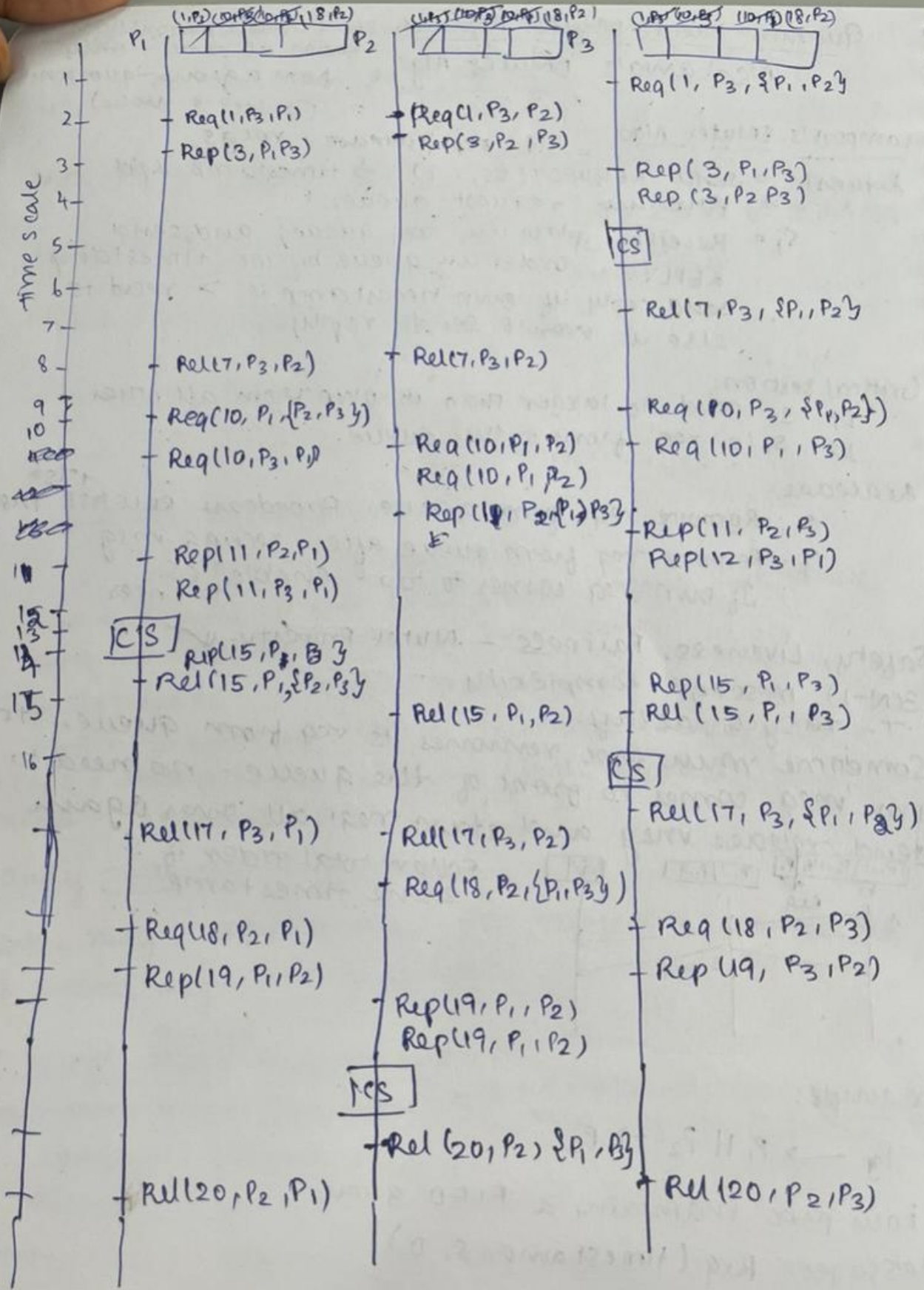
UNIT 2: Chandy Lamport global st. recording algo - how it won't lead to inconsistency.
- marker sending & receiving.
- Empty marker - foll. msgs must be recorded.
- Receiver not recorded - save local st., send empty marker
- If st. saved, start recording the channel. → Avoid incons.

UNIT 3: Mutex & deadlock algorithms.
Properties - 2M. phantom deadlock, idle token.
- Non-token - Lamport, Ricart agarwala.
- Token based - Suzuki Kazami (photo) Req. number array (RN) - use seq. no. to id. the nth req. proc is making
  ↳ outdated or new request. Array & Queue. LN & RN
- Quorum based - leads to deadlock. ↩ blocked st. till rec. replies X
- Deadlock ~~avoidance~~ prevention - acquire all rsrc - not possible
- Deadlock avoidance - need to know global st.
- Deadlock prevention - easier than above 2.
  ↳ identify victim and release rsr.

Deadlock ~~detection~~ models - New classification.

- Single rs
  when
  Suri
- Learn

- Single rsrc - public & private id. - public shared
  when public = private - deadlock detected
  Initiator - victim. → Don't have
- Learn which are preemptive & not (path pushing
  edge pushing)



- HW: $P_1 \rightarrow P_1 \,||\, P_2 \,||\, P_3 \rightarrow P_1 \rightarrow P_2$

→ Suzuki kawasi Algo

Eg :→ P1 → P2 → (P3 || P4) (initial token with P3)

P_1	P_2	P_3	P_4
RN [0 0 0 0]	RN [0 0 0 0]	RN [0 0 0 0]	RN [0 0 0 0]
~~RN~~		LN [0 0 0 0]	
RN [1 0 0 0]	~~RN [1 0 0 0]~~	~~RN [1 0 0 0]~~	~~RN [1 0 0 0]~~
req (1, 1)			
	RN [1 0 0 0]	RN [1 0 0 0]	RN [1 0 0 0]
		~~R~~N [0 0 0 0]	
		Q = (1)	
		Send token to 1	
CS			
RN [1 0 0 0]			
LN [1 0 0 0]			
Q = ( )			
	RN [1 1 0 0]		
	req (2, 1)		
RN [1 1 0 0]		RN [1 1 0 0]	RN [1 1 0 0]
LN [1 0 0 0]			
Q = (2)			
Send tkn to 2			
	CS		
	RN [1 1 0 0]		
	LN [1 1 0 0]		
	Q = ( )		

__/__/____

P_1	P_2	P_3	P_4
		RN [1110]	RN [1101]
		req (3,1)	req (4,1)
RN [1110]	RN [1110]		RN [1111]
RN [1111]	RN [1111]	RN [1111]	
	LN [1100]		
	Q = [3,4)		
	send tkn to 3		
		CS	
		RN [1111]	
		LN [1110]	
		Q = (4)	
		send tkn to 4	
			CS
			RN [1111]
			LN [1111]
			Q = ( )