

CAT - 2

Portion

## Global State of a DS

↳ collection of local states of its components, i.e., the process & communication channels

State of process → content of processor registers, stack & local memory

State of channel → set of msgs in transit in the channel

send(m) → affects state of sending process & channel

recv(m) → affects state of receiving process & channel

$LS_i^{x_i} \rightarrow$  local state of process  $P_i$  after occurrence of event  $e_i^{x_i}$  & before event  $e_i^{x_i+1}$

$SC_{ij}^{x,y} \rightarrow$  state of channel bw processes  $P_i \& P_j$  consisting of all msgs sent by  $P_i$  upto  $e_i^{x_i}$  & which process  $P_j$  had not recv'd until event  $e_j^y$

$$GGS = \{ U_i LS_i^{x_i}, U_{j,k} SC_{j,k}^{y_j, z_k} \}$$

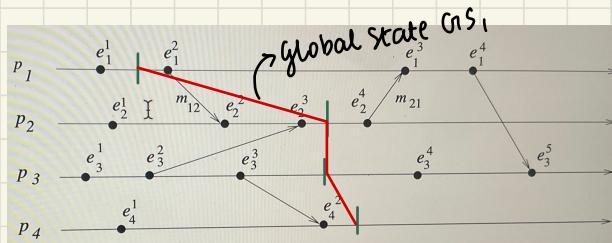
## Consistent Global States

↳ Should follow causality

↳ state should not record a receive before/without its corresponding send

$$GGS = \{ U_i LS_i^{x_i}, U_{j,k} SC_{j,k}^{y_j, z_k} \} \text{ is consistent iff}$$

$$\forall m_{ij}: \text{send}(m_{ij}) \notin LS_i^{x_i} \Leftrightarrow (\text{rcv}(m_{ij}) \in LS_j^{y_j}) \wedge (m_{ij} \notin SC_{ij}^{x_i, y_j})$$



$$GGS_1 = \{ LS_1^1, LS_2^3, LS_3^3, LS_4^2 \}$$

Inconsistent as:  $\text{send}(m_{12}) \notin LS_1^1$  but

$\text{rcv}(m_{12}) \in LS_2^3$

i.e.,  $P_1$  has not recorded send of  $m_{12}$  but

$P_2$  has recorded rcv of  $m_{12}$

causality violation

transit

One of the 2 Conditions for Consistency

↳  $\text{send}(m_{ij}) \in LS_i \Rightarrow \text{rcv}(m_{ij}) \in LS_j \oplus m_{ij} \in SC_{ij}$

↳  $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rcv}(m_{ij}) \notin LS_j$

Absence of Shared Memory leads to

Inconsistency) eg; Bank Withdrawal  
wrt state

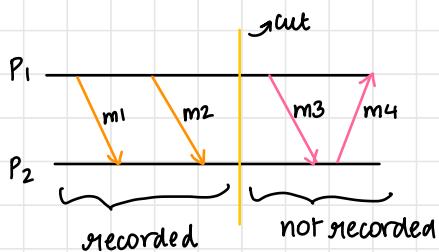
To solve this, we use global states  $\rightarrow$  Snapshot of local states

$$\text{Global State} = \bigcup_i^N (\text{local state}_i)$$

Global state recorded using snapshot  $\rightarrow$  only local states up till ss is recorded in Global state,  
other info abt actions are stored in local state until next ss

Snapshot in a cut : split time-space graph into 2

e.g :



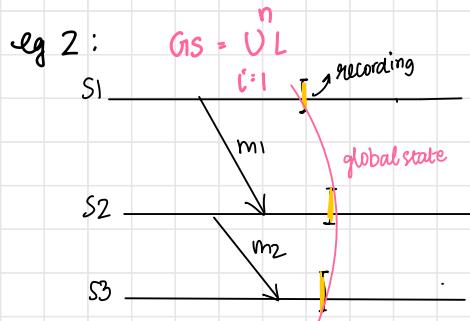
Localstates :  $LS(P_1) = \{ \text{send}(m_1), \text{Send}(m_2) \}$

wrt Cut  $LS(P_2) = \{ \text{rcv}(m_1), \text{rcv}(m_2) \}$

$\text{send}(m_3), \text{rcv}(m_3) \& \text{send}(m_4), \text{rcv}(m_4)$  are not recorded in global state

## Global States

Store states to ensure consistency & for rollback



$$\begin{aligned} LS_1 \rightarrow local\ state &= \{send(m_1)\} \\ LS_2 &= \{send(m_2), receive(m_1)\} \\ LS_3 &= \{receive(m_2)\} \end{aligned}$$

} strongly consistent

## Consistency

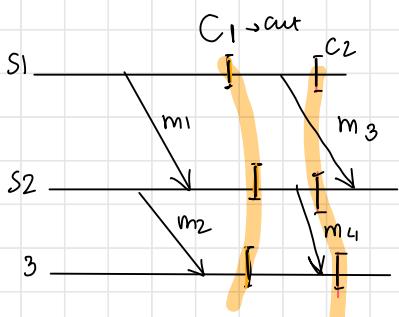
Is global state consistent?

↪ If every send has mcv  $\Rightarrow$  strongly consistent

How to implement strong consistency?

↪ NOT practically possible as stopping to record affects performance.

Hence, we strive for only consistency

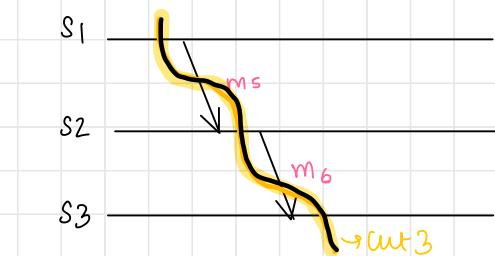


for C2

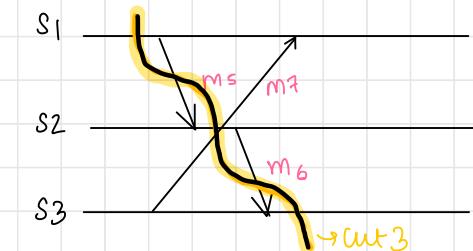
$$\begin{aligned} LS_1 &= \{send(m_3)\} \\ LS_2 &= \{send(m_4)\} \\ LS_3 &= \{recv(m_4)\} \end{aligned}$$

$recv(m_3)$ ? is in transit  
⇒ not strongly consistent

**Consistent state**: each receive should have corresponding send  
 ↪ has msg in transit (only send recorded)

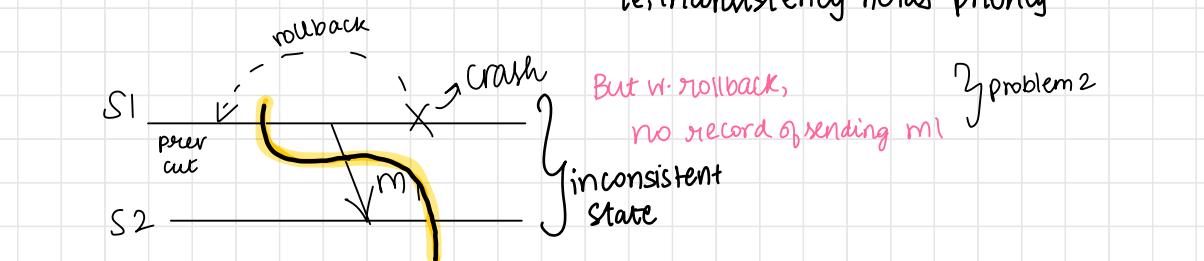


$LS_1 = \{ - \}$   
 $LS_2 = \{ rcv(m5) \}$  ↪ no corresponding send.  
 $LS_3 = \{ rcv(m6) \}$  ↪ send.  
 ⇒ inconsistent state  
 ie, rcv recorded before send logically incorrect - problem

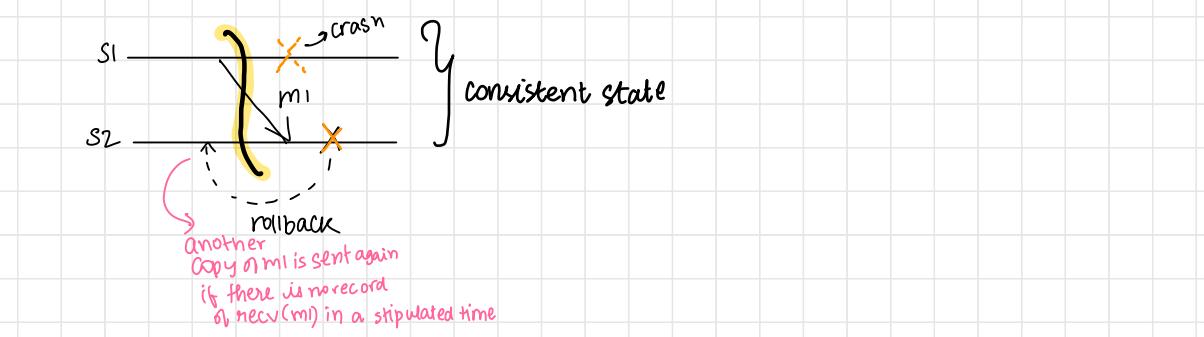


$LS_1 = \{ - \}$  inconsistency  
 $LS_2 = \{ rcv(m5) \}$   
 $LS_3 = \{ send(m7), rcv(m6) \}$  intransit

NOTE: Even if 1 instance of inconsistency ⇒ global state inconsistent  
 ie, inconsistency holds priority



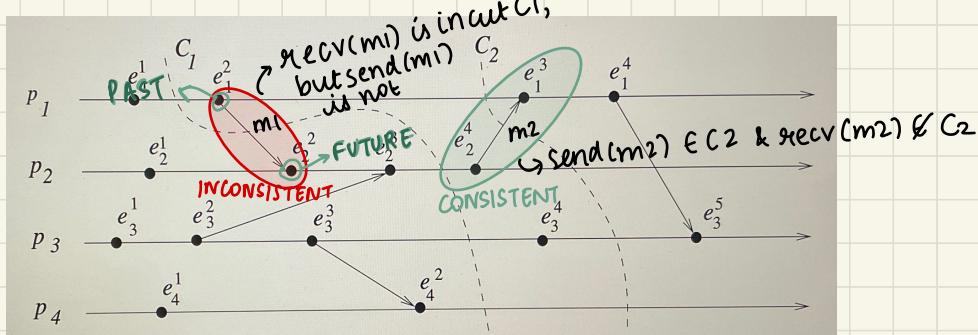
But w. rollback,  
 no record of sending m! ↪ problem 2



Cuts → Split space-time diagrams & thus the events into PAST & FUTURE

Every cut corresponds to a global state

left of cut      right of cut



NOTE: A cut is consistent if all msgs that cross the cut from the PAST to FUTURE are intra proc in corresponding fns

In  $C_1$ ,  $e_1^2 \rightarrow$  Past &  $C_1$  but  $e_2^2 \rightarrow$  Future  $\in C_2$

### Past Cone of an Event

An event  $e_j$  is only affected by events in its past i.e.,  $e_i \rightarrow e_j$

$$\text{Past}(e_j) = \{e_i \mid \forall e_i \in H, e_i \rightarrow e_j\}$$

let  $\text{Past}_i(e_j)$  be set of events in  $\text{Past}(e_j)$  that are on Proc  $p_i$

$\text{Past}_i(e_j)$  is a totally ordered set by relation  $\rightarrow_i$

$\max(\text{Past}_i(e_j)) \rightarrow$  last event on  $p_i$  to affect  $e_j$

$$\max \text{past}(e_j) = \bigcup_{p_i} (\max(\text{Past}_i(e_j))) \rightarrow \text{set of last events on every process to affect } e_j$$

### Future Cones

Future of event  $e_j$  is set of all events  $e_i$  that are causally affected by  $e_j$ , i.e.  $e_j \rightarrow e_i$

$$\text{Future}(e_j) = \{e_i \mid \forall e_i \in H, e_j \rightarrow e_i\}$$

All events after  $\max \text{past}(e_j)$  & before  $\min \text{future}(e_j)$  are concurrent with  $e_j$  i.e., H-Past-fut.

$\min(\text{Future}_i(e_j)) \rightarrow$  first event on  $p_i$  to be affected by  $e_j$

$$\min(\text{Future}(e_j)) = \bigcup_{p_i} (\min(\text{Future}_i(e_j))) \rightarrow \text{set of all first events on processes to be affected by } e_j$$

## Models of Process Communication

- ↳ **Synchronous** → blocking type; sender sends msg & blocks until receiver rcs msg  
→ sender resumes execution only after it learns abt rcv
- ↳ **Asynchronous** → non-blocking type; sender & rcvr dont synchronize, ie sender doesn't wait after sending msg  
→ msg is buffered by system until rcvr is ready to rcv.

**Asynch** → needs more complex buffer mgmt but provides higher degree of parallelism. Difficult to design & implement. Better performance

**Message in Transit**:  $\text{transit}(\text{LS}_i, \text{LS}_j)$  [msg in transit b/w  $P_i$  &  $P_j$  on channel  $SC$ ]  
 $= \{m_{ij} \mid \text{send}(m_{ij}) \in \text{LS}_i \wedge \text{rec}(m_{ij}) \notin \text{LS}_j\}$

FIFO model → each channel acts as FIFO message queue, ie, order preserved

NON FIFO model → channel acts as set in which sender adds msgs & rcvr removes msgs from it randomly

For causality,

If  $\text{send}(m_{ij}) \rightarrow \text{send}(m_{kj})$  then  
 $\text{rcv}(m_{ij}) \rightarrow \text{rcv}(m_{kj})$

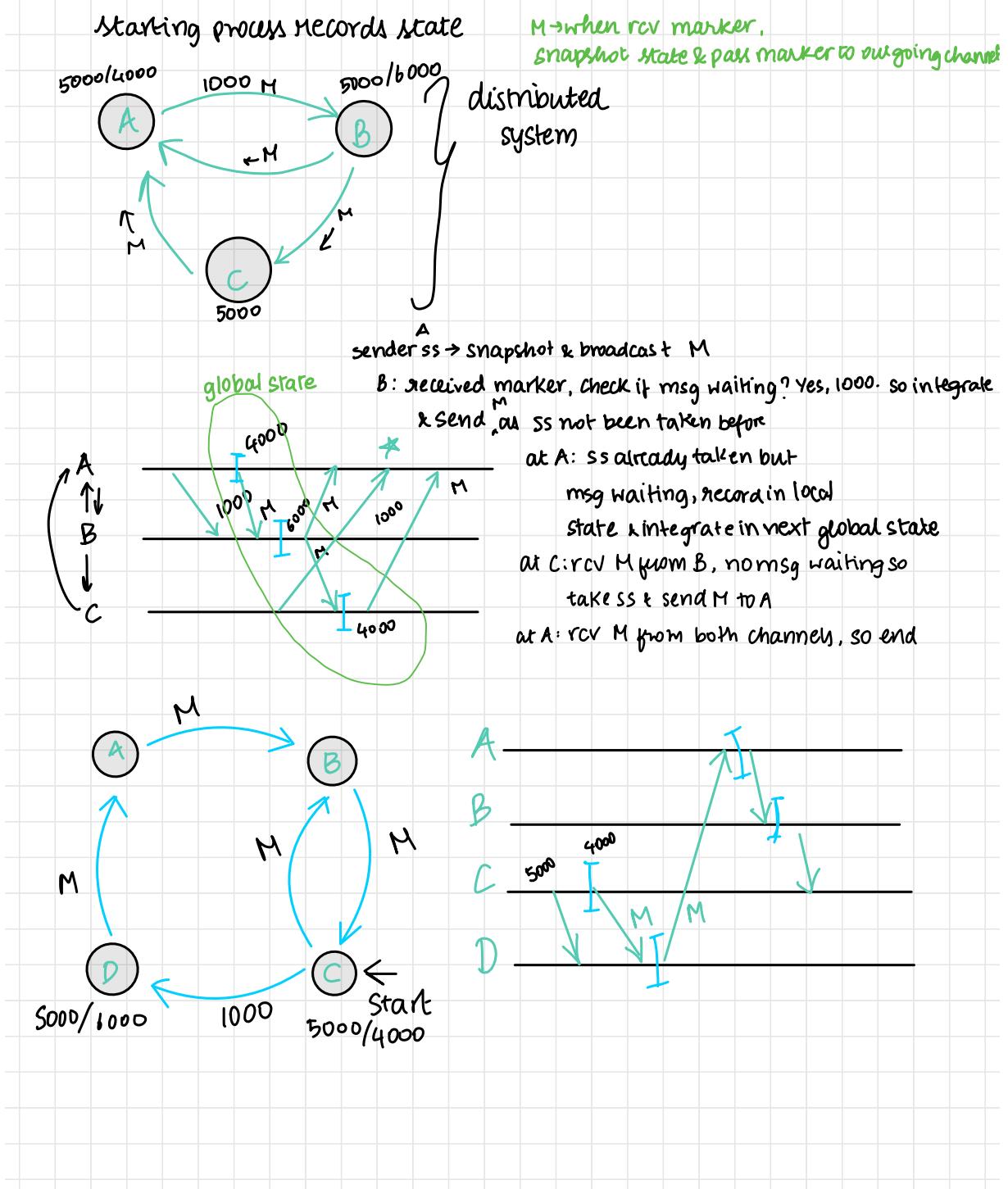
## Chandy Lamport Algorithm

- ↳ uses control msg, "marker" whose role in a FIFO system is to separate msgs in channel
- ↳ after a node takes SS, send marker to all outgoing channels before next msg
- ↳ Marker separates msgs in channel into those to be and not included in SS
- ↳ 2 rules

<b>Marker Sending Rule for process <math>i</math></b>
① Process $i$ records its state.
② For each outgoing channel $C$ on which a marker has not been sent, $i$ sends a marker along $C$ before $i$ sends further messages along $C$ .
<b>Marker Receiving Rule for process <math>j</math></b>
On receiving a marker along channel $C$ :
if $j$ has not recorded its state <b>then</b>
Record the state of $C$ as the empty set
Follow the "Marker Sending Rule"
<b>else</b>
Record the state of $C$ as the set of messages received along $C$ after $j$ 's state was recorded and before $j$ received the marker along $C$

- ↳ Algorithm terminates after each process received a marker on all of its incoming channels

## global Snapshot Recording algorithm /Chandy Lamport



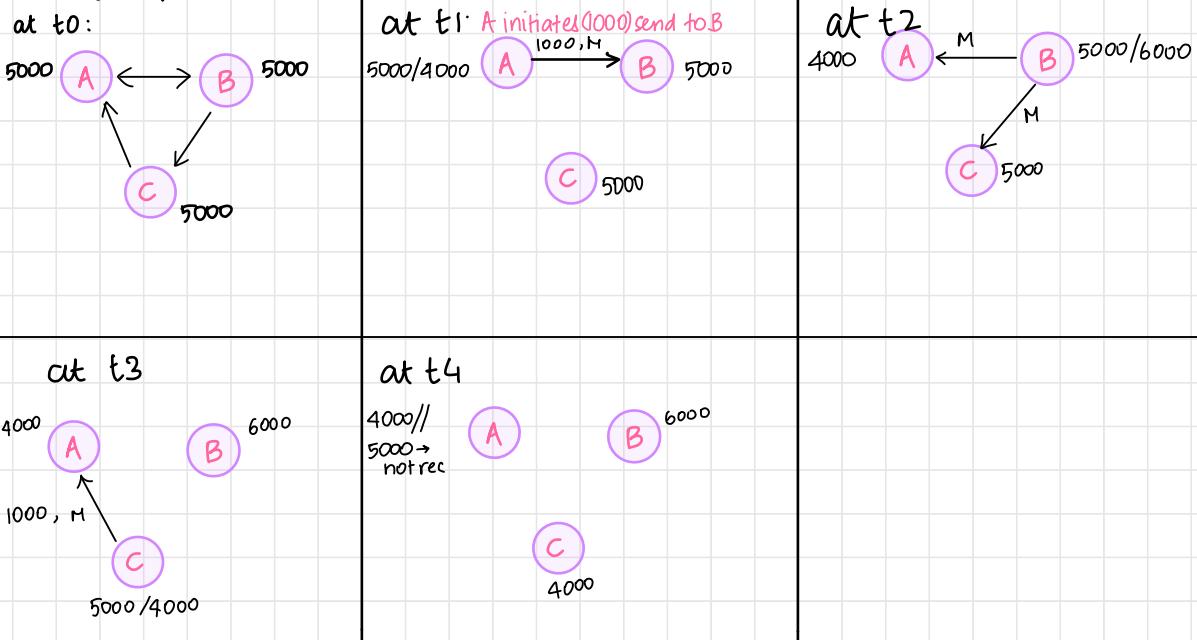
Any node can start Algorithm.

- On send:
  - if msg on channel, integrate
  - (initiator) send marker M through all outgoing channels

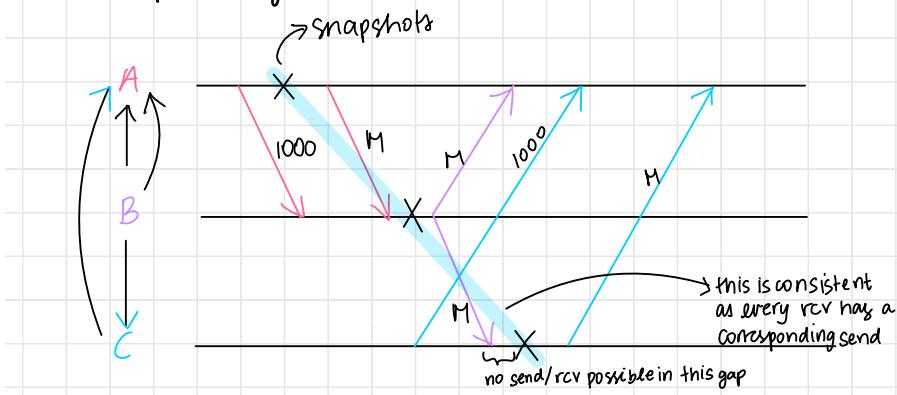
- On receive:
  - record states
  - follow send rule

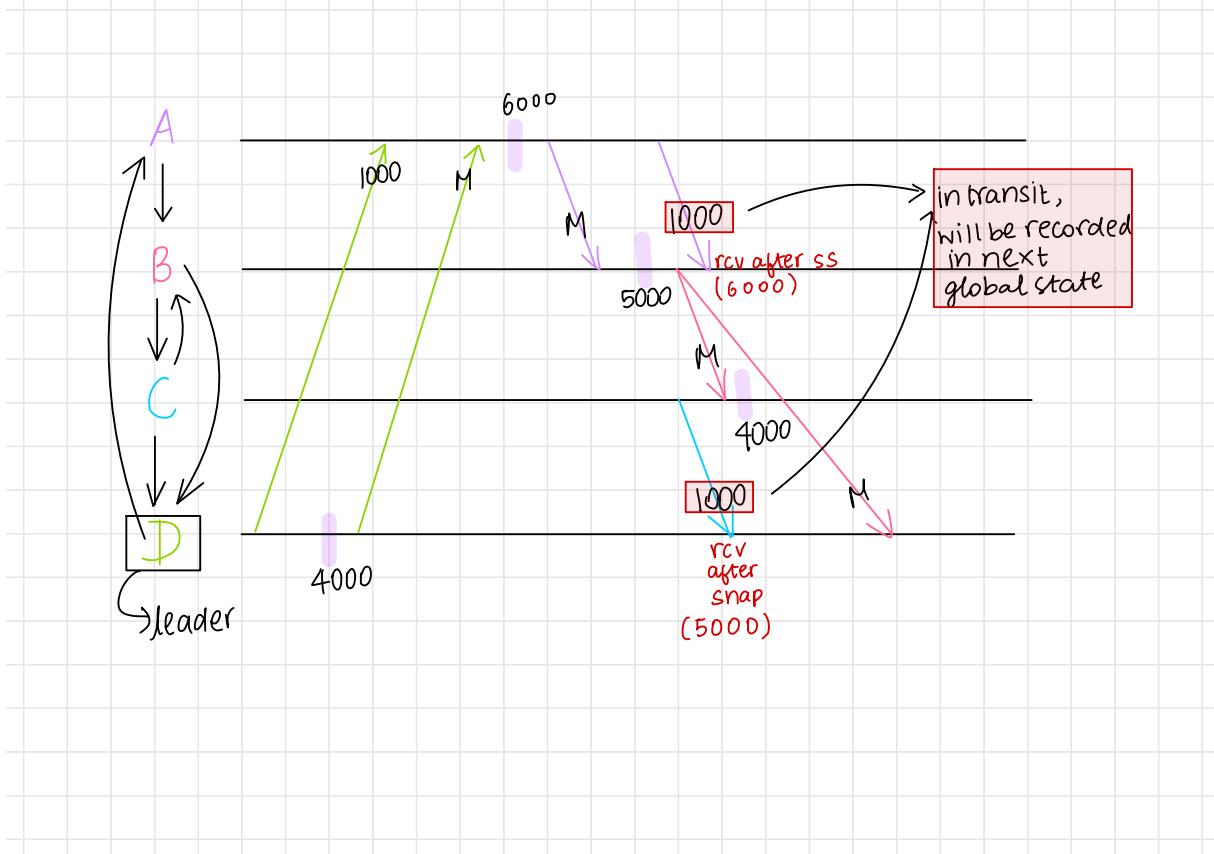
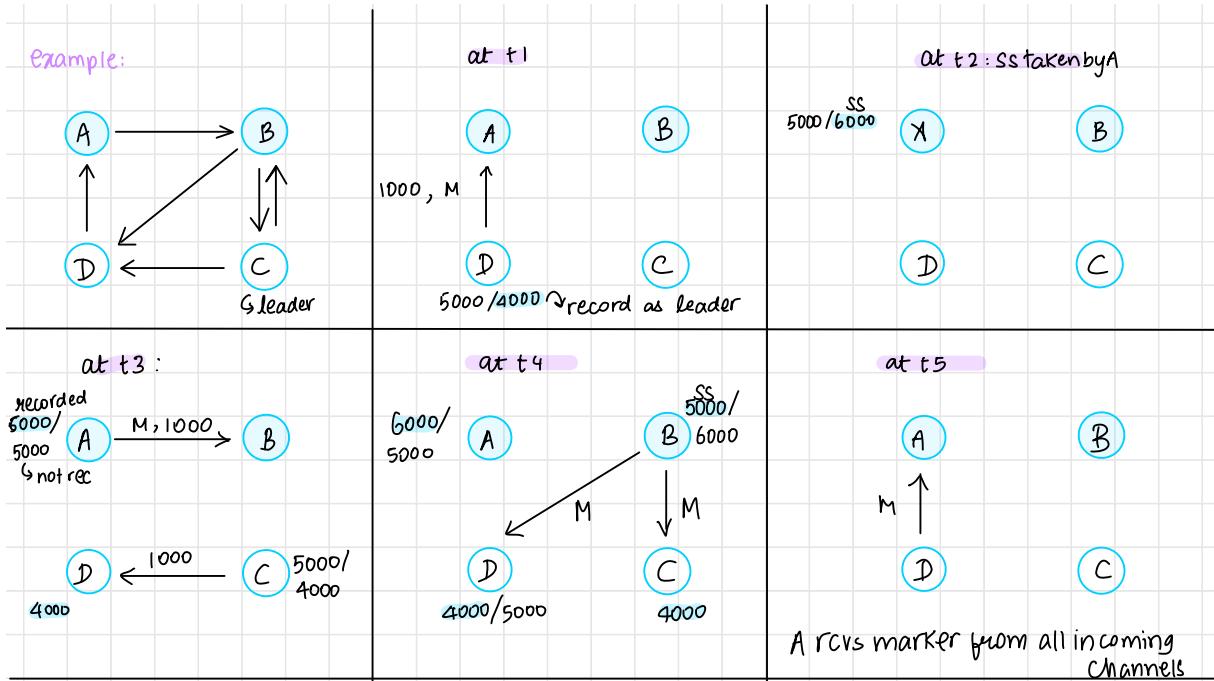
For ending algorithm: if initiator receives marker from all incoming channels

Step-by-step example

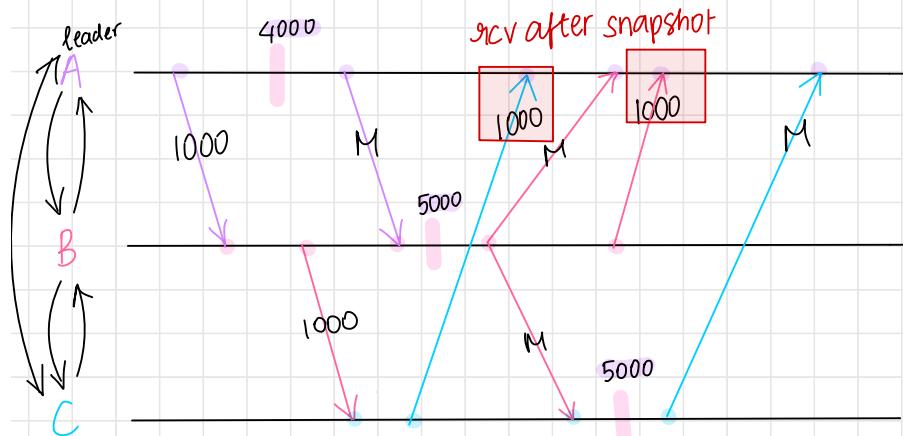
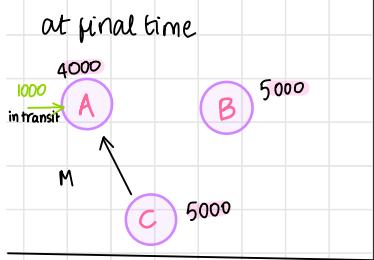
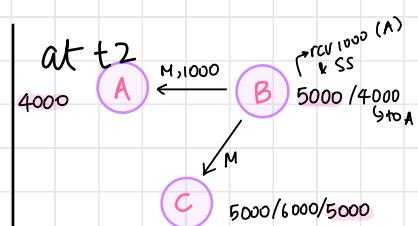
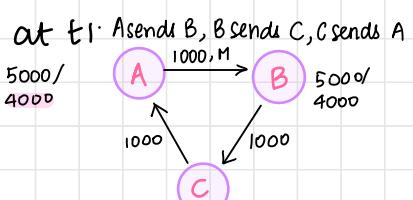
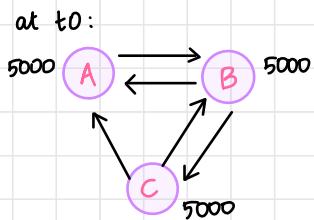


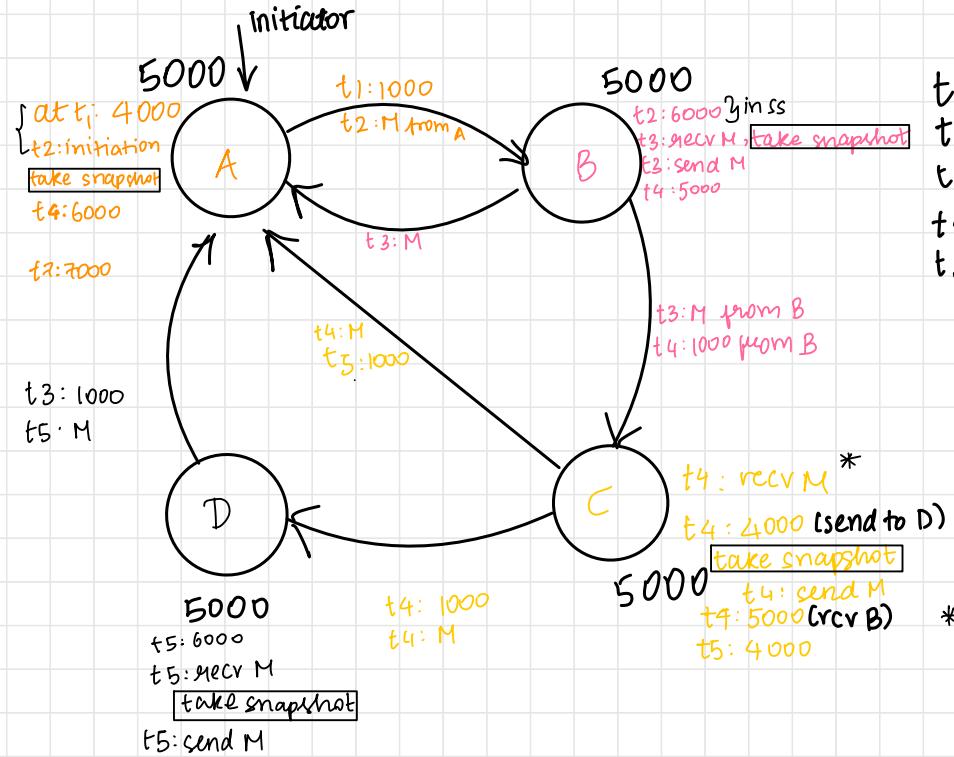
time-space diagram:





### Example

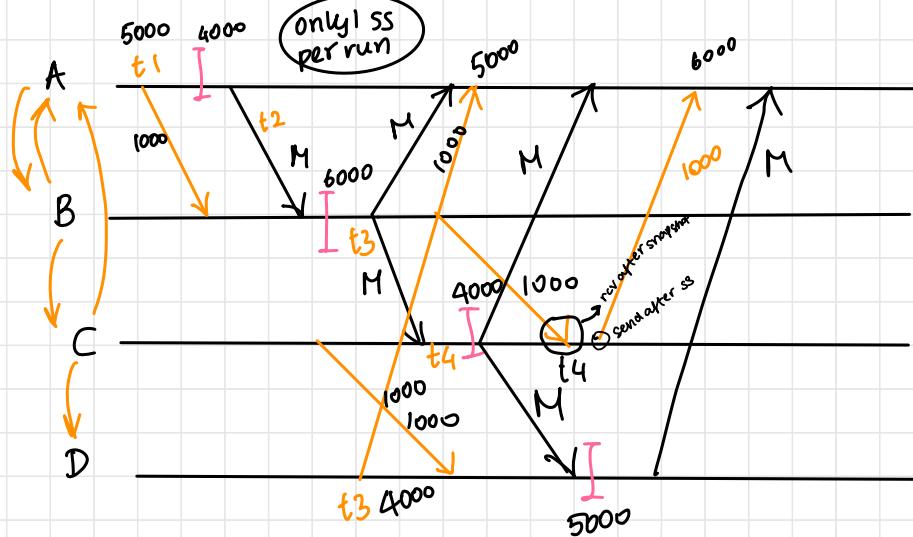




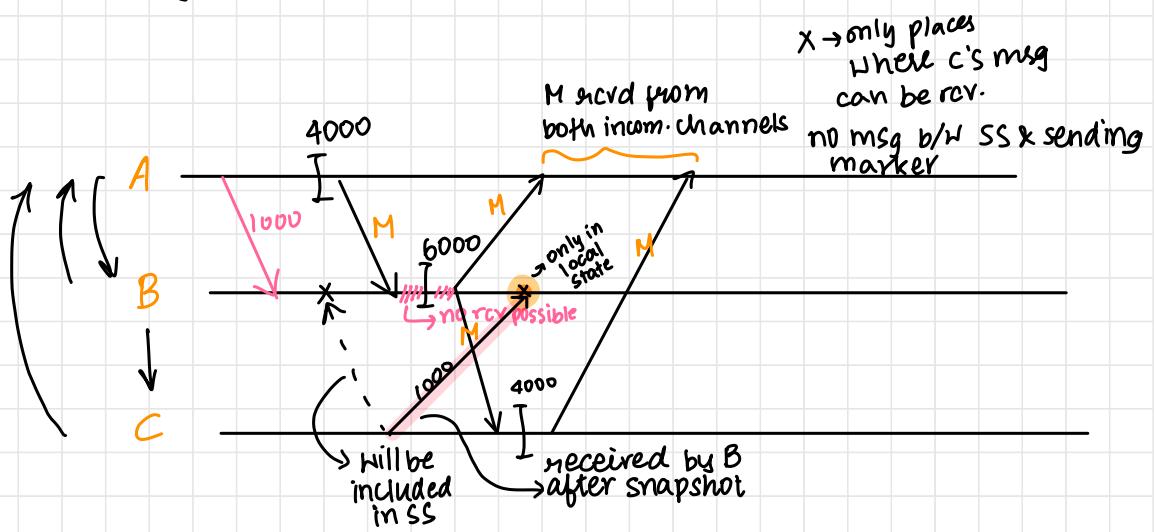
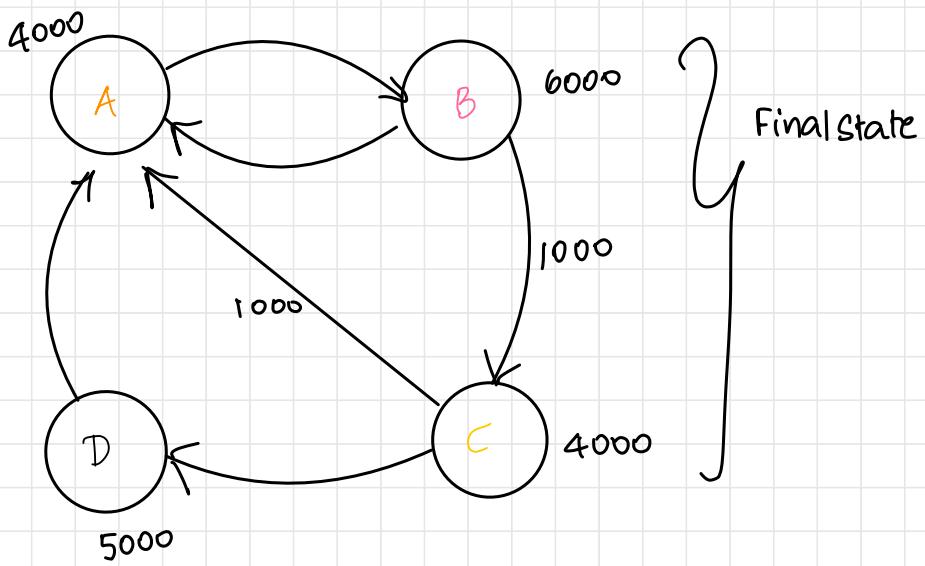
- t<sub>1</sub>: A → B
- t<sub>2</sub>: initiation
- t<sub>3</sub>: D → A
- t<sub>4</sub>: B → C, C → D
- t<sub>5</sub>: C → A

\* always receive marker before msg from a sender because it leads to inconsistency (RW10S)

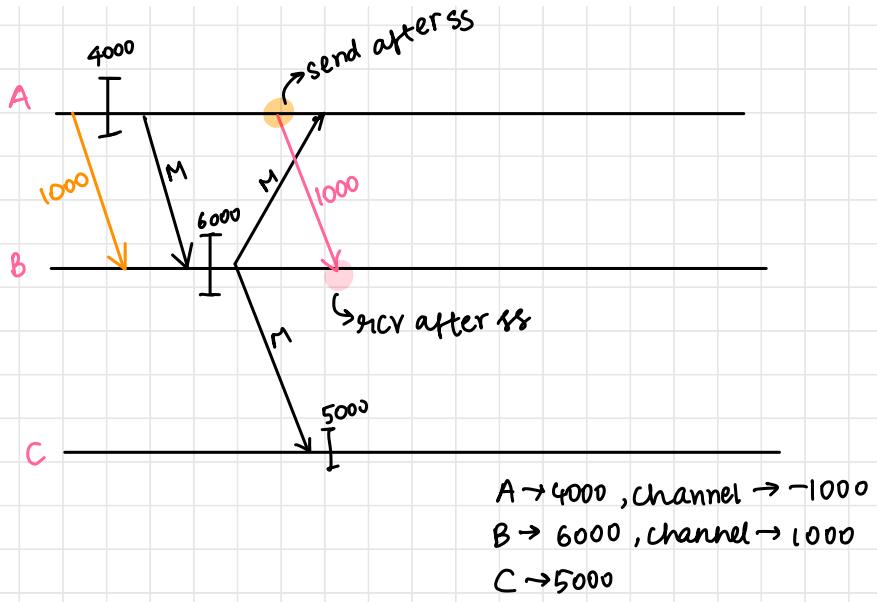
\* can't send msg in  
b/w rec a marker.  
taking ss & sending  
marker



Initiator has to rcv Marker msgs from all incoming channels.



final states :  
 A  $\rightarrow$  4000  
 B  $\rightarrow$  6000 . channel  $\rightarrow$  1000  
 C  $\rightarrow$  4000



By rec Global state : ↳ deadlocks prevented

### Unit - 3

**Mutual Exclusion:** Concurrent access to a shared resource by a process is done in a mutually exclusive manner  
ie, only 1 process is allowed to execute the CS at given time

In a distributed system, shared variables (semaphores)/local kernel can't be used to implement mutex

Message passing is the sole means to implement DMutex

### System Model

System has N sites:  $S_1, S_2, \dots, S_N$

We assume that 1 process ( $P_i$ ) is running at each site ( $S_i$ )

↪ site can be in

↪ idle : executing outside CS

↪ in CS

↪ requesting for CS : site is blocked, no further req for CS allowed

So, even if site has multiple req for CS, it queues them up & serves them 1 at a time

### Requirements for Mutex algorithms

↪ Safety Property : at any instant, only 1 process can execute CS

↪ Liveness Property : absence of deadlocks & starvation, ie, no site should wait endlessly

↪ Fairness Property : every process gets fair chance to execute CS, ie, CS req served in the order they arrive

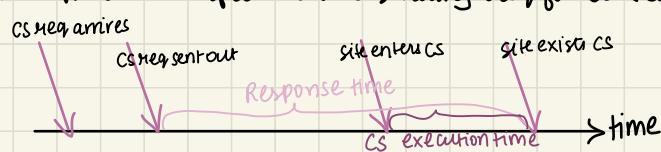
### Performance Metrics

i) Message Complexity : no of msgs required per CS execution by a site

ii) Synchronization delay : time taken b/w 1 site leaving CS & the next site entering CS



iii) Response Time : time taken from a site sending req for CS & completing its CS



iv) System Throughput: rate at which system executes requests for CS

$$= \frac{1}{\text{Synchronization delay} + \text{execution delay}} = \frac{1}{SD + E}$$

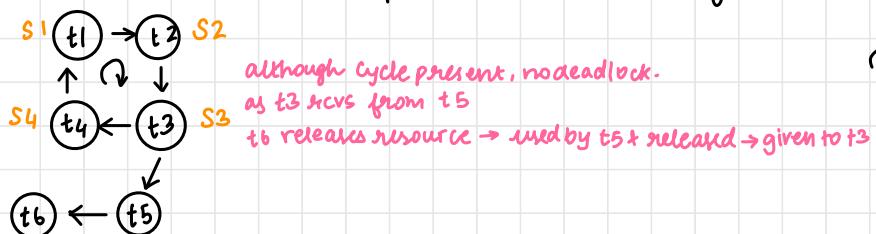
↳ Loads  
↳ low load → rarely >1 concurrent req for CS

↳ high load → always a pending req for CS at a site

## Knapp's Classification

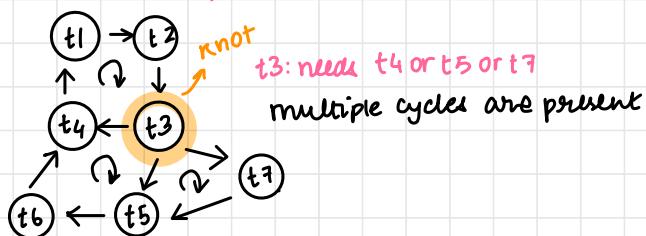
- i) Path Pushing
- ii) Edge Chasing (AND deadlocks)
- iii) Diffusion Computation (OR deadlocks)
- iv) global state based - GisRA algorithm

**OR based deadlocks** - can request for many resources  
but can proceed when it recvs one of it



min. starvation occurs

What is a knot?



If a node is a knot then

↳ all outgoing edges must lead to a cycle  
ie, no loose edges

↳ it is a subgraph within same graph - all edges lead back to same node  
Not all cycles have knots but all knots lead to cycles

**AND request model** cycle for Deadlocks

**OR request model** knot for deadlocks

## Lamport's Algorithm

- ↳ Requests for CS are executed in increasing order of timestamps
- ↳ Every site has a request queue to hold mutex req ordered by timestamps
- ↳ Communication channels deliver msgs in FIFO order

## 3 Phases of Algorithm

### ↳ Request Phase:

- When site  $S_i$  wants to enter CS, broadcast Request (seqno, process id) & places it on its request queue  $(ts_i, i)$
- When site  $S_j$  receives request  $(ts_i, i)$ , it places it on its req.queue & returns reply  $(ts_i, i)$  → Reply Phase

### ↳ Executing Critical Section

Site  $S_i$  enters CS when the following conditions hold

- $S_i$ 's request is at the top of the queue → L2
- $S_i$  acrd reply messages from all other sites → L1

### ↳ Release Phase

- After  $S_i$  finishes execution, it deletes request  $(ts_i, i)$  from queue & broadcasts a release msg to all other sites
- When  $S_j$  recvs release  $(ts_i, i)$  from  $S_i$ ,  $S_j$  removes req.  $(ts_i, i)$  from its queue

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.queue*s and condition L1 holds at both 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<sub>j</sub>*. This implies that  $S_j$ 's own request is at the top when  $S_j$  was executing its CS. This implies that  $S_j$ 's own request is at the top when  $S_j$  was executing its CS. This contradicts the FIFO property of the communication channels.

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<sub>i</sub>*. This is a contradiction!

LamPort's requires  $3(N-1)$  msgs per CS invocation

- 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  $S_j$ '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.

## Distributed Mutual Exclusion

In DS, states are inconsistent

Fairness & liveliness are required in a DS  
 ↳ To solve ->

Token based algo (e.g. Ring Topology)

Non-token based algo

i) Token based: arrange nodes cyclic order. Pass token once used to next node

ii) NonToken based: request is broadcasted

↳ Node which possesses resource should id who needs resource next

↳ same resource shouldn't be given

## Lamport's Algorithm

3 Phases → i) Request - node wants access to critical section. Should know which req is being served  
 needs process id & seq no (current req+1). Place on self queue, queue manager broadcasts

↑ seq. no

ii) Reply - everyone replies back to req in order of recv. But if some node objects, replies immediately

iii) Release - release critical section & broadcast to all nodes

	$(D, 1)$ at release	$(D, 1)$ at release from P1	$(D, 1)$ at release from P1
time	P1	P2	P3
1:	Req(1, 1) ↑ PID		
2:		Rcv Req(1, 1) has no objection	Rcv Req(1, 1)
3:		Reply Req(1, 1)	Reply Req(1, 1)
4:	Reply rcvd (1, 1) from P2		
5:	(1, 1) from P3		
6:	execute critical section		
7:	Release Req(1, 1) delete req(1, 1)		
8:		Rcv rel(1, 1)	Rcv rel(1, 1)
9:		delete req(1, 1)	del req(1, 1)

→ Phase 1 Request

→ Phase 2 Reply

→ Phase 3 Release

## Case 2

P1	P2	P3
( <del>1,1</del> ) (2,2)	(1,1) (2,2)	(1,1) (2,2)
req (1,1)	mcv req (1,1) req (2,2) <i>Concurrently</i>	mcv req (1,1) req req (1,1)
mcv req (2,2) <i>No reply</i>	rep req (1,1) <i>Hairness as P1</i>	mcv req (2,2)
rcv rep (1,1) from P3	mcv rep (2,2) from P3	rep req (2,2)
Exec. Critical Section		
del req (1,1) & release		
rep req (2,2)	rcv rel. req (1,1)	mcv rel. req (1,1)
	del req (1,1)	del req (1,1)
	rcv rep (2,2) from P1	
Exec. Critical Section		
del req (2,2) & release		
mcv rel. req (2,2)		mcv rel. req (2,2)
del req (2,2)		del req (2,2)

Case 3

P1	P2	P3
$\text{req}(1,1)$ $\text{rcv req}(1,2)$ Wont reply as P1 req at top of its queue	$\text{req}(1,2)$ $\text{rcv req}(1,1)$ no reply as $\text{req}(1,2)$ at front of queue forced to shuffle	$\text{req}(1,1)$ $\text{rcv req}(1,2)$
$\text{rcv resp}(1,1)$ from P3	$\text{rcv resp}(1,2)$ from P3	$\text{rep resp}(1,1)$ $\text{rep resp}(1,2)$
$\text{rcv resp}(1,1)$ from P2	$\text{rcv resp}(1,2)$ from P3	$\text{rep resp}(1,1)$
<b>Exec. Critical Section</b>	<b>Starvation</b> so order according to Proc ID $P1 > P2 > P3$	$\text{rep resp}(1,2)$
Release $\text{req}(1,1)$	$\text{rcv rel}(1,1)$ $\text{del req}(1,1)$	$\text{rcv rel}(1,1)$ $\text{del req}(1,1)$
$\text{rep resp}(1,2)$	$\text{rcv resp}(1,2)$ from P1 <b>Enter critical section</b> & continue w. release & broadcast	

NOTE: To ensure fairness, only one outstanding req allowed per process

If Comm. problem,  $\text{req}(1,1) \not\rightarrow P3$  *(downread)*

after timeout,  $\text{req}(2,1)$  again by P1

if after re-request,  $\text{req}(1,1)$  reaches P3  $\Rightarrow$  discard if  $\text{rcvd req.no} < \text{prev req.no}$

for 1 cycle:  $3(n-1)$  msgs needed  $\rightarrow$  msg complexity

## Ricart-Agarwala algorithm

- ↳ assumes communication channels are FIFO
- ↳ uses 2 types of messages **Request & Reply** (same as in Lamport)
- ↳ requests are served in order of timestamp
- ↳ each process has a Request-Deferred Array, RD

```
for Process Pi & Pj
    ∀ i, ∀ j RDi[j] = 0 initially
    if Pi defers request from Pj
        RDi[j] = 1
    after Pi replies to req from Pj
        RDi[j] = 0
```

### Phases

#### i) Request

- When site S<sub>i</sub> wants to enter CS, it broadcasts req<sub>i</sub>(t<sub>S<sub>i</sub></sub>, i) to all sites
- If a site S<sub>j</sub> recvs req<sub>i</sub> from S<sub>i</sub>,
  - if it is idle / req CS with t<sub>S<sub>j</sub></sub> > t<sub>S<sub>i</sub></sub> ⇒ reply to S<sub>i</sub>
  - else set RD<sub>j</sub>[i] = 1

#### ii) Execute CS

- Site S<sub>i</sub> enters CS after receiving reply from all other sites

#### iii) Release CS

- When S<sub>i</sub> exits CS, send reply to all deferred msgs
  - ie, if RD<sub>j</sub>[i] = 1, reply to S<sub>j</sub> & set RD<sub>j</sub>[i] = 0

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

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

Requires 2(N-1) msg per CS execution

**Optimization**: if P<sub>i</sub> got Reply from P<sub>j</sub> for prior CS access, P<sub>i</sub> doesn't need to send requests to P<sub>j</sub> for further CS access until P<sub>j</sub> requests CS access  
reduces no of msg to [0 - 2(N-1)]

Ricart-Agarwala's algorithm (D-mutex)

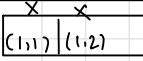
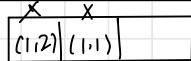
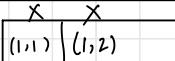
↳ reduces msg complexity to  $2(n-1)$

P1	P2	P3
$\boxed{(1,1) \mid (2,2)}$ X X	$\boxed{(1,1) \mid (2,2)}$ X	$\boxed{(1,1) \mid (2,2)}$ X X
$\text{req. } (1,1)$	$\text{rcv req. } (1,1)$	$\text{rcv req. } (1,1)$
	$\text{req. } (2,2)$	$\text{rep req. } (1,1)$
$\text{rcv req. } (2,2)$	$\text{rep req. } (2,2)$	$\text{rcv req. } (2,2)$
$\text{rcv rep. } (1,1) \text{ from P3}$		$\text{rep req. } (2,2)$
$\text{rcv rep. } (1,1) \text{ from P2}$	$\text{rcv rep. } (2,2) \text{ from P3}$	
Enter critical section <div style="border: 1px solid red; padding: 2px;">del Req.(1,1) &amp; send req.(1,1) to P2 &amp; P3</div>	Lamport's Algorithm	<div style="border: 1px solid black; padding: 2px; margin-left: 20px;">ignore for this Algo</div>
Now, immediately after responses from P2 & P3, delete (1,1) from queue.  after critical section, send response <sup>to P2</sup> & delete (2,2) from queue	delete (1,1) after sending response to P1  This ensures no release msg is needed.	delete (1,1) & (2,2) after responding to P1 & P2

NOTE:

Self requests are deleted after executing critical sections

Other requests are deleted after sending response to it

P1	P2	P3
 $\text{req}(1,1)$ $\text{recv req}(1,2)$ $\text{recv resp}(1,1)$ from P3 $\text{recv resp}(1,1)$ from P2 <b>critical section</b> $\text{delete}(1,1)$ from queue $\text{rep req}(1,2)$ & delete $(1,2)$ from queue	 $\text{req}(1,2)$ $\text{recv req}(1,1)$ $\text{recv resp}(1,2)$ from P3 $\text{process id takes priority}$ $\Rightarrow \text{rep req}(1,1)$ & delete $(1,1)$ from queue $\text{recv resp}(1,2)$ from P1 <b>Enter critical section</b> $\text{delete}(2,2)$ from queue	 $\text{recv req}(1,1)$ $\text{recv req}(1,2)$ $\text{rep req}(1,1)$ & delete $\text{rep req}(1,2)$ & delete

To ensure fairness,

on receiver side  
unlike in Lamport, req deleted immediately after response

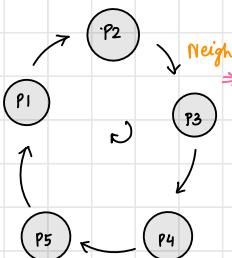
$\Rightarrow$  fairness may be violated?

To ensure fairness, req not deleted from sender until after execution

## Token based D-Mutex (Suzuki-Karumi's algorithm) (SE-B ppt refer)

↳ Arrange nodes in circular order

resources passed around sequentially



Neighbor nodes are predefined

↳ if P1 has token P5 needs,  
it still has to pass through all nodes b4 P5 gets it  
node w/ token doesn't even know who needs token

↳ bad mutex mgmt;  
leads to starvation  
↳ resource free, but not able to be used  
by reqd node

Problem: Starvation; know demands & pass token to that node

Data structures : i) request vector, R at each node . (size → n (n: total no of nodes))

ii) token vector, T (only 1 copy /instance; available at node which last used token)

### Request

Inc  $R_j[i]$  ( $i^{th}$  value of requesting node's R)

broadcast  $R_i$

### Response

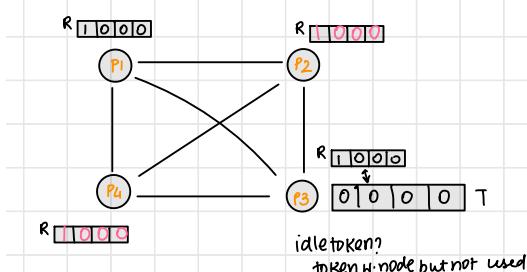
On rcv broadcasted msg.

$R_j[i] = \max(R_j[i], R_i[i])$  (update  $i^{th}$  place of receiver node's R vector)

if node had T, then

if  $T[k] = R_j[i] - 1$ , pass the token

↳ explanation:



i) P1 wants token

$R_1 = 1000$  & broadcast

$\Rightarrow R_2 = R_4 = 1000$  as  $\max(R_1[i], R_2[i])$

↳ why max?

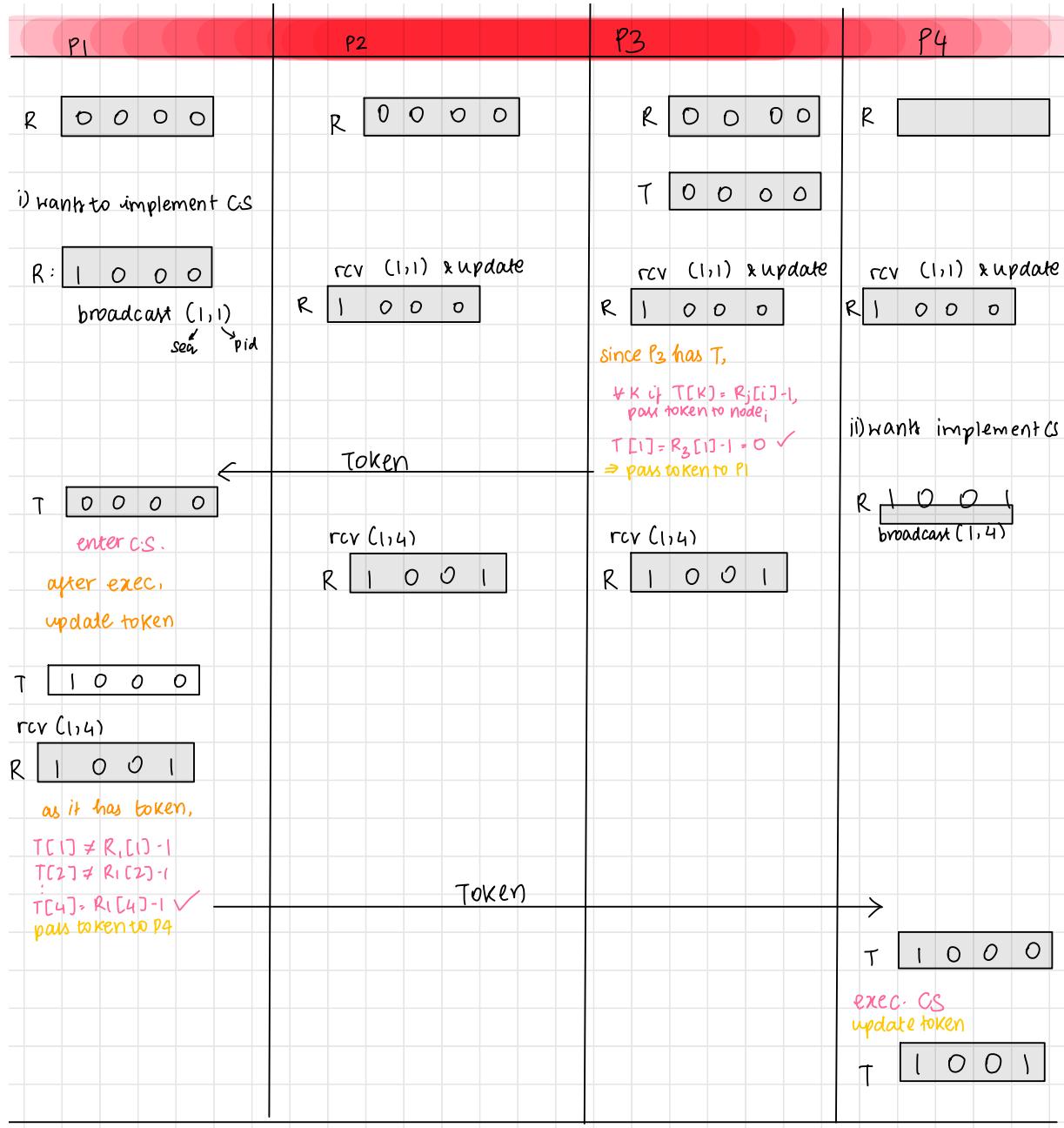
• rcvd req may be outdated as a req  
may have been resent. To ignore old req, take max

for P3, (high T)

$\forall k, T[k] = R_3[1] - 1$

for N1, condition satisfied,  
so pass token to N1

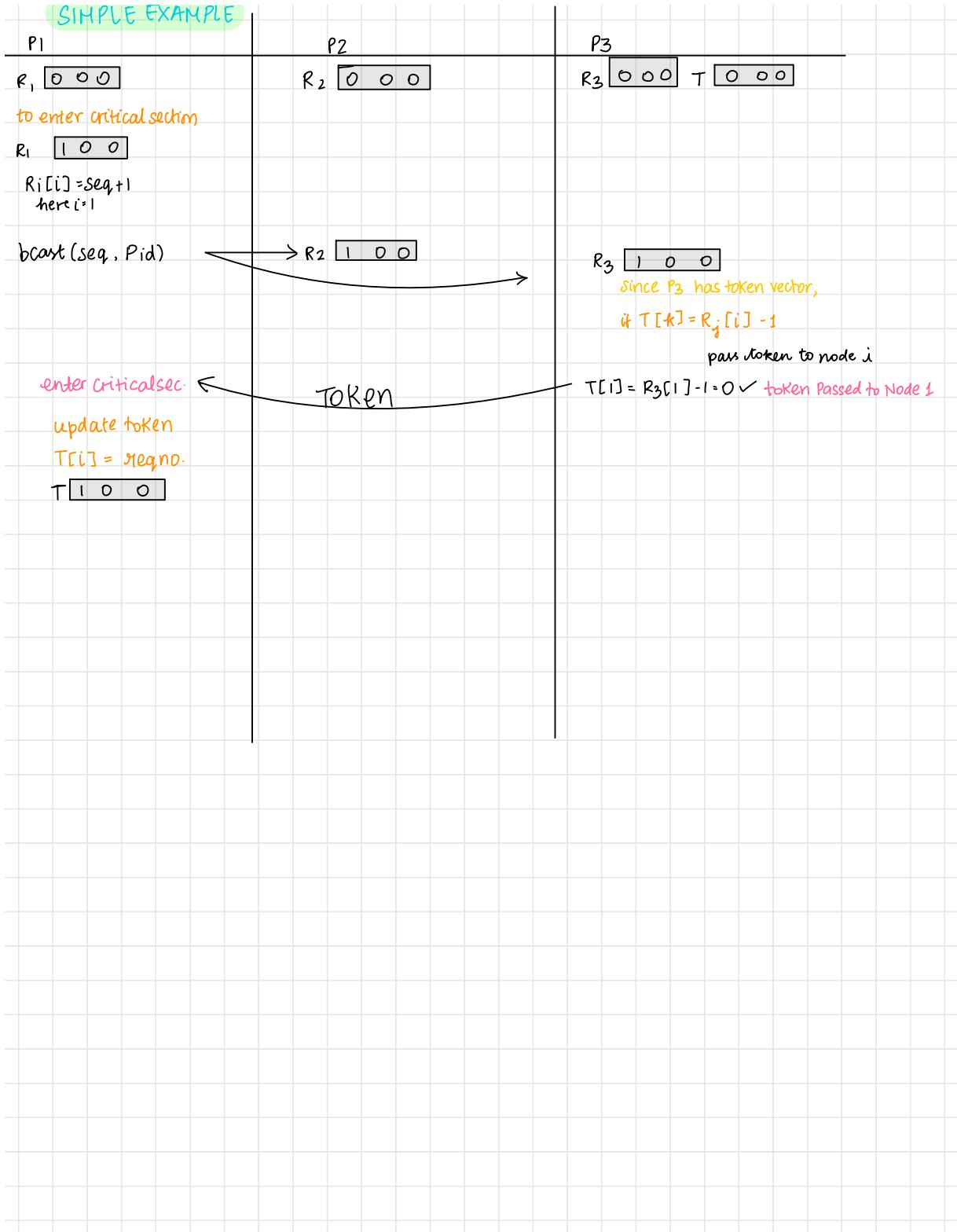
scan token L to R, first value to  
satisfy, pass token to that node.



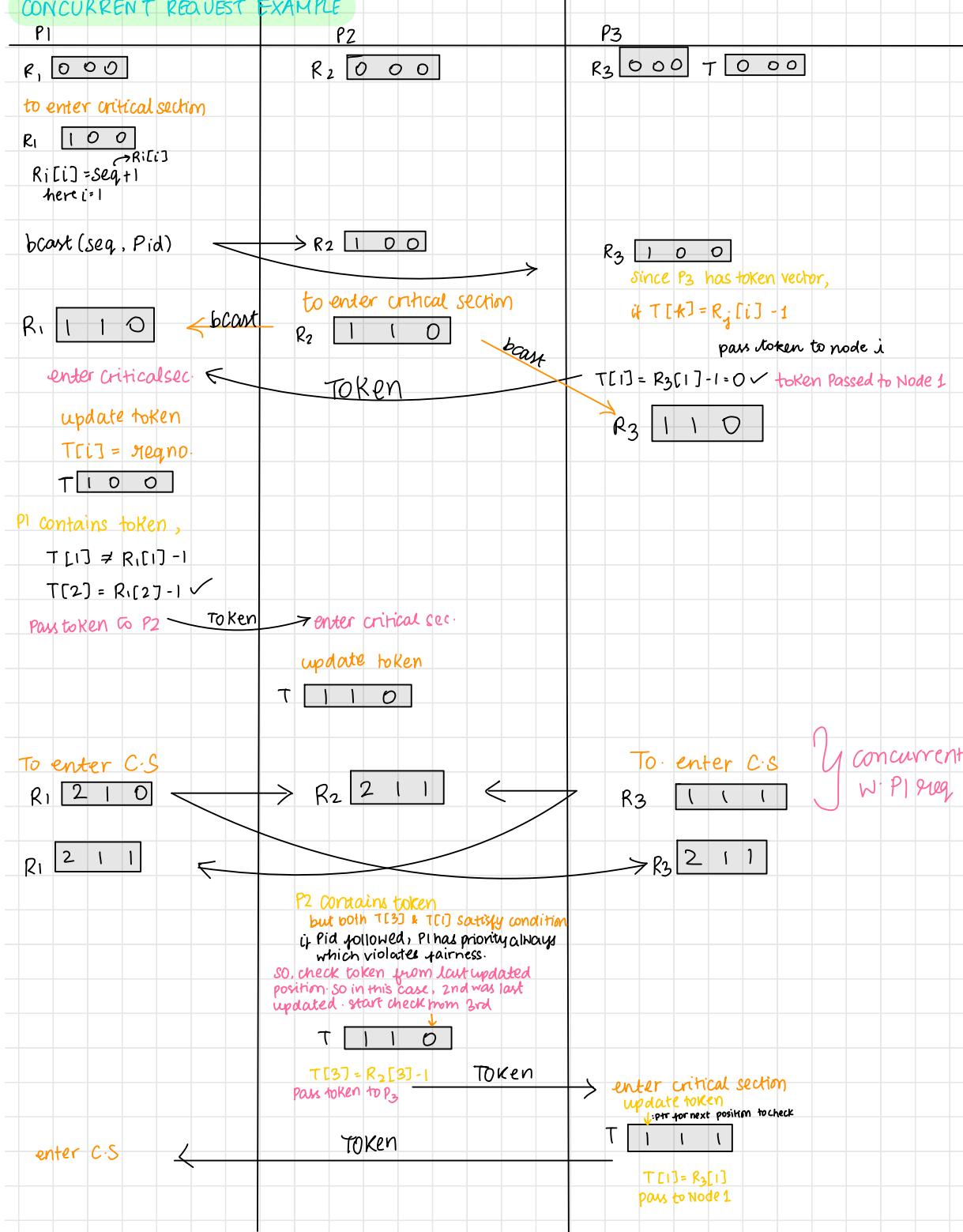
How to ensure fairness (no 1 node requests < 1 times)

↳ maintain copy of local req.

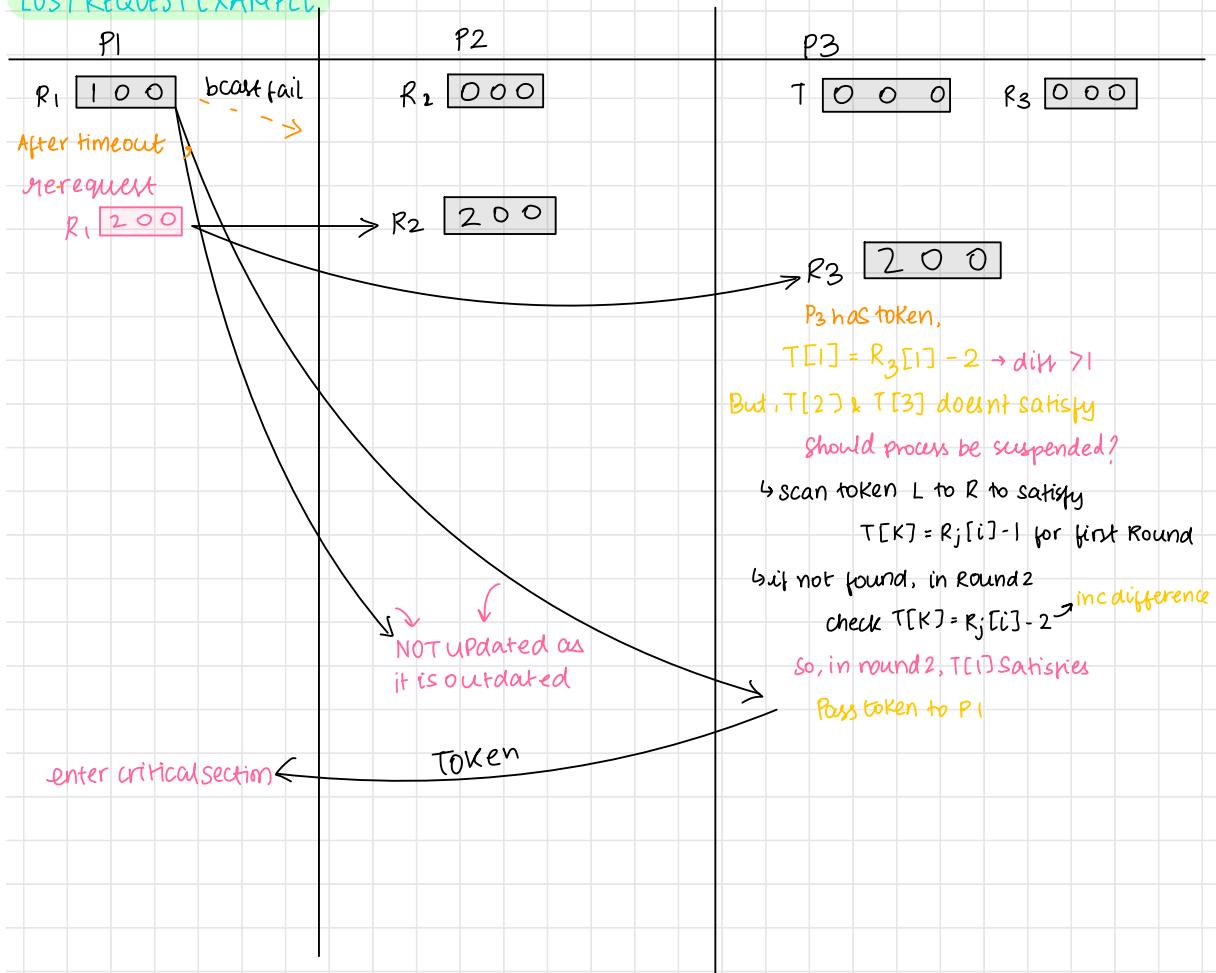
### SIMPLE EXAMPLE



### CONCURRENT REQUEST EXAMPLE

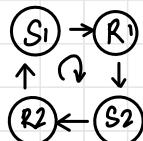


### LOST REQUEST EXAMPLE



## Distributed Deadlocks

Deadlock example:

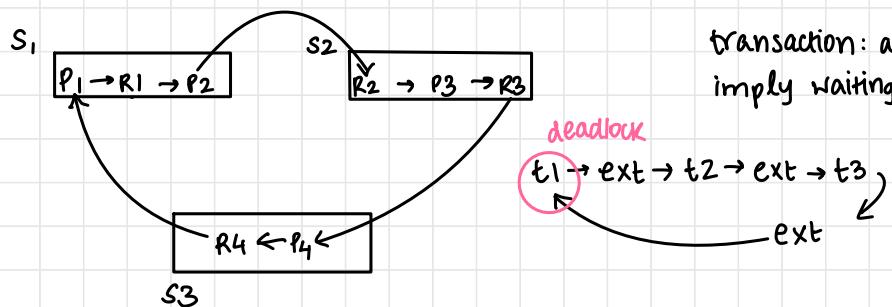


↳ central leader not suitable for DS  
any node can initiate a task

**Phantom Deadlock**: deadlock doesn't exist currently but DS  
mistakenly assumes or vice versa

Path pushing algorithm

Occurs when there is delay in comm. channel



transaction: abstract word to  
imply waiting

## Deadlock Detection in DS

**Deadlock:** a state where a set of processes request resources that are held by other processes in the set

Deadlocks occur if sequence of resource allocations aren't controlled

each on diff processor

If system with async processes  $P_1, P_2 \dots P_n$  communicate, since there is no global clock, processes have instantaneous access

### Assumptions

- ↳ All resources are reusable
- ↳ Processes are allowed only exclusive access to resources
- ↳ Only one copy of each resource

### States of a process

- i) Running  $\rightarrow$  has all reqd resources & is either executing / ready
- ii) Blocked  $\rightarrow$  waiting for resource

### Wait For graph (WFG)

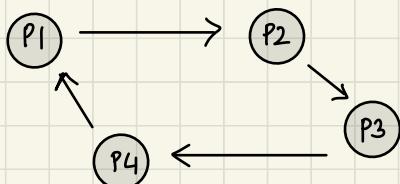
↳ nodes are processes

If there is a directed edge b/w nodes  $P_1$  &  $P_2$

$\Rightarrow P_1$  is blocked & is waiting for a resource that  $P_2$  holds

deadlocks from WFG  $\rightarrow$  if there is a cycle / knot

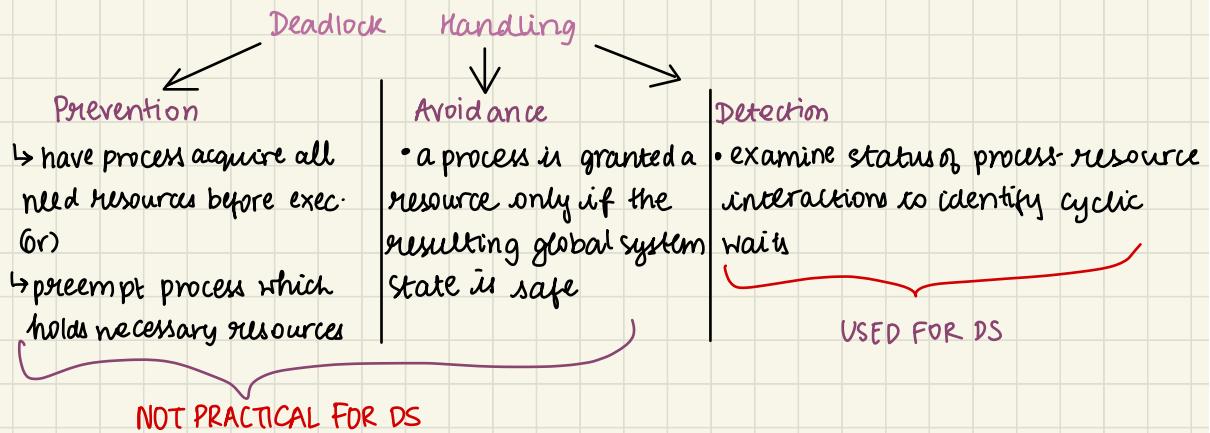
WFG example with deadlock cycle



with knot

## Deadlock Handling

- Complicated in DS because no site has accurate knowledge of current state of system & communication involves finite/unpredictable delay



Deadlock detection → maintain WFO & search for cycles / knots  
↑ AND deadlock → OR deadlock

deadlock detection algorithm must satisfy

↳ Progress (No undetected deadlocks)

- Algorithm must detect all existing deadlocks in finite time  
ie, after all wait-for dependencies are formed, algo shouldn't wait for events

↳ Safety (no false deadlocks)

- Shouldn't report Phantom deadlocks

Deadlock Resolution → breaking wait-for dependencies to solve deadlocks  
→ involves rollback of deadlocked processes & reassigning resources to blocked processes

## Different Models

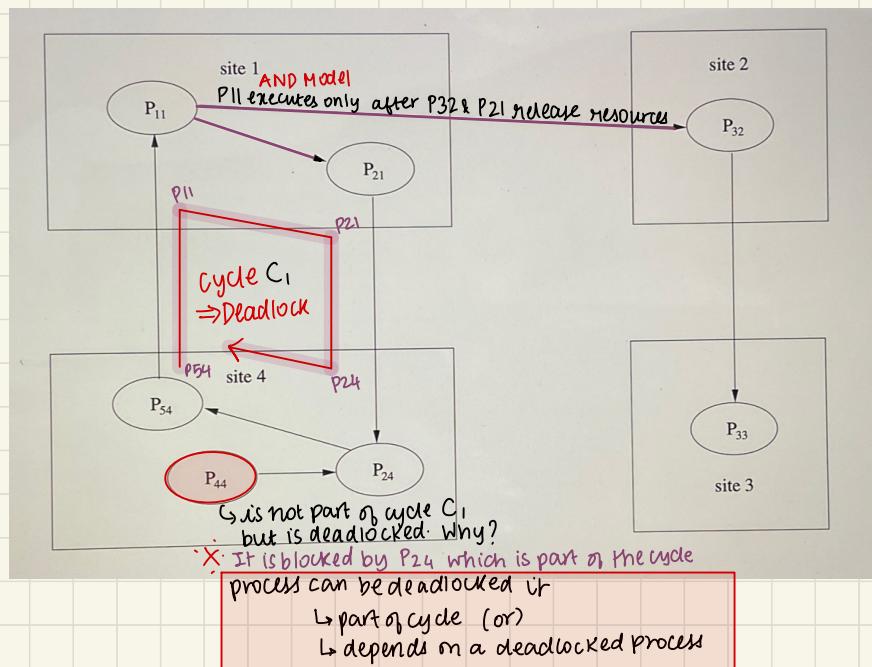
### ↳ Single Resource Model

- one process can have at most 1 outstanding req for a resource, ie,  
outdegree of WFG for a node = 1
- presence of cycle  $\Rightarrow$  deadlock

### ↳ AND model

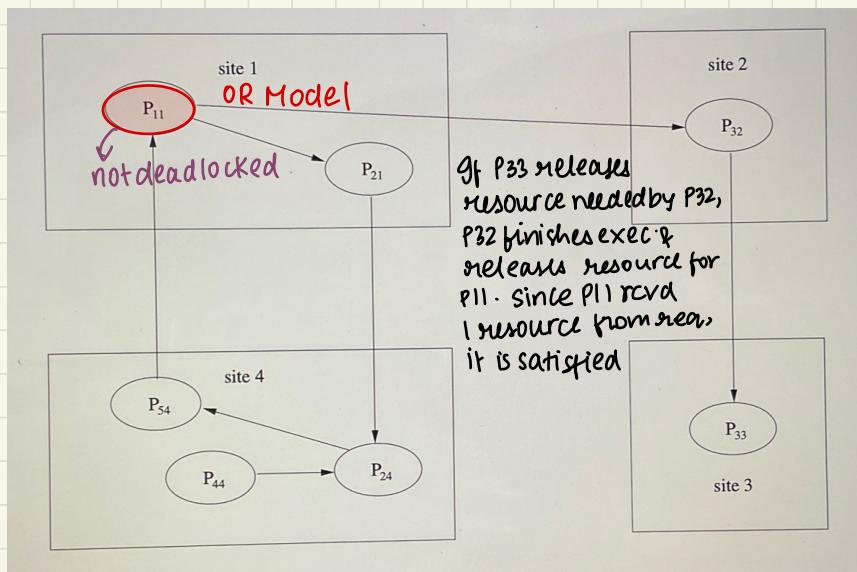
- process can have  $>1$  resource req simultaneously & satisfied only after all resources rcvd, ie, execution happens only after node rcvs all resources  
outdegree of WFG for a node  $\geq 1$
- presence of cycle  $\Rightarrow$  deadlock

NOTE: AND model is more general than single resource model



### ↳ OR Model

- process can have  $>1$  outstanding req for resource simultaneously, but begins execution if even 1 of the resources is granted  
req satisfied
- presence of cycle  $\not\Rightarrow$  deadlock
- presence of knot  $\Rightarrow$  deadlock



### ↳ AND-OR Model

- generalisation of both AND & OR models
- process specifies kind of request as (and) / (or)
  - eg: req can be  $x$  and ( $y$  or  $z$ )
- can't use WFGs.
- have to use repeated application of the test for OR-model deadlock

### ↳ $\binom{P}{q}$ model

- $P$  out of  $q$  model allows process to obtain any  $k$  ( $p$ ) resources out of pool of  $n$  ( $q$ ) resources
- same expressive power as AND-OR but more compact request  
Every  $\binom{P}{q}$  request can be expressed as AND-OR & vice versa

$$\text{• AND req} \equiv \binom{P}{p}$$

$$\text{• OR req} \equiv \binom{P}{1}$$

### ↳ Unrestricted model

- no assumptions about requests other than deadlock is stable  
(livelock, deadlock)
- concerns ab. properties of problem separated from DS computations  
(msg passing vs. sync comm)

## Distributed deadlock detection algorithms

### ↳ Path Pushing

- deadlocks detected using explicit global WFGs
- each site has local WFGs, if it performs deadlock computation, send WFGs to neighbors
- repeated until some site has sufficient info abt global state to announce presence / absence of deadlocks

### Drawbacks

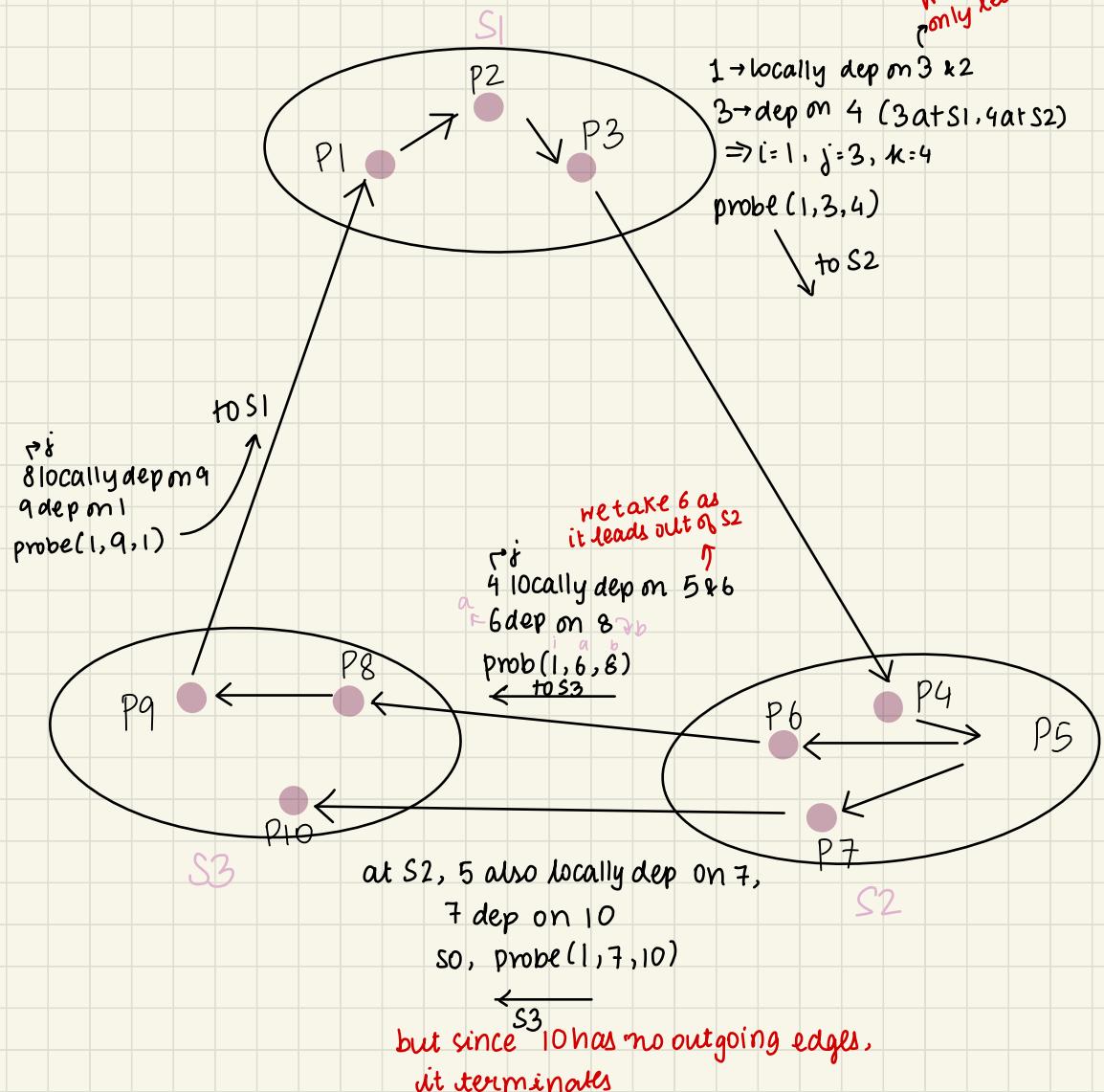
- It detects phantom deadlocks
- algo sends  $n(n-1)/2$  msgs to detect deadlock in  $n$  sites
- size of msg :  $O(N)$
- delay in detecting deadlock :  $O(n)$ , so by the time its detected, it would be resolved

### ↳ Edge Chasing

- presence of cycle detected by propagating 'probes' along edges of graph
- cycle detected when a node receives a probe identical to one it sent previously
- only blocked processes forward Probe messages through outgoing edges
- Probes are fixed in size & short

$\rightarrow P_k$  is locally dependent on  $P_a$   
 $\rightarrow P_a$  is waiting for  $P_b$   
 $\rightarrow P_a, P_b$  are on different sites  
 send probe  $(i, a, b)$  to home site of  $P_b$

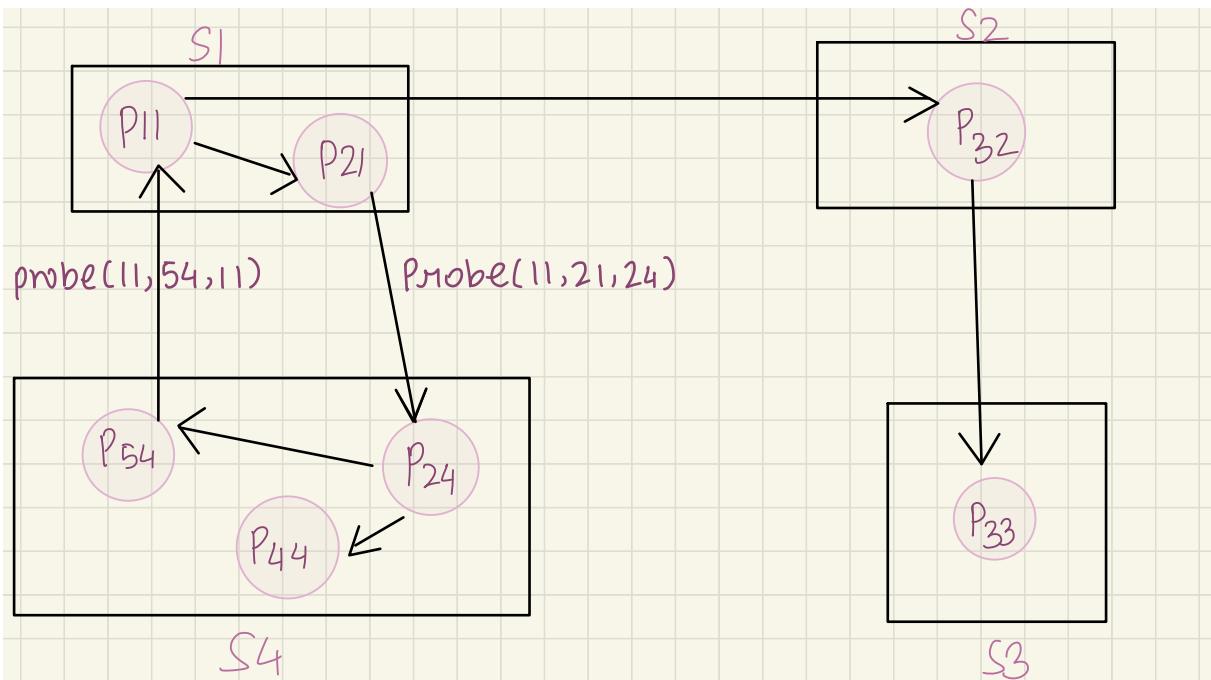
we take 3 as it  
 only leads out of  $S_1$



For  $m$  process at  $n$  sites, it exchanges  $m(n-1)/2$  msgs.

Msg size  $\rightarrow$  3 int values

delay =  $O(n)$



①

at Site 1

- ↳ P<sub>11</sub> locally dep on P<sub>21</sub>
- ↳ P<sub>21</sub> dep on P<sub>24</sub> at S4
- probe (11, 21, 24) to S4  
i j k

②

at Site 4

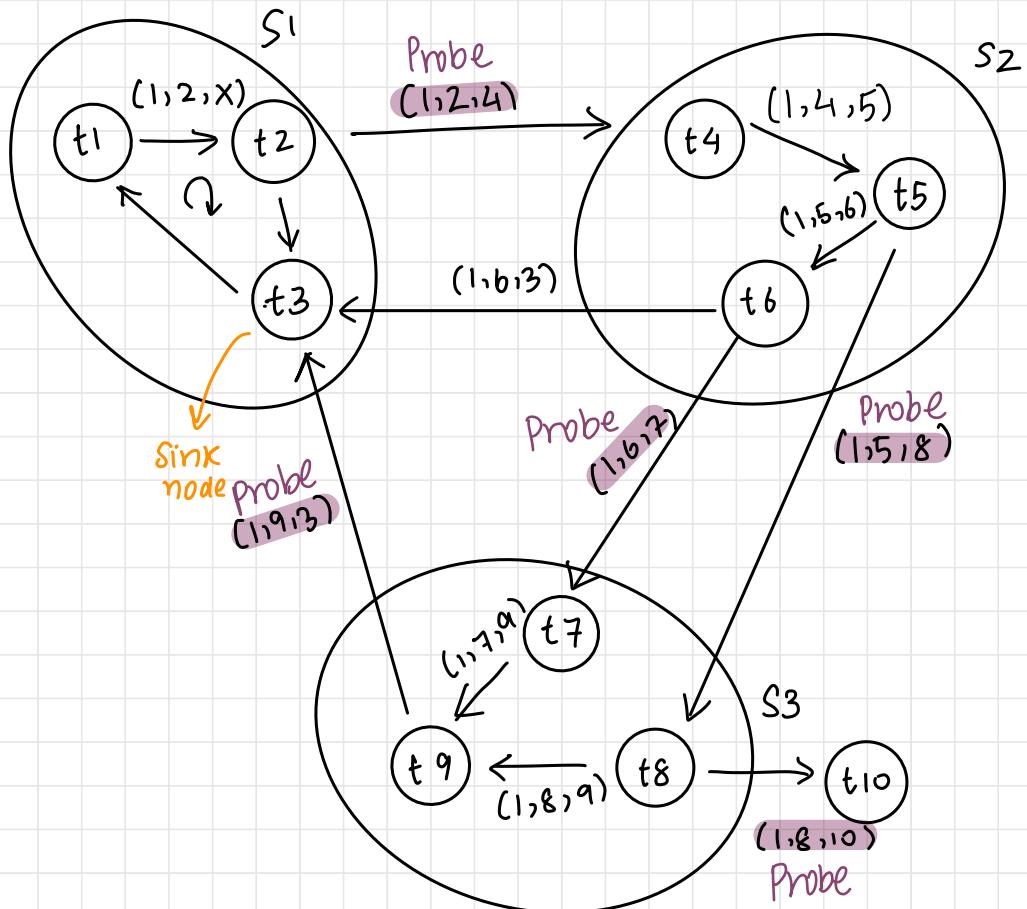
- ↳ P<sub>24</sub> locally dep on P<sub>44</sub> & P<sub>54</sub>
- Only P<sub>54</sub> leads out of S4,
- ↳ P<sub>54</sub> dep on P<sub>11</sub>
- probe (11, 54, 11) to S1  
i a b

### ↳ Diffusion Computation based

- deadlock detection is diffused through WFCN of the system
- make use of echo algorithm
- This computation superimposed over DS computation. If termination, initiator declares a deadlock
- To detect a deadlock, a process sends out a query through all outgoing edges in the WFCN
- Queries are propagated through edges (specific)
- When a blocked process recvs first - query for a deadlock detection initiation, it only replies after all its own queries have received replies
- For subsequent queries from that same deadlock det. initiation, it replies immediately.
- Initiator of deadlock detection detects a deadlock when it recvs a reply for every query it sent out

initiator sends a query (if there is an outgoing edge)

*Get expected response*



## Chandy Misra Haas algorithm for AND model

↳ Based on edge chasing

↳ uses a probe  $(i, j, k) \rightarrow$  deadlock detection initiated by  $P_i$   $\times$  is being sent by  $P_j$  at home site to  $P_k$  at another site

↳ if  $P_i$  receives a probe back, there is a deadlock.

**Dependency:** a process  $P_j$  is dependent on a process  $P_k$  if

there is a seq,  $P_j, P_{i1}, P_{i2}, \dots, P_{im}, P_k$  such that

$\rightarrow P_{i1} \dots P_{im}$  are blocked &  $P_i$  is waiting for resource from  $P_{i1}$

&  $P_{im}$  waiting for  $P_k$  &  $P_j$  is waiting for  $P_i$

i.e., except  $P_k$  all processes are blocked & each process except  $P_j$  holds a resource for which the prev. process in seq. is waiting

**Local dependency:**  $P_j$  is locally dependent on  $P_k$  if  $P_j$  depends on  $P_k$  & both  $P_j$  &  $P_k$  are in same site

## Data Structures

↳ Each process  $P_i$  has an array  $\text{dependent}_i$

$\text{dependent}_i(j) = \text{true}$  if  $P_i$  knows  $P_j$  depends on  $P_i$  given in ppt but wrong initially,  $\text{dependent}_i(j) = \text{false}$   $\forall i \& \forall j$

$\text{dependent}_i(j) \Rightarrow$  all  $P_j$  that holds a resource that  $P_i$  wants

Algorithm exactly same as Edge Chasing

## Chandy-Misra-Haas Algorithm for OR model

↳ Based on diffusion computation

2 types of messages:

- ↳  $\text{query}(i, j, k)$  ↳ diffusion comp initiated by  $P_i$ ,
- ↳  $\text{reply}(i, j, k)$  sent from  $P_j$  to  $P_k$

↳ Initiated by a blocked process → sends query to all processes in its dependent set,  
(e.g., all its outgoing edges)?

↳ If active node (node w. no outgoing edges), discard query

↳ When blocked process  $P_k$  recvs  $\text{query}(i, j, k)$

    If this is the first query msg rcvd by  $P_k$  initiated by  $P_i$  → engaging query

- $P_k$  propagates query to all processes in its dependent set
- sets local variable  $\text{num}(i) = \text{no. of query msg sent}$

    If it isn't engaging query,

- If  $P_k$  has been blocked since engaging query, send reply immediately
- else, discard

Process  $P_k$  has variable  $\text{wait}_k(i)$  [true/false] to denote it has been blocked  
since  $\text{query}(i, j, k)$  from  $P_i$

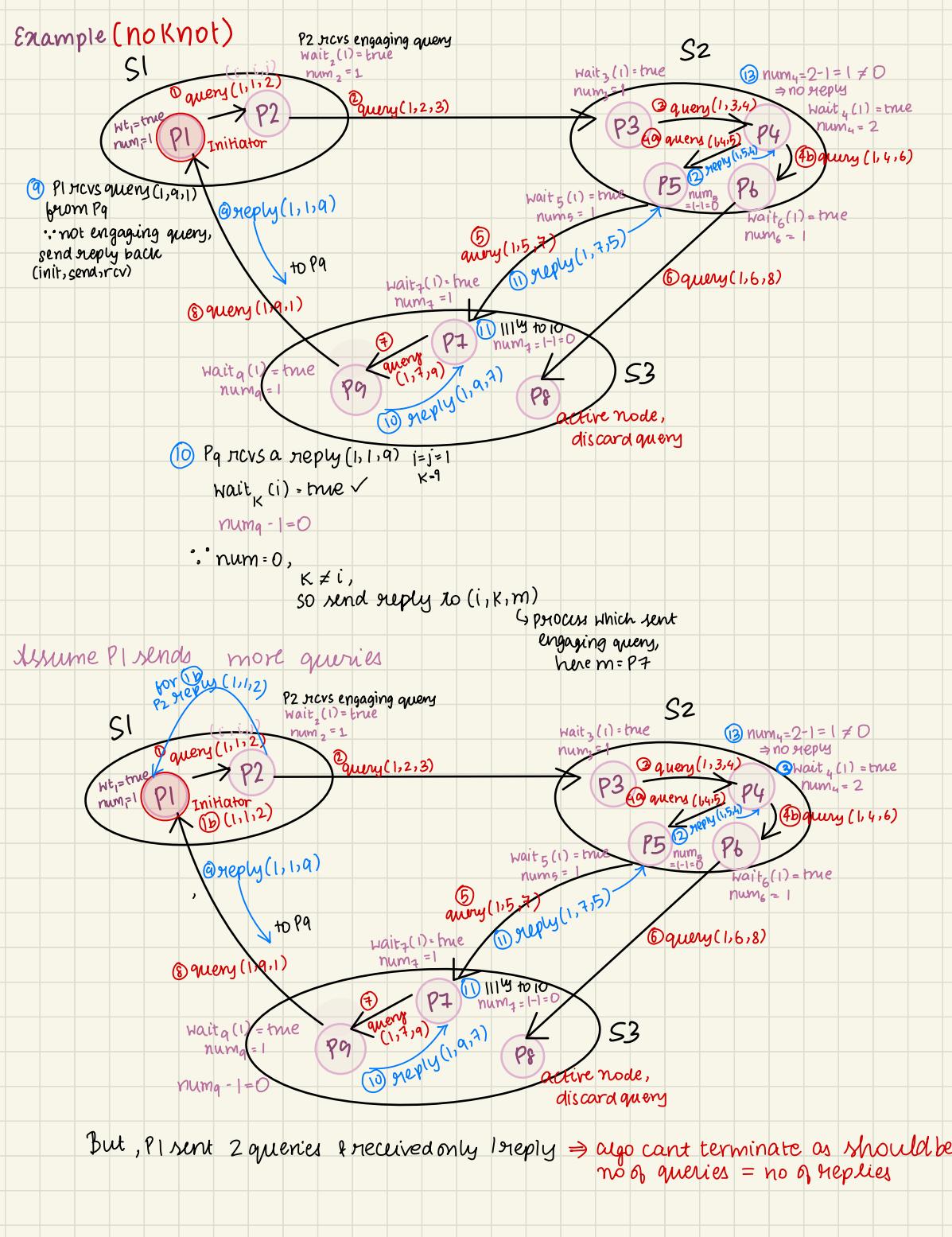
NOTE: Process sends a reply to an engaging query after it recvs replies to every  
query it sent out for this engaging query

↳ Initiator detects deadlock when it rcvs replies to all queries it sent out

<https://www.youtube.com/watch?v=qPINNeXtfUc>

Algorithm:

1. Initiation by a blocked process  $P_i$ :  
 $\xrightarrow{\text{Initiator is sender}}$   
send  $\text{query}(i, i, i)$  to all processes  $P_j$  in the dependent set  $DS_i$  of  $P_i$ ;  
 $\text{num}(i) := |DS_i|$ ;  
 $\text{wait}(i) := \text{true}$ ;
2. Blocked process  $P_k$  receiving  $\text{query}(i, j, k)$ :  
if this is **engaging** query for process  $P_k$  /\* first query from  $P_i$  \*/  
then send  $\text{query}(i, k, m)$  to all  $P_m$  in  $DS_k$ ;  
 $\text{num}(i) := |DS_k|$ ;  $\text{wait}(i) := \text{true}$ ;  
else if  $\text{wait}(i)$  then send a  $\text{reply}(i, k, j)$  to  $P_j$ .  
    ↳ from the process  $P_j$  that  $P_k$  rcvd query from
3. Process  $P_k$  receiving  $\text{reply}(i, j, k)$   
if  $\text{wait}(i)$  then  
 $\text{num}(i) := \text{num}(i) - 1$ ;  
if  $\text{num}(i) = 0$  then  
    if  $i == k$  then declare a **deadlock**.  
    else send  $\text{reply}(i, k, m)$  to  $P_m$ , which sent the engaging query.



Modify diagram by adding link  $b/w$   $P \rightarrow P_1$  creating a knot

