UCS2701 DISTRIBUTED SYSTEMS

Unit 2

Logical Time and Global State

Logical Time : Physical clock synchronization : NTP - A framework for a system of logical clocks - scalar time - Vector time : Message ordering and group communication : Message ordering paradigms - Asynchronous execution with synchronous communication - synchronous program order on an asynchronous system - Group communication - Causal Order (CO) , Total order : Global state and snapshot recording algorithms: Introduction - system model and definitions - Snapshot algorithms for FIFO channels.

* Physical Clock Synchronization

→ In centralized systems, there is only a single clock. A process gets the time by simply issuing a system call to the Kernel.

→ In distributed systems, there is no global clock or common memory.

→ Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time. Due to different clock rates, the clocks at various sites may diverge with time. Clock synchronization is used to correct clock skew.

→ Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed physical clocks.

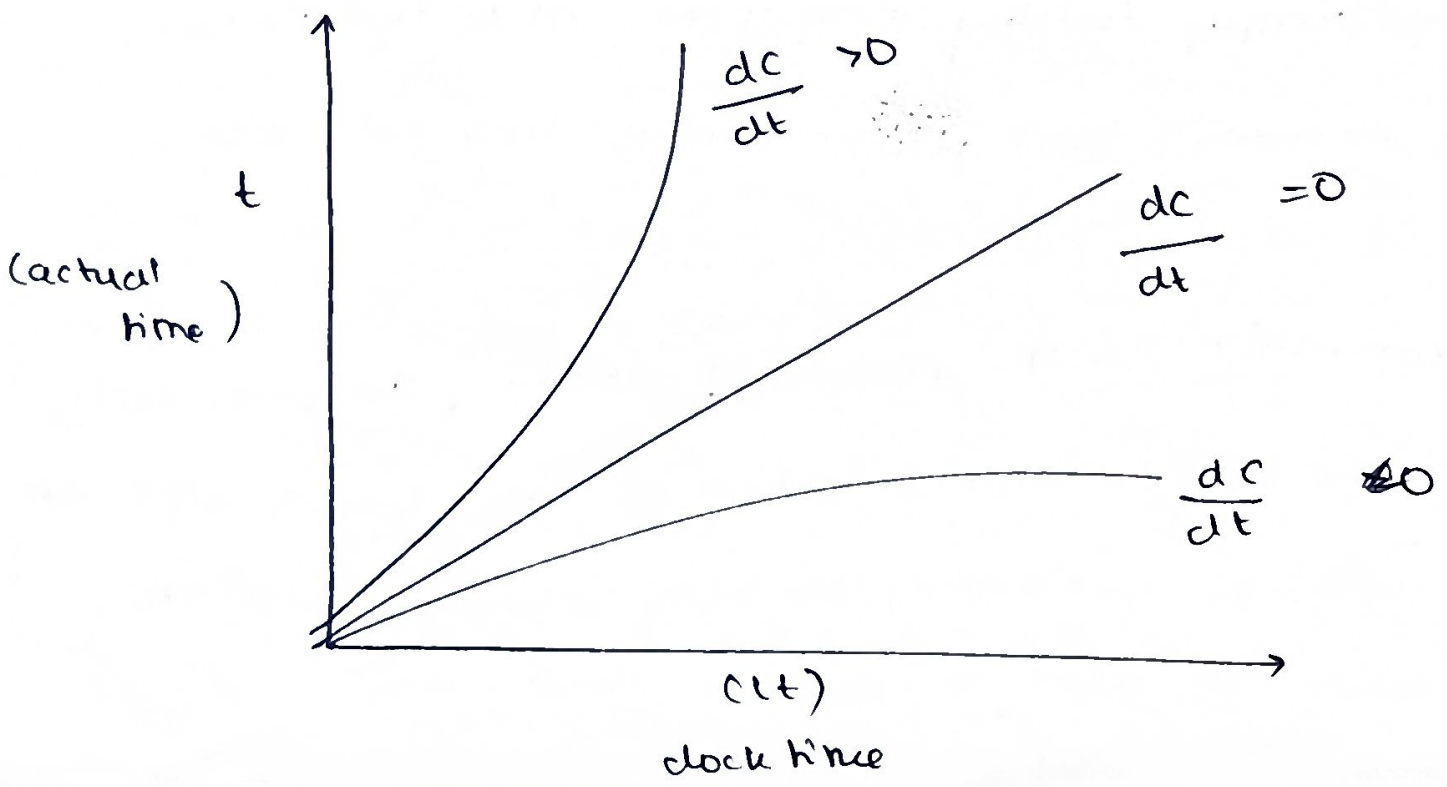
Physical Clock Inaccuracies

① Offset : Difference between the time reported by a clock and the real time.

$$\text{offset}(\text{Clock}_a) = C_a(t) - t$$

② Skew : The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock C_a relative to clock C_b at time t is $(C'_a(t) - C'_b(t))$

③ Drift : The drift of clock C_a is the second derivative of the clock value with respect to time, namely $C''_a(t)$. Relative to another clock C_b, it would be $C''_a(t) - C''_b(t)$.

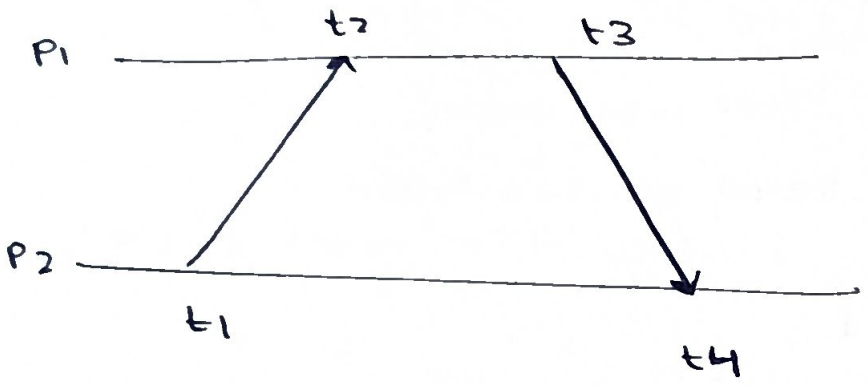


Algorithms for Synchronizing Physical Clocks

1. Christian's algorithm
2. Berkeley's algorithm
3. Network Time Protocol (NTP)

A. Chrishans Algorithm

Consider two clocks P1 and P2



$t_4 - t_1$ = round triptime (RTT)

$t_3 - t_2$ = computation delay

$t_2 - t_1$, $t_4 - t_3$ } = communication delay

Step1 : P2 computes RTT

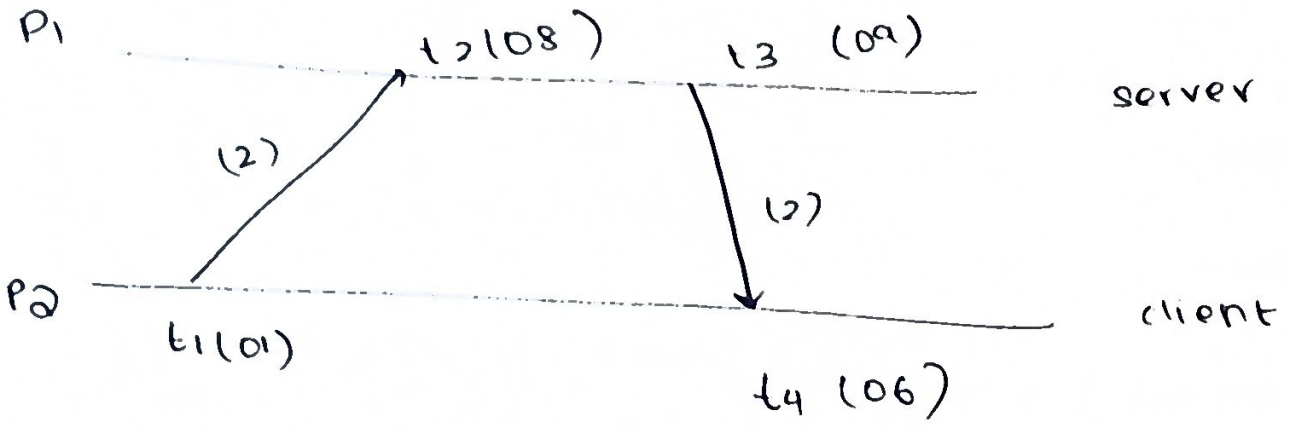
Step2 : P2 sends the computation delay

Step3 : P2 calculates $\frac{(RTT - comp.\ delay)}{2}$ = one-way delay

Step4 : Adjust t_4's time :

$$t_4' = t_3 + \text{one way delay}$$

Example Consider the following scenario:



Step1 : compute RTT = $t_4 - t_1 = 5$

Step2 : compute computation delay = $t_3 - t_2 = 9 - 8 = 1$

Step3 : compute $\frac{(RTT - comp.delay)}{2} = \frac{5-1}{2} = 2$ → one way delay

Step4 : t4 needs to adjust its time based on t3's time

$t_4' = t_3 +$ one-way delay

$t_4' = 9 + 2$

$t_4 = 11$

Limitations of Christian's Algorithm → algo will fail when the two communication delays are not the same.

B. Berkeley Algorithm

→ The Berkeley algorithm works by averaging the clocks of all participating nodes to minimize discrepancy.

→ One node (called the master) is responsible for initiating and coordinating the synchronization process

Step1 : Master polls the nodes - The master node polls all participating nodes for their current time. Each node replies with its local clock time.

Step2 : Adjust for Round Trip Delay - When the master receives the replies, it estimates the round trip time (RTT) for each node, and adjusts the received time values to account for the communication delay.

$$T_{adjusted} = T_{received} + \frac{RTT}{2}$$

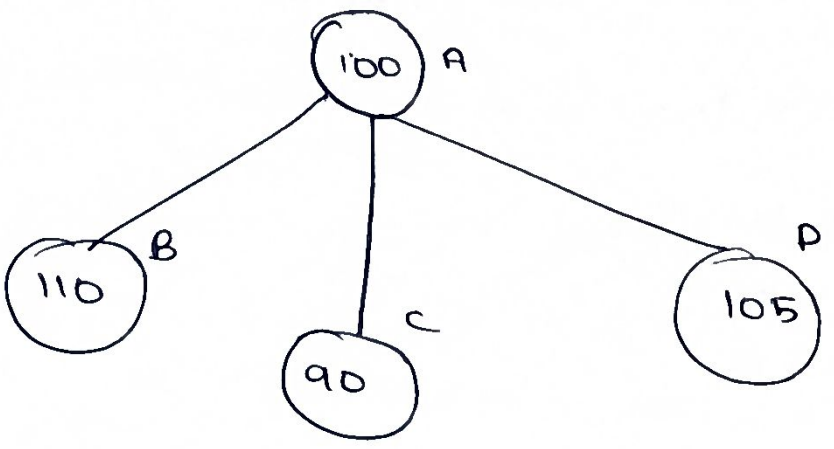
Step3 : Calculate average time

$$T_{average} = \frac{\Sigma \text{Adjusted Times of all nodes}}{\text{number of nodes}}$$

Step4 : Send adjustments to nodes

$$Offset = T_{average} - T_{local}$$

Example Concider the following scenario



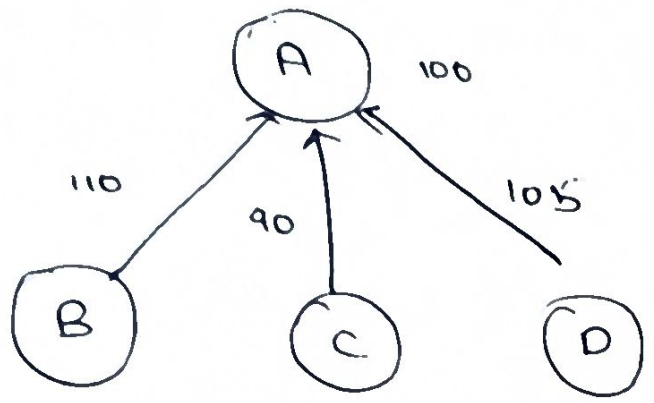
RTT

B = 4

C = 6

D = 8

Step1 : A polls times from each node



Step2 : Adjust for RTT

one way delay ⟶ node B = 2

node C = 3

node D = 4

Adjusted values: B = 112

C = 93

D = 109

Step3 : Calculate the average time

$$\text{average time} = \frac{100 + 112 + 93 + 109}{4} = 103.5$$

Step4 : Compute offsets

offset = average time - node's time

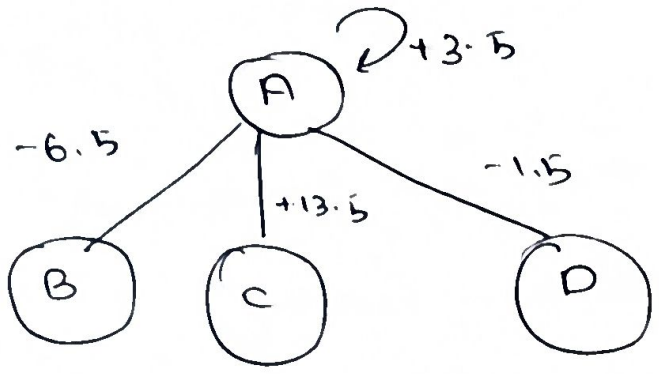
A : 103.5 - 100 = +3.5

B : 103.5 - 110 = - 6.5

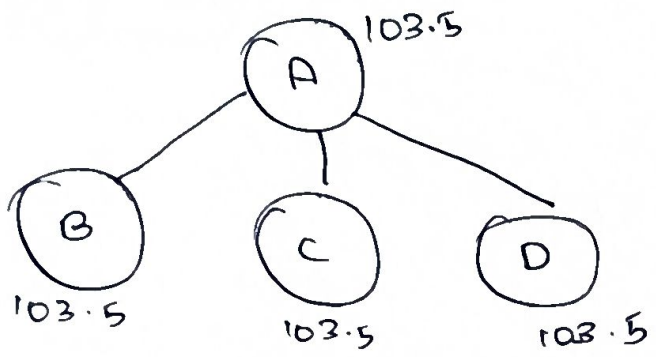
C : 103.5 - 90 = +13.5

D : 103.5 - 105 = - 1.5

Send offsets to each node



Steps : Compute the new time



c. **Network Time Protocol (NTP)**

→ Enables clients across the Internet to be accurately synchronized to UTC despite message delays.

→ works in a hierarchical fashion

→ Primary servers are connected directly to a time source such as a radio clock ; secondary servers are synchronized with primary servers.

→ The servers are connected in a logical hierarchy called a synchronization subnet whose levels are called strata.

→ Stratum 2 servers are synchronized directly with the primary servers.

→ Stratum 3 servers are synchronized with stratum 2 servers and so on.

Working

→ Both servers exchange timing messages to compute the **Offset(O)** and Delay(D) between their clocks.

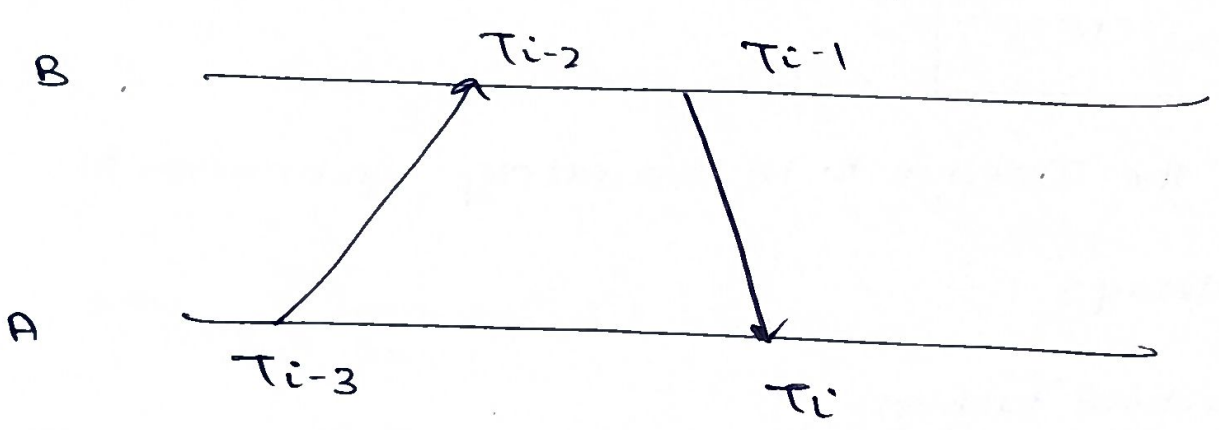
Oi → how much the 2 clocks differ

Di → measures the time taken for messages to travel between the servers.

→ The key relationship is: $A(t) = B(t) + O$

→ The servers exchange 4 time shamps:

T_{i-3}, T_{i-2}, T_{i-1}, T_i



Step1 : Calculate offset

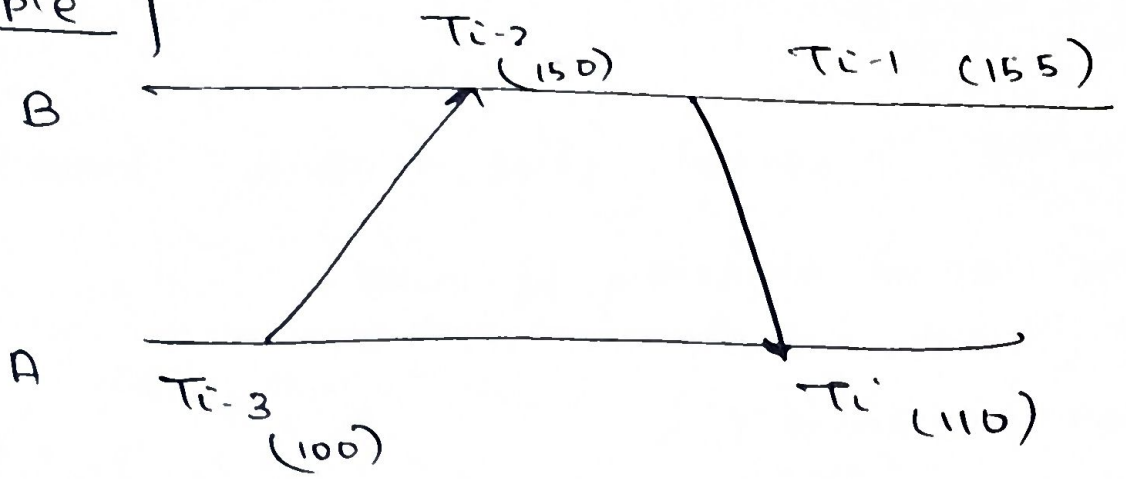
$$O = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2}$$

Step2: Calculate round trip delay

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$$

Step3: The algorithm stores several pairs of (O_i, D_i). Choose the offset corresponding to the minimum delay D_i.

Example



Step1 : Compute O

$$O = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2}$$

$$= \frac{(150-100) + (155-110)}{2} \quad = \frac{50+45}{2} \quad = \underline{47.5}$$

Step2: Compute Round Trip Delay

$$D_i = (T_i - T_{i-3}) + (T_{i-1} - T_{i-2})$$

$$= (110-100) + (155-150)$$

$$= 5$$

Step3 : Adjust A's clock value : $T_i + O$

$$= 110 + 47.5 \quad = \underline{57.5}$$

* Physical Clocks vs. Logical Clock

Physical Clock - internal clock present in a computer

Logical Clock - Keeps track of event ordering among related (causal) events.

* Happened Before Relation

→ Lamport defined the 'happened before' relation - denoted as $\rightarrow$, which describes the causal ordering of events.

→ There are 2 possible cases:

(i) a and b are in the same process and a occurs before b then

$$a \rightarrow b.$$

(ii) a and b are in different processes, and if a is in the event of sending a message 'm' in one process, and b is the event of receiving that message m in another process, then

$$a \rightarrow b.$$

$a \rightarrow b \Rightarrow$ a causally affects b

* Strict Partial Order

→ Lamport's causal ordering also has the following properties:

(i) irreflexivity $\Rightarrow a \not\rightarrow a$

(ii) antisymmetric $\Rightarrow a \rightarrow b \Rightarrow b \not\rightarrow a$

(iii) transitive $\Rightarrow a \rightarrow b, b \rightarrow c \Rightarrow a \rightarrow c$

* Lamport's Logical Clock Algorithm

if $a \rightarrow b$, then $c(a) < c(b)$

1. If a and b belong to the same process and $a \rightarrow b$

$$c_i(b) = c_i(a) + d$$

d is usually 1

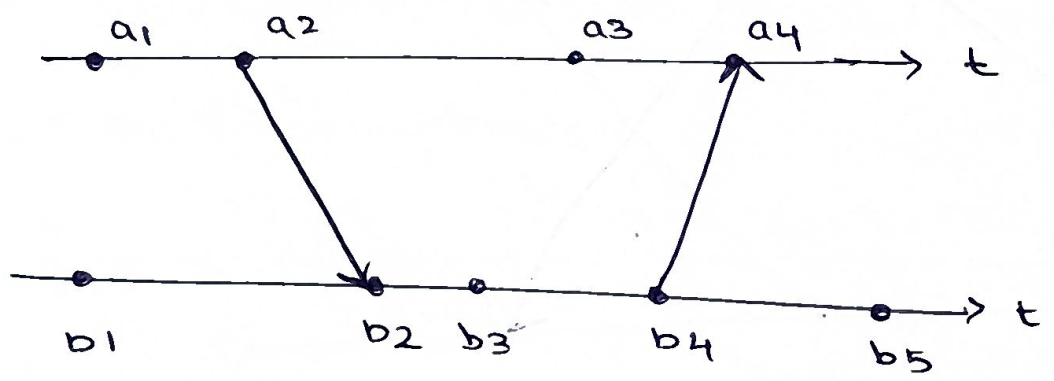
2. If and b belong to different processes and $a \rightarrow b$, i.e a is a send event and b is a receive event, m is the message

$$t_m = c_i(a)$$

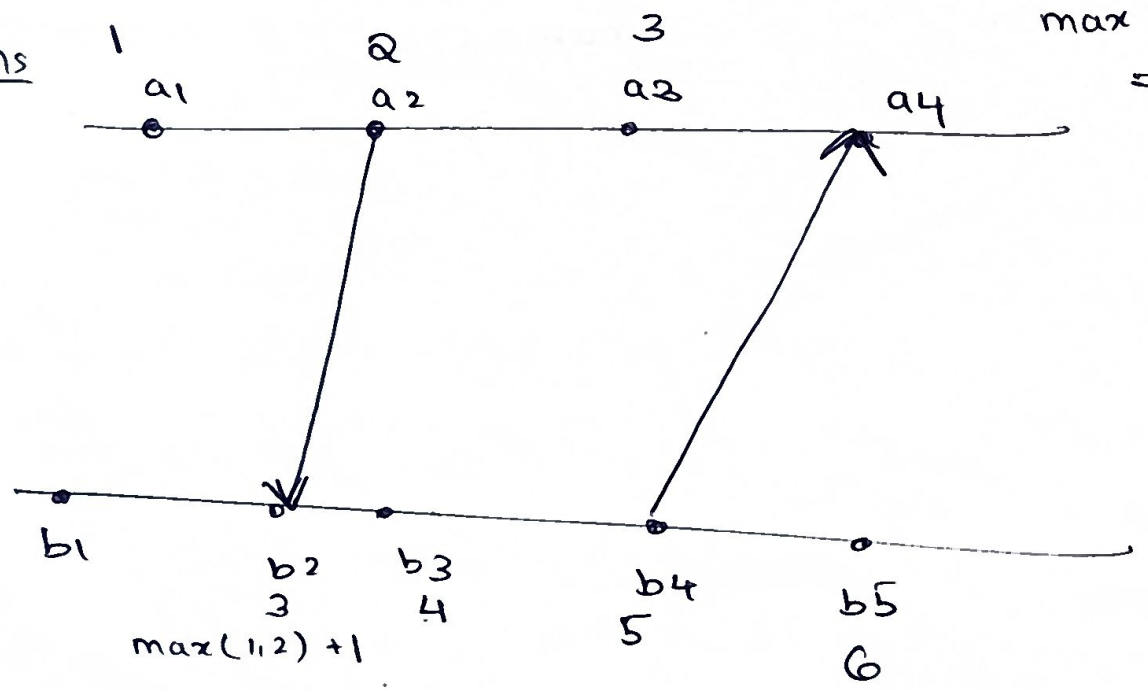
$$c_j(b) = \max(c_j(b), t_m) + 1$$

Example 1 Calculate the Lamport's logical clock values

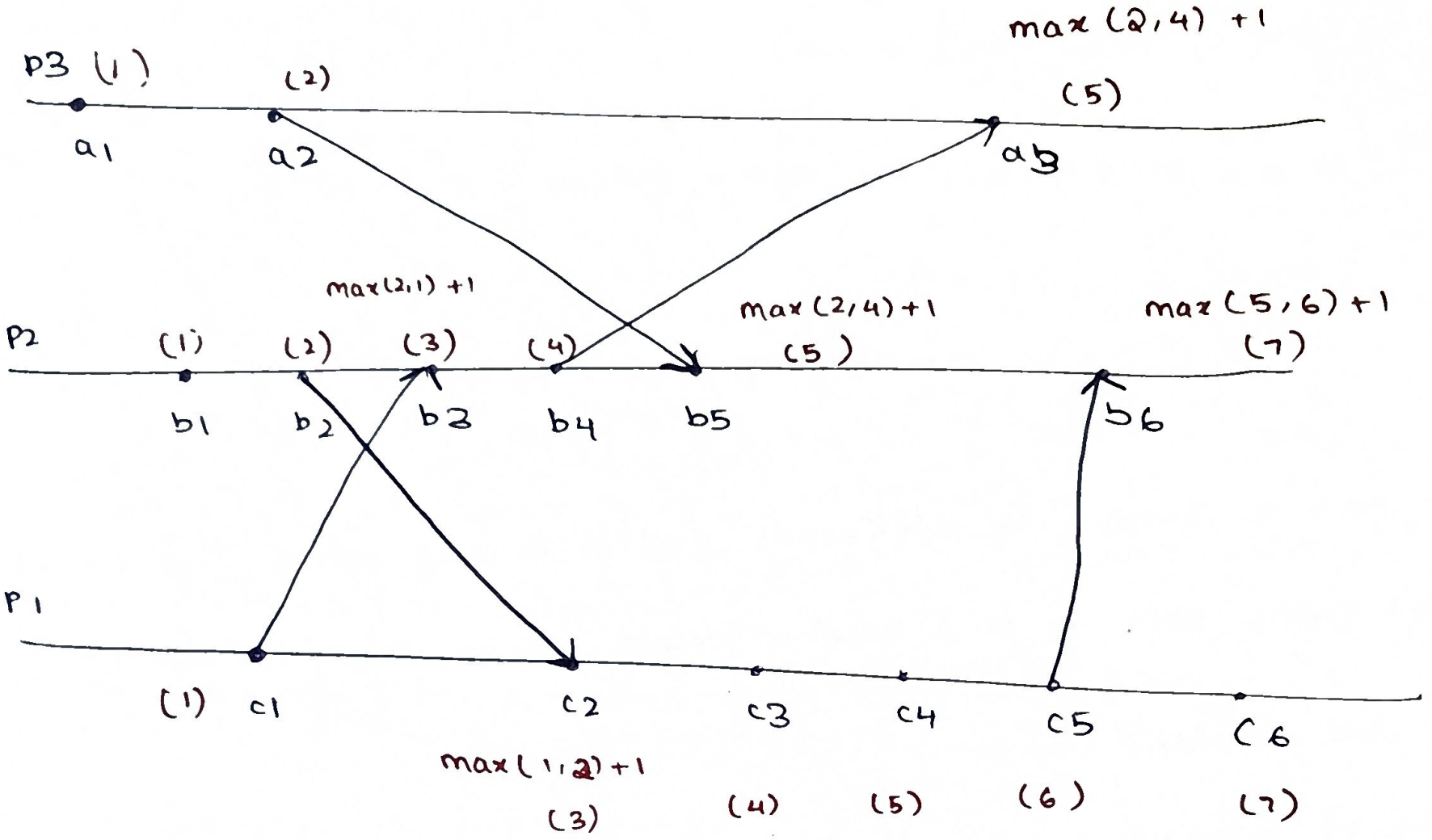
(i)



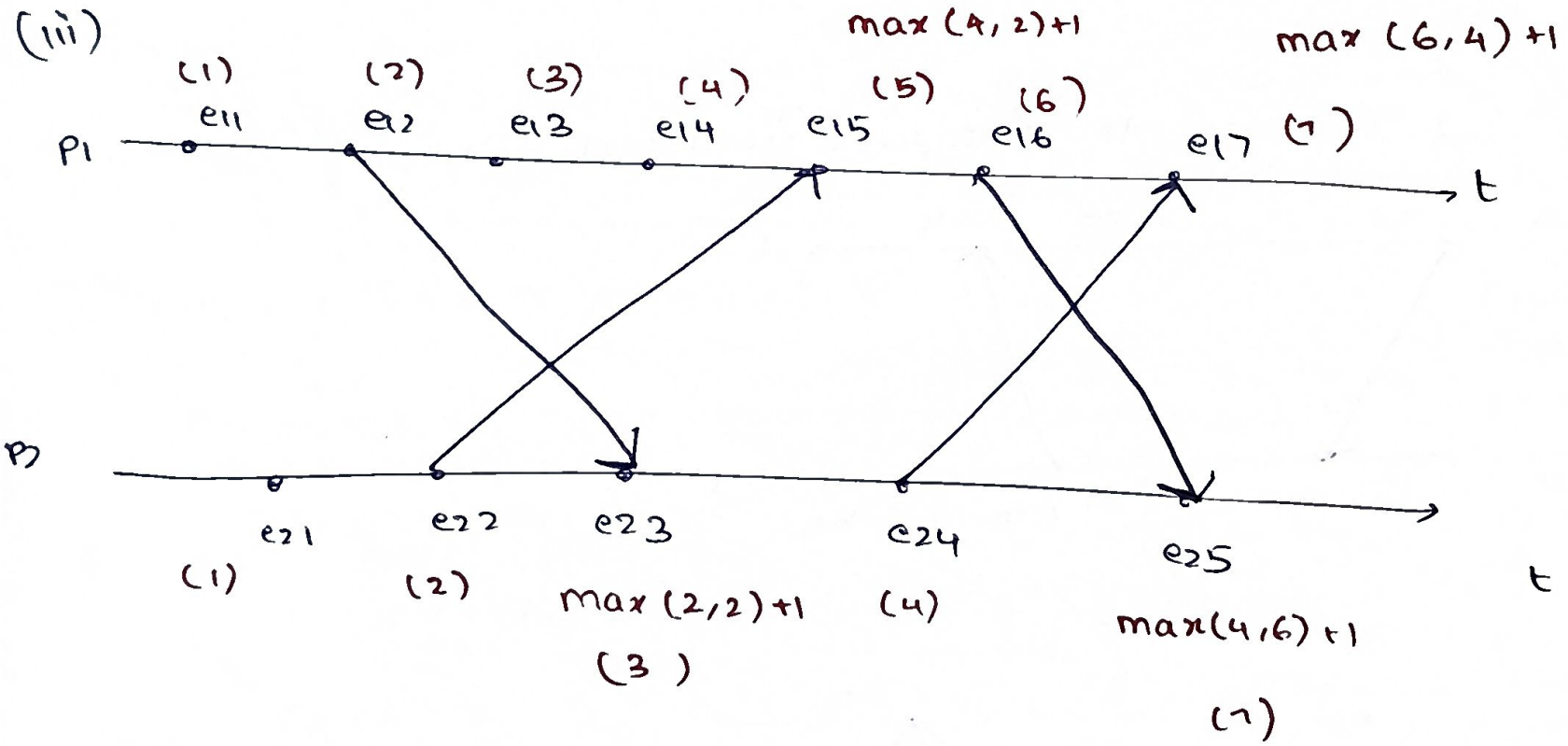
Ans



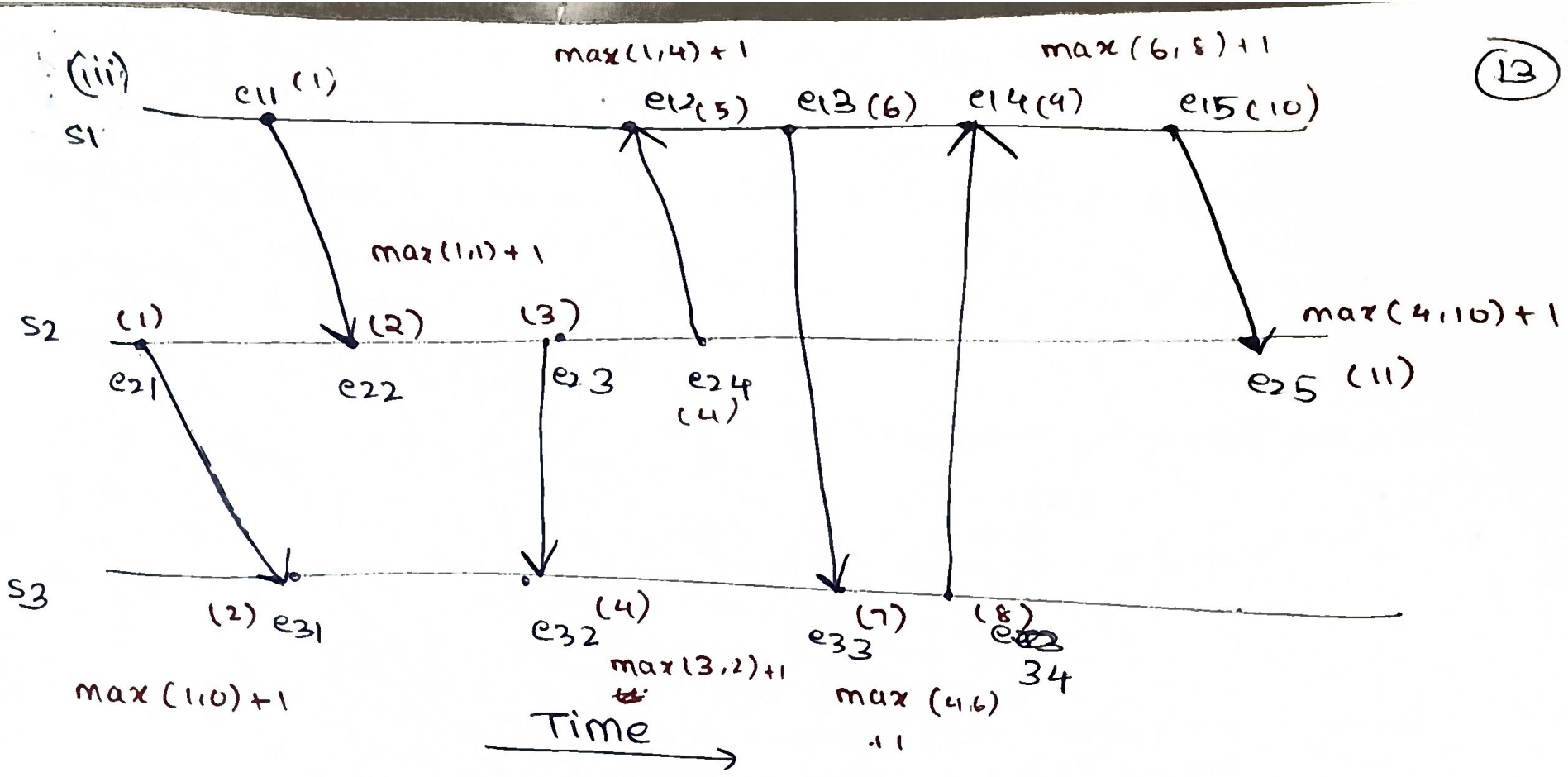
(ii)


max (2,4) +1
P3 (1)
(2)
(5)
a1
a2
a3
max(2,1) +1
max(2,4)+1
max (5,6) +1
P2
(1)
(2)
(3)
(4)
(5)
(7)
b1
b2
b3
b4
b5
b6
P1
(1)
c1
c2
c3
c4
c5
c6
max(1,2)+1
(3)
(4)
(5)
(6)
(7)

(iii)


max (4,2)+1
max (6,4) +1
(1)
(2)
(3)
(4)
(5)
(6)
(7)
P1
e11
e12
e13
e14
e15
e16
e17
t
P2
e21
e22
e23
e24
e25
t
(1)
(2)
max (2,2)+1
(3)
(4)
max(4,6)+1
(7)

(iii)



* Limitations of Lamport's logical clock

→ With Lamport's logical clocks, if $a \to b$, then $C(a) < C(b)$

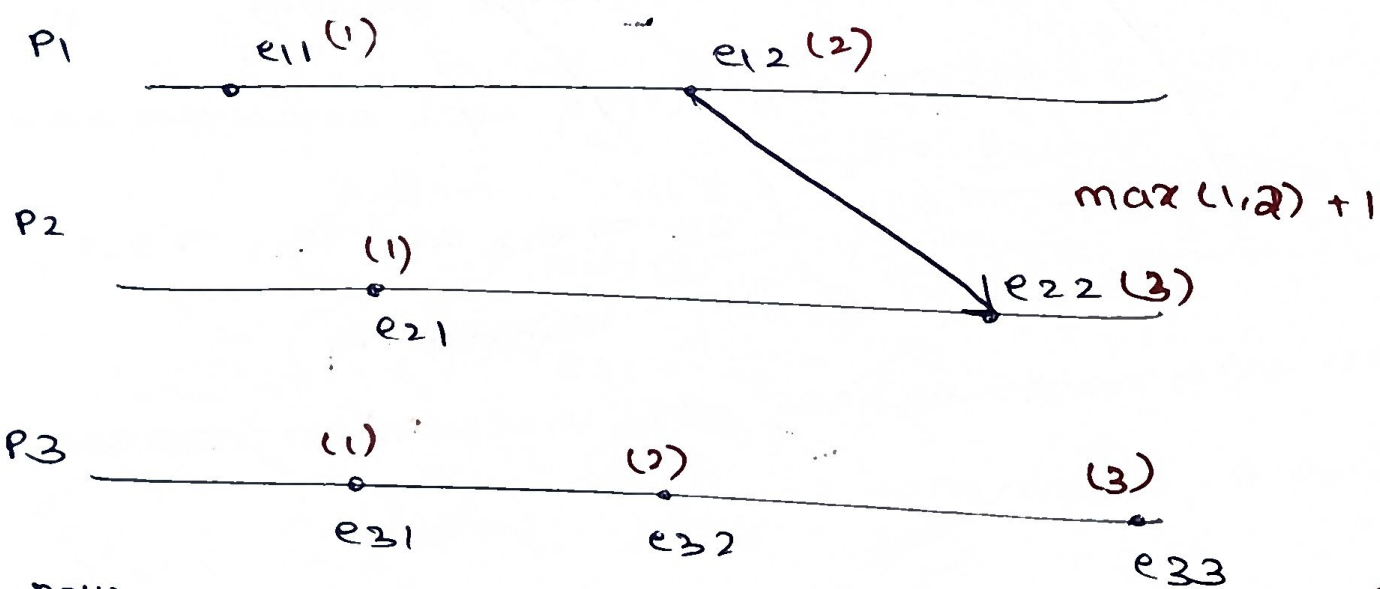
→ However, if $C(a) < C(b)$, it is not necessarily true that $a \to b$

i.e

CANNOT DETERMINE CAUSALTY FROM TIMESTAMPS

Consider the following example:

refer to concurrency first



now

$C(e_{11}) < C(e_{22})$ and $e_{11} \to e_{22}$ = True

$1 < 3$

$e_{11} \to e_{12}$

$e_{12} \to e_{22}$

(not concurrent events)

but,

$C(e_{11}) < C(e_{32})$ but $e_{11} \not\to e_{32}$

$1 < 2$

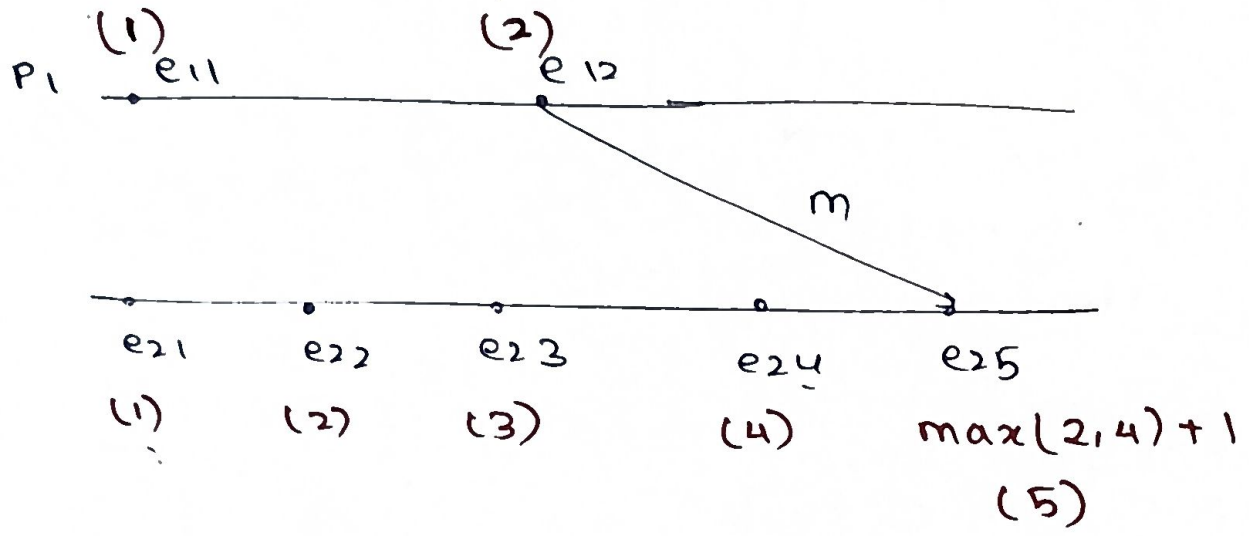
e_{11} and e_{32} are concurrent events, not causally related

* Concurrency of Events

If $a \not\rightarrow b$ and $b \not\rightarrow a$

then all b

Consider the following example:



Here, $e_{11} \parallel e_{21}, e_{22}, e_{23}, e_{24}$

$e_{12} \parallel e_{21}, e_{23}, e_3, e_{24}$

$e_{11} \not\parallel e_{12}, e_{25}$

$e_{12} \not\parallel e_{25}$

Determine concurrency by checking for causality first

$e_{11} \rightarrow e_{12}$ (same process)

$e_{12} \rightarrow e_{25}$ (msg sent across processes)

$e_{11} \rightarrow e_{25}$ ($e_{11} \rightarrow e_{12}$ and $e_{12} \rightarrow e_{25}$, transitivity)

All other combinations are independent and are hence concurrent

* Vector Clocks

→ maintains a vector of values for every event that happens in all processes

→ In vector clocks, if $a \rightarrow b$

$$\forall k \quad C_a[k] < C_b[k]$$

and if $a \not\rightarrow b$

$$\forall k \quad C_a[k] \not< C_b[k]$$

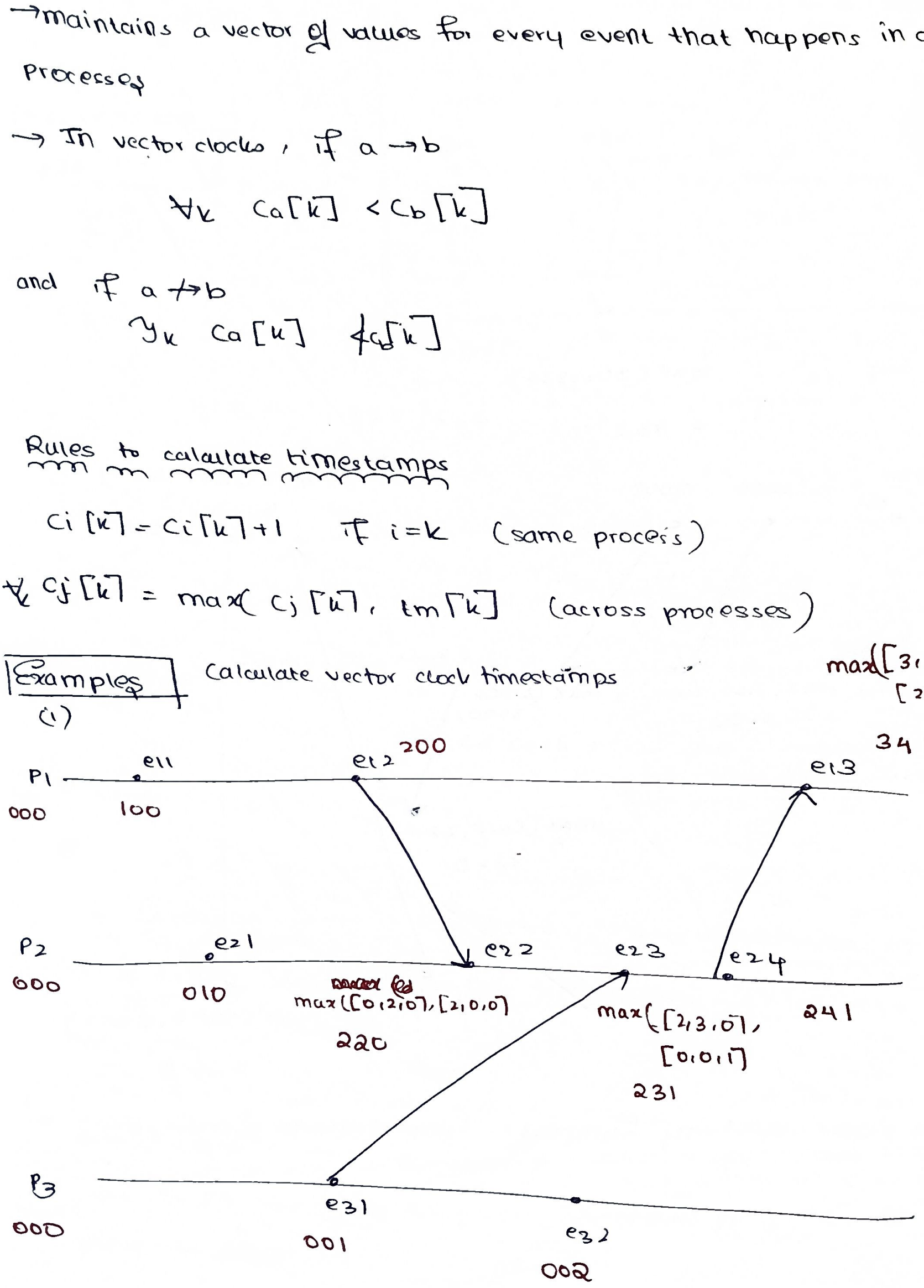
Rules to calculate timestamps

$C_i[k] = C_i[k] + 1$ if $i = k$ (same process)

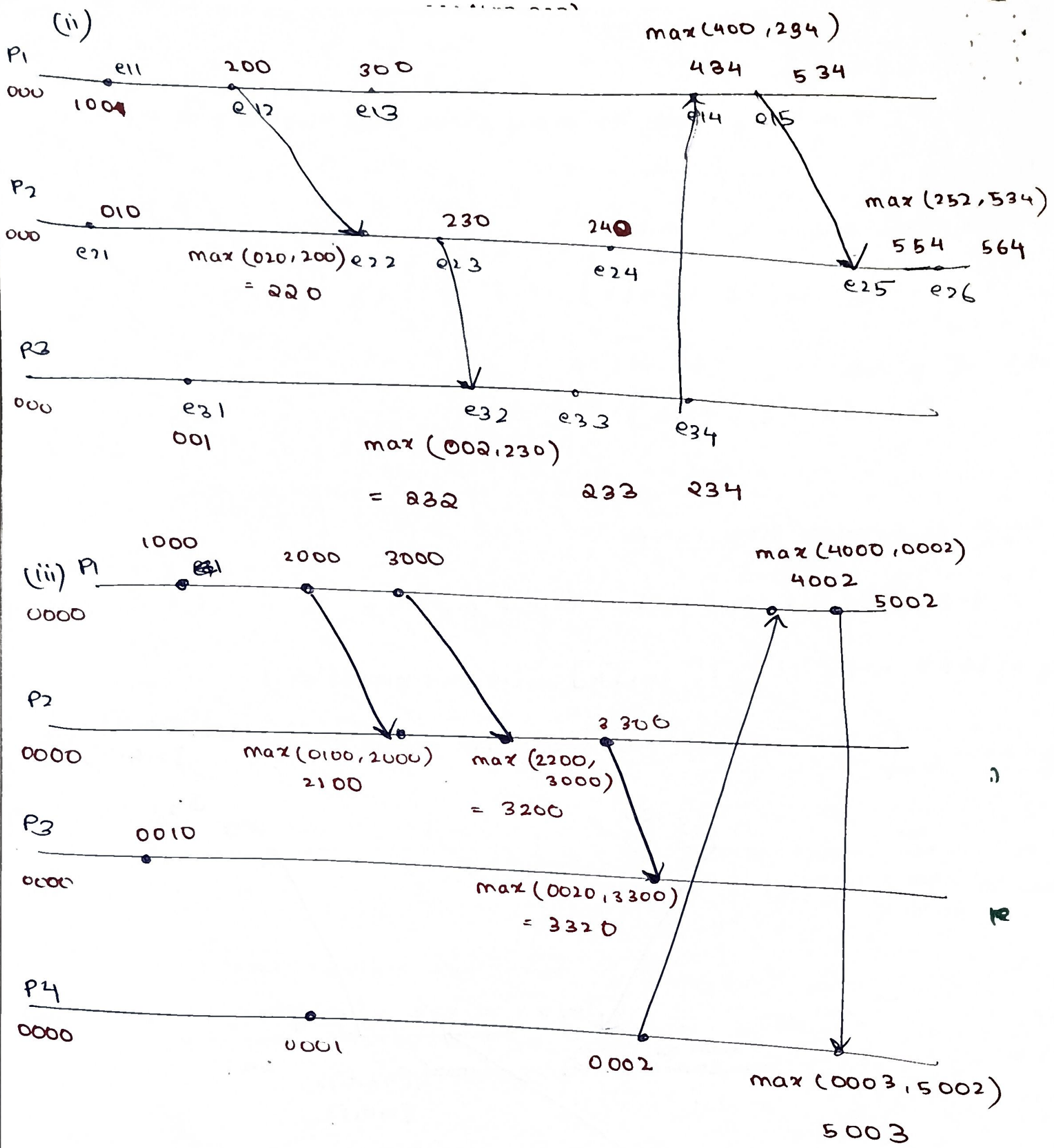
$\forall k \; C_j[k] = \max(C_j[k], t_m[k])$ (across processes)

Examples Calculate vector clock timestamps

(1)



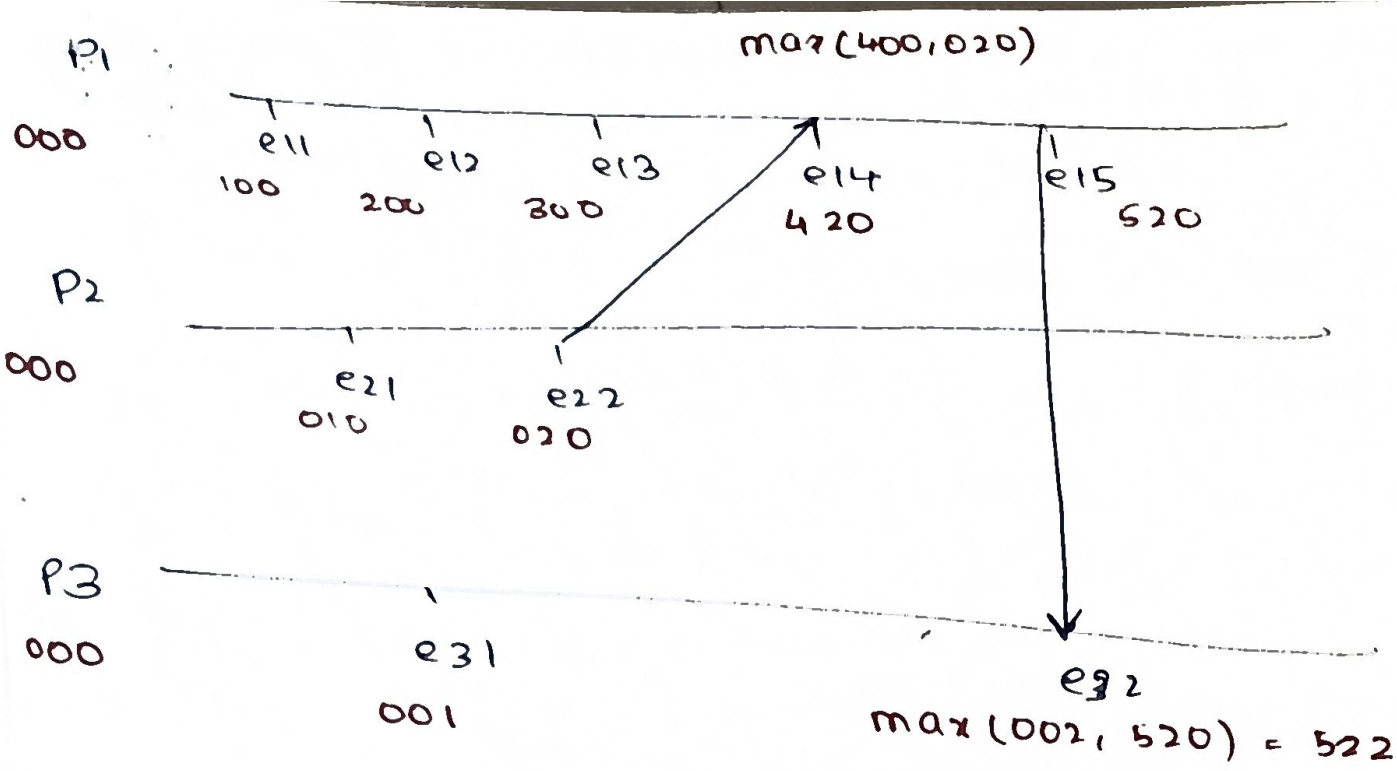
(ii)



(iii)

* Vector @locks resolving Lamport's logical @lock limitations

Consider the following scenario:



consider e13 and e31

(300) (001)

$3 > 0$

$0 \leq 0$ ⇒ we cannot say that $e13 \rightarrow e31$

$0 \leq 1$ ⇒ they are concurrent events

consider e14 and e32

(420) (522)

$4 < 5$

$2 \leq 2$ ⇒ $e14 \rightarrow e32$

$0 \leq 2$

→ Vector clocks resolve the issue of Lamport's Logical clock, because by looking at the timestamp, we can determine causality

Applications

1. Distribute debugging
2. causal distributed shared memory
3. Establish global breakpoints

* Drawbacks of vector clocks

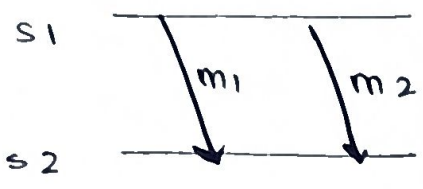
→ need to maintain memory units for tracking all events in all processes as vectors.

→ The total no. of processes may not be known in advance.

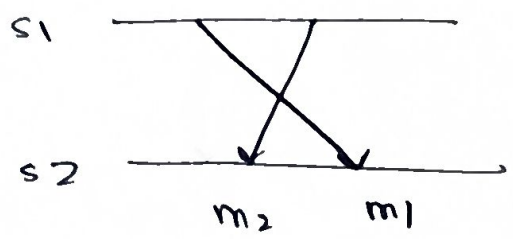
* Causal Ordering of Messages

if send (m_1) $\longrightarrow$ send (m_2) then

rec (m_1) $\longrightarrow$ rec (m_2)



ideal behaviour



non-ideal behaviour

A. Causal Ordering of Messages in Broadcast Condition

Sender -i

increment ci[i]

tm = ci[i]

broadcast (tm)

Receiver -j

1. If $c_j[i] = tm[i] - 1$
2. If $\forall k\ c_j[k] \geq tm[u]$, where $k \neq i$

If ① and ② satisfy, deliver

else buffer the message

Case1 - Broadcasts are not mixed with other concurrent sends/ concurrent broadcasts

Case2 - Concurrent broadcasts

Case3 - missing receives, single buffer item

Case4 : multiple buffer items for a single system

Case5 : missing msg. from sys other than current sys

Case - multiple buffers

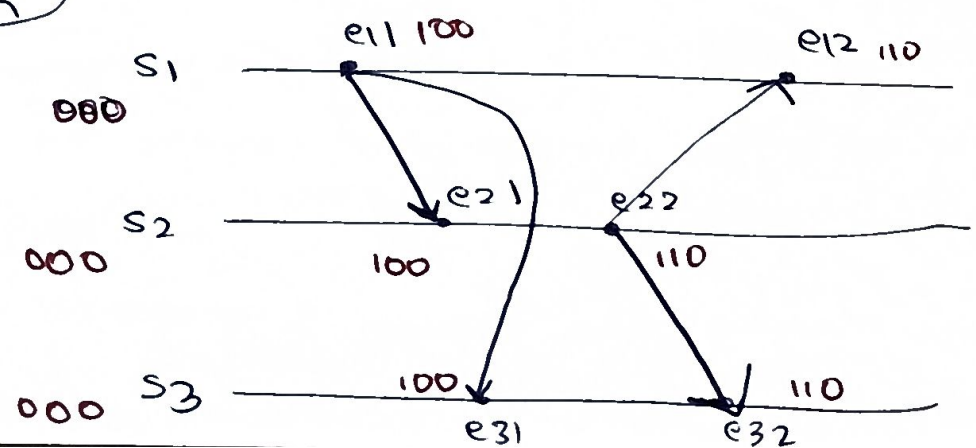
What the rules mean

Rule1 : Have I missed a message from the current sender?

Rule2 : Have I missed a message from any other sender other than the current sender?

Example 1 Apply the causal ordering of messages on the following broadcast scenario

Type Case1



(i) at e_{21}^m $C_2 = 000$ for that process

$t_m = 100$ t at sender's side

Rule 1 $C_j[i] = t_m[i] - 1$

$\Rightarrow 0 == 1 - 1$ ✓ satisfies

Rule 2 $C_2[2,3] \geq t_m[2,3]$

$\Rightarrow 0 \geq 0$

$0 \geq 0$ ✓ satisfies

calculate e_{21} as

$\max[(0,0,0), (1,0,0)]$

$\Rightarrow$ deliver the message

$\Rightarrow$ update the vector clock

$e_{21} = 100$

(X) ~~X~~

For the vector clock update, only increment ~~for the process~~ alone the sender's process

i.e (0,0,0) → (1,0,0)

This is because we should consider only communicating events and not computing / processing events.

(ii) at e_{31} $C_3 = 000$

$t_m = 100$

Rule 1 $C_j[i] = t_m[i] - 1$

$\Rightarrow 0 = 1 - 1$ ✓ satisfies

Rule 2 $C[2,3] \geq t_m[2,3]$

$0 \geq 0$ and $0 \geq 0$ $\Rightarrow$ satisfies ✓

$\Rightarrow$ deliver the message

$\Rightarrow$ update the vector clock

$e_{31} = 100$

(X) The most important thing is - don't increament the recipient's timestamps when processing C_0 & t_m. This is how this differs from the OG vector clock algo.

(iii) e_{22} = 110 (basic increment)

(iv) at e_{12}

C_1[100] tm [110]

Rule 1 $C_1[2] == tm[2] - 1$ ✓ satisfies

$0 == 1 - 1$

Rule 2 $C_1[1,3] \geq tm[1,3]$

$(1,0) \geq (1,0)$

⇒ satisfies ✓

→ deliver the message

→ update the vector clock e_{12} = 110

(v) at e_{32}

C_3 [100] tm [110]

Rule 1 $C_3[2] == tm[2] - 1$

$0 == 1 - 1$ satisfies

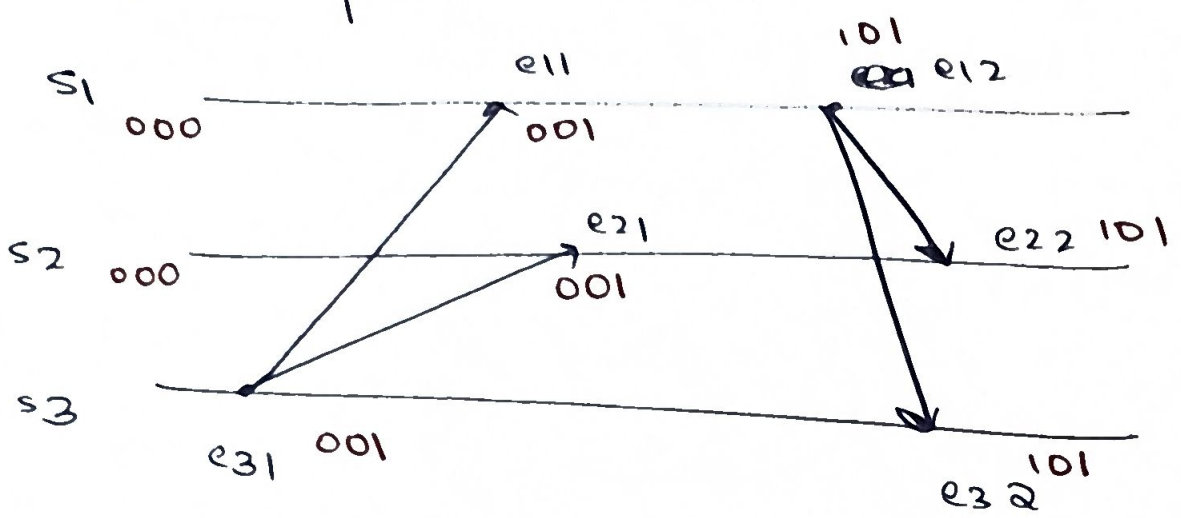
Rule 2 $C_3[1,3] \geq tm[1,3]$

$(1,0) \geq (1,0)$ ⇒ satisfies

→ deliver the message

→ update the vector clock e_{32} = 110

Example 2 (Type case 1) Apply the causal ordering of messages on the following broadcast scenario



(i) at e11

$c_1[000]$ $tm[001]$

Rule 1 $c_1^r[3] == tm^s[3] - 1$

$0 == 1 - 1$

$= 0$ satisfies ✓

Rule 2 $c_1[1,2] \geq tm[1,2]$

$(0,0) \geq (0,0)$ satisfies

⇒ deliver message

⇒ $e_{11} = 001$

(ii) at e_{21}

$c_2[000]$ $tm[001]$

at receiver message

Rule 1 $c_2^r[3] == tm^s[3] - 1$

$0 == 1 - 1$

$= 0$ satisfies ✓

Rule 2 $c_2[1,0] \geq tm[1,2]$

$(0,0) \geq (0,0)$ satisfies ✓

→ deliver msg

→ $e_{21} = 001$

(iii) $e_{12} = 101$

(iv) at e_{22} $C_2[0\,0\,1]$ $tm[101]$

Rule 1 $C_2^{r}[i^{s}] == tm[i] - 1$

$0 == 1 - 1$ satisfies ✓

Rule 2 $C_2[2,3] \geq tm[2,3]$

$(0,1) \geq (0,1)$ satisfies

⇒ deliver

$e_{22} = 101$

(v) at e_{32} $C_3[0\,0\,1]$ $tm[101]$

Rule 1 $C_3^{r}[i^{s}] == tm[i] - 1$

$0 == 1 - 1$
$= 0$
⇒ satisfies ✓

Rule 2 $C_3[2,3] \geq tm[2,3]$

$(0,1) \geq (0,1)$ ✓ satisfies

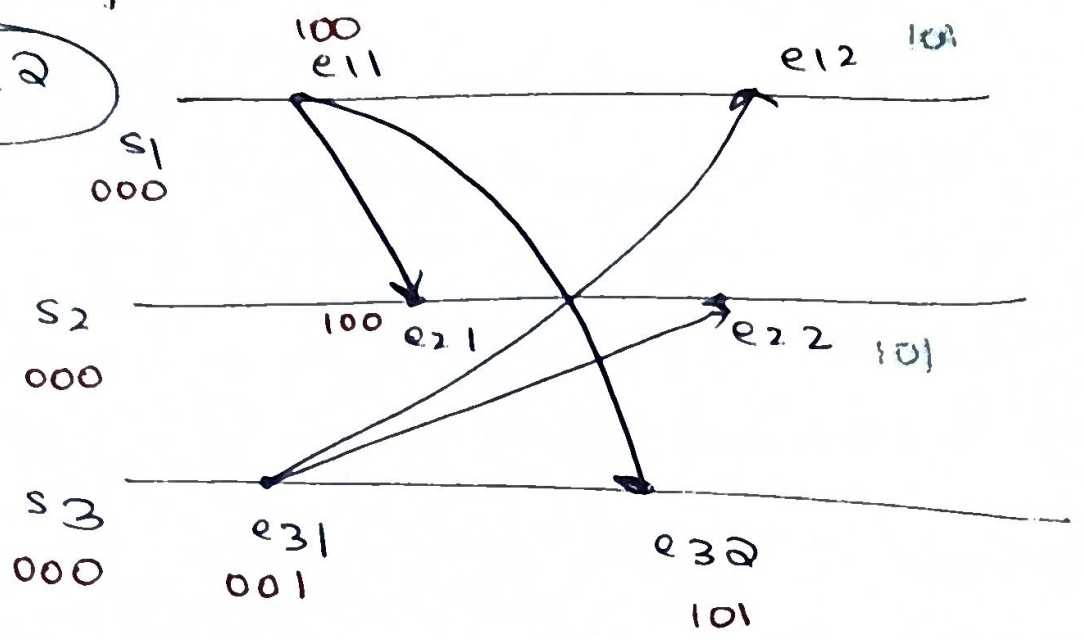
⇒ deliver

$e_{32} = 101$

Example 3

Type : Case 2

broadcast are concurrent



(i) at e21 : $C_2[000]$ $tm[100]$

Rule 1 $C_2^r[i] == tm[i] - 1$

$0 == 1 - 1$ ⇒ satisfies

Rule 2 $C_2[2,3] \geq tm[2,3]$

$(0,0) \geq (0,0)$ ⇒ satisfies

⇒ deliver, $e_{21} = 100$ ← $e_{21} = max(000, 100)$

~~Step: remember we don't do~~

(ii) at e32 : $C_3[001]$ $tm[100]$

Rule 1 $C_3^r[i] == tm[i] - 1$

$0 == 1 - 0$ ⇒ satisfies

Rule 2 $C_3[2,3] \geq tm[2,3]$

$(0,1) \geq (0,0)$ ⇒ satisfies

→ deliver

$e_{32} = max(001, 100) = 101$

(iii) at e_{12} $C_1[100]$ $tm = [001]$

Rule 1 $C_1'[3] == tm[3] - 1$

$0 == 1 - 1$ ⇒ satisfies

Rule 2 $C_1[1,2] \geq tm[1,2]$

$(1,0) \geq (0,0)$ ⇒ satisfies

⇒ deliver

⇒ $e_{12} = max(100, 001) = 101$

(iv) at e_{22} $C_2[100]$ $tm = [001]$

Rule 1 $C_2'[3] == tm[3] - 1$

$0 = 1 - 1 = 0$ ⇒ satisfies

Rule 2 $C_2[1,2] \geq tm[1,2]$

$(1,0) \geq (0,0)$ ⇒ satisfies

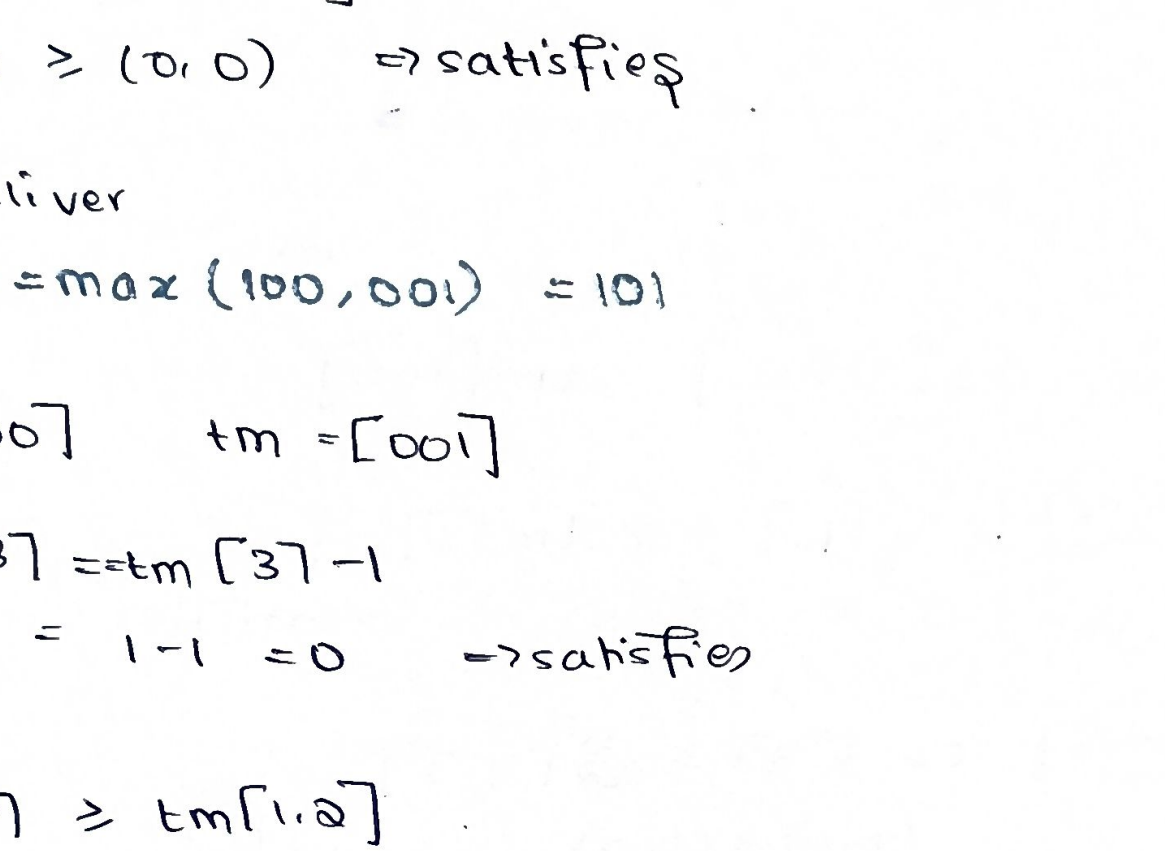
⇒ deliver

$e_{22} = max(100, 001)$

$= 101$

Example 4

Type : case 3 includes missing receives



$C_2 = [000]$

(i) e21 $tm = [100]$

Rule1 $C_2^r[0] == tm[1] - 1$

$0 == 1 - 1$ ⇒ satisfies

Rule2 $C_2[2,3] >= tm[2,3]$

$(0,1,0) >= (0,1,0)$ ⇒ satisfies

⇒ deliver

⇒ $e_{21} = max(000, 100) = 100$

(ii) e22 $C_2 = [100]$

$tm = [200]$

Rule1 $C_2^r[1] == tm[1] - 1$

$1 == 2 - 1$ ⇒ satisfies

Rule2 $C_2[2,3] \geq tm[2,3]$

$(0,1,0) \geq (0,1,0)$ ⇒ satisfies

⇒ $e_{22} = max(100, 200) = (2,1,00)$

(iii) e31 $C_3 = [000]$

$tm = [200]$

Rule1 $C_3^r[1] == tm[1] - 1$

$0 \neq 2 - 1$ ⇒ error

⇒ there is a message that has been missed

⇒ push (sender, tm) onto S_3's buffer (queue)

(1, 200)

(iv) e32 $C_3 = [000]$

$tm = [100]$

Rule1 $C_3'[1] == tm[1] - 1$

$0 == 1 - 1$ ⇒ satisfies

Rule 2 $C_3[2,3] \geq tm[2,3]$

$(0,0) \geq (0,0)$ ⇒ satisfied

⇒ deliver

⇒ $e32 = max(000, 100) = 100$

Now that there has been a successful case, handle the case in the buffer

(v) e31 B $C_3 = [100]$

$tm = [200]$

Rule1 $C_3'[1] == tm[1] - 1$

$1 == 2 - 1$ ⇒ satisfies

Rule2 $C_3[2,3] \geq tm[2,3]$

$(0,0) \geq (0,0)$ ⇒ satisfies

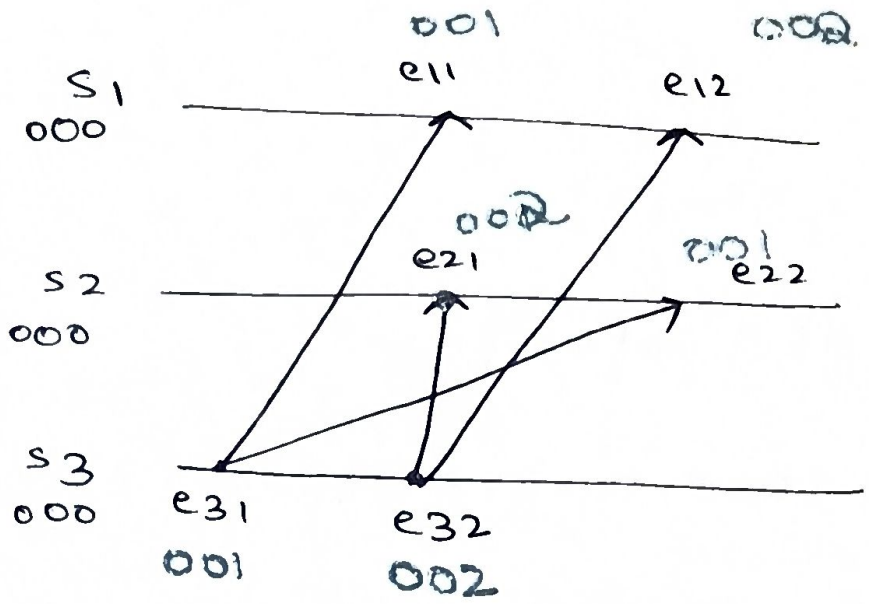
⇒ deliver

⇒ $e31 = max(100, 200)$

$= (200)$

Example 5

Type - case 3



(i) e11 $c_1 = [000]$
$tm = [001]$

Rule1 $c_1^r[3] == tm[3] - 1$ ✓

Rule2 $c_1[1,2] \geq tm[1,2]$ ✓

$\Rightarrow e_{11} = max(000, 001) = 001$

(ii) e12 $c_1 = [001]$
$tm = [002]$

Rule1 $c_1^r[3] == tm[3] - 1$ ✓

Rule2 $c_1[1,2] \geq tm[1,2]$ ✓

$\Rightarrow e_{12} = max(001, 002) = 002$

(iii) e21 $c_2 = [000]$
$tm = [002]$

Rule1 $c_2^r[3] == tm[3] - 1$ ✗ Fails

$\Rightarrow$ missing recieve

push onto buffer

S2 | (3,002) |

(iv) e_{22} $c_2 = [000]$
$tm = [001]$

Rule1 $c_2'[3] = tm[3] - 1$ ✓

Rule2 $c_2[1,2] \geq tm[1,2]$ ✓

→ $e_{22} = max(000, 001) = 001$

(v) e_{21B} $c_2 = [001]$
$tm = [002]$

Rule1 $c_2[3] = tm[3] - 1$ ✓

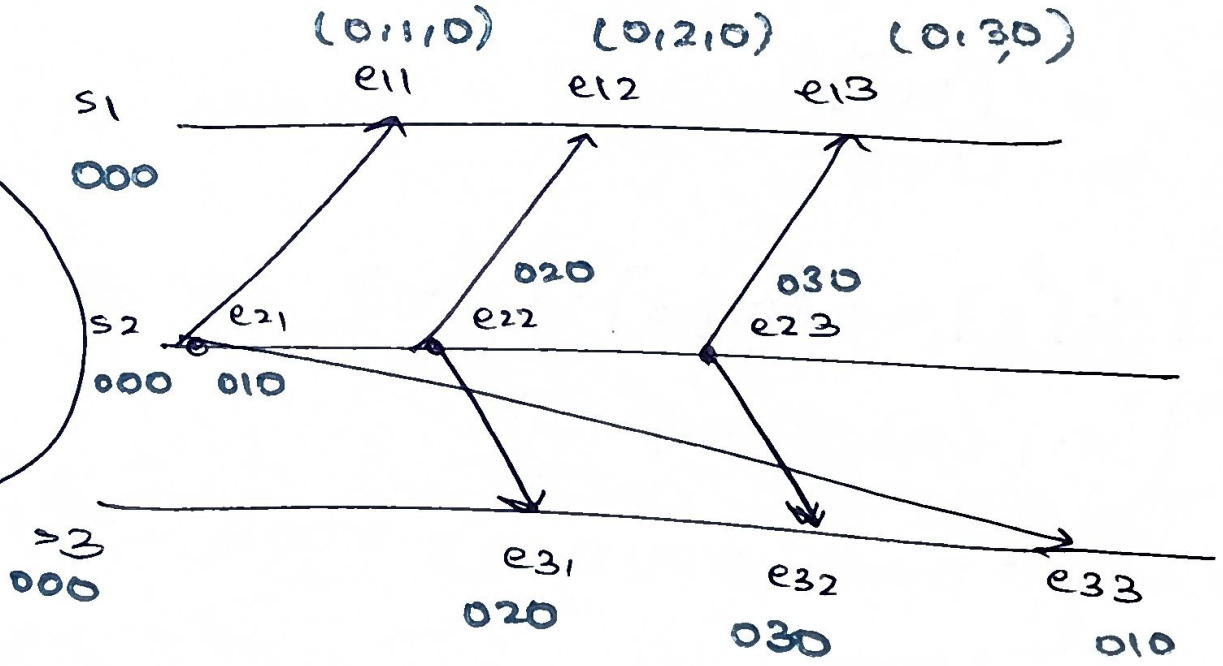
Rule2 $c_2[1,2] \geq tm[1,2]$ ✓

→ $e_{21B} = max(001, 002) = 002$

Example 6

Type: Case 4

Multiple buffer items in a single system



Ans e_{11}, e_{12} & e_{13} will not have any problem of messages received out of order

Consider (i) e_{31} $c_3 = [000]$
$tm = [020]$

$c_3'[2] == tm[2] - 1$ X → push (2, 020) onto buffer

(ii) Try e_{32} $\quad c_3 = [000]$

$tm = [030]$

$c_3'^{r}[2] == tm[2] - 1$ X $\longrightarrow$ push (2, 030) onto the buffer

(iii) Try e_{33} $\quad c_3 = [000]$

$tm = [010]$

$c_3'^{r}[2] == tm[2] - 1$ ✓

$c_3[2', 3] \geq tm[2', 3]$ ✓

$(0,0) \geq (0,0)$

$\Rightarrow$ deliver

$\Rightarrow e_{33} = max(000, 010) = 010$

(iv) By means of FIFO, now try e_{31} B

$c_3[010]$

$tm = [020]$

$c_3'^{r}[2]^{s} == tm[2] - 1$ ✓

$c_3[1,3] \geq tm[1,3]$ ✓

$(0,0) \geq (0,0)$

$\Rightarrow$ deliver

$\Rightarrow e_{31}{}_{B} = max(010, 020) = 020$

(v) Now by means of FIFO, attempt e_{32} B

$c_3 = [020]$

$tm = [030]$

$c_3'^{r}[2]^{s} == tm[2] - 1$ ✓

$c_3[1,3] \geq tm[1,3]$

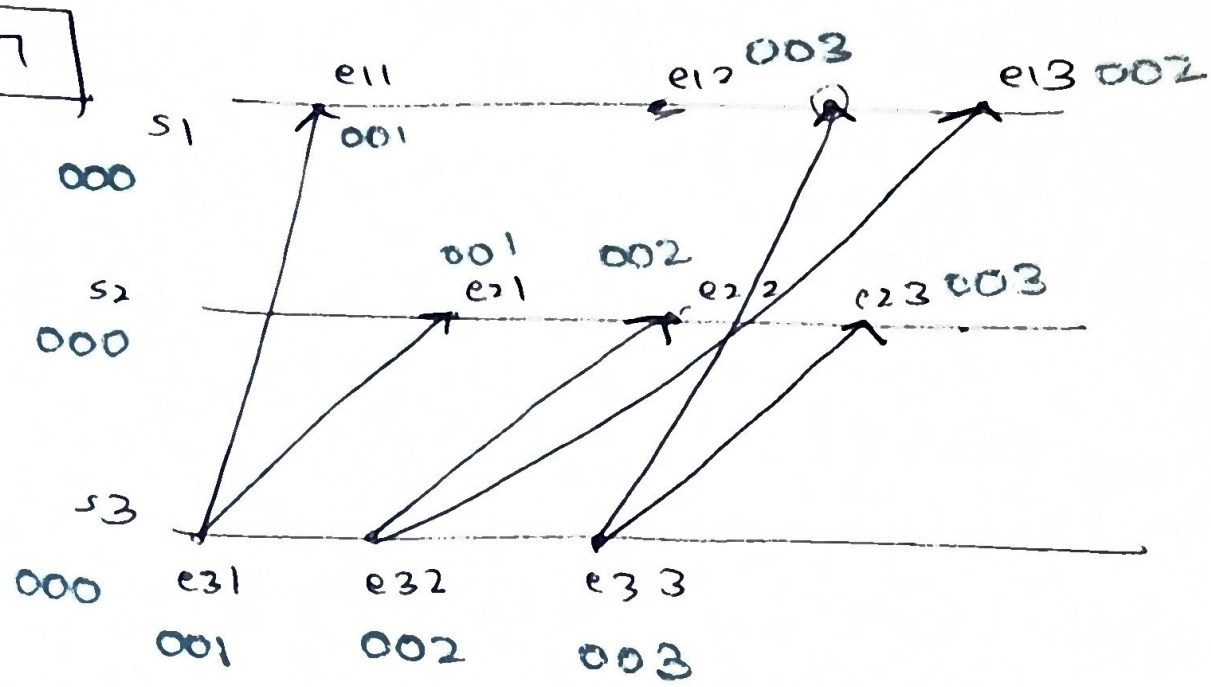
$(0,0) \geq (0,0)$ ✓

$\Rightarrow$ deliver

$\Rightarrow e_{32}\,B = max(020, 030) = 030$

Example 7

(Type = case3)



e21 and e11 are just regular updates

$e_{11} = max(001, 000) = 001$

$e_{21} = max(001, 000) = 001$

Same for e_{22} and e_{23}

$e_{22} = max(002, 001) = 002$

$e_{23} = max(003, 002) = 003$

(i) For e_{12} $e_1^{'s} = [001]$

$tm = [003]$

$c_1'[3] == tm[3] - 1$ X

place (3, 003) in the buffer of S_1

(ii) For e_{13} $c_1 = [001]$

$tm = [002]$

$c_1[3] == tm[3] - 1$ ✓

$c_1[1,2] \geq tm[1,2]$ ✓

⇒ deliver ⇒ $e_{13} = max(002, 001) = 002$

(iii) From the buffer e_{12} B

$c_1 = [002]$

$tm = [003]$

$c_1[3] == tm[3] - 1$ ✓

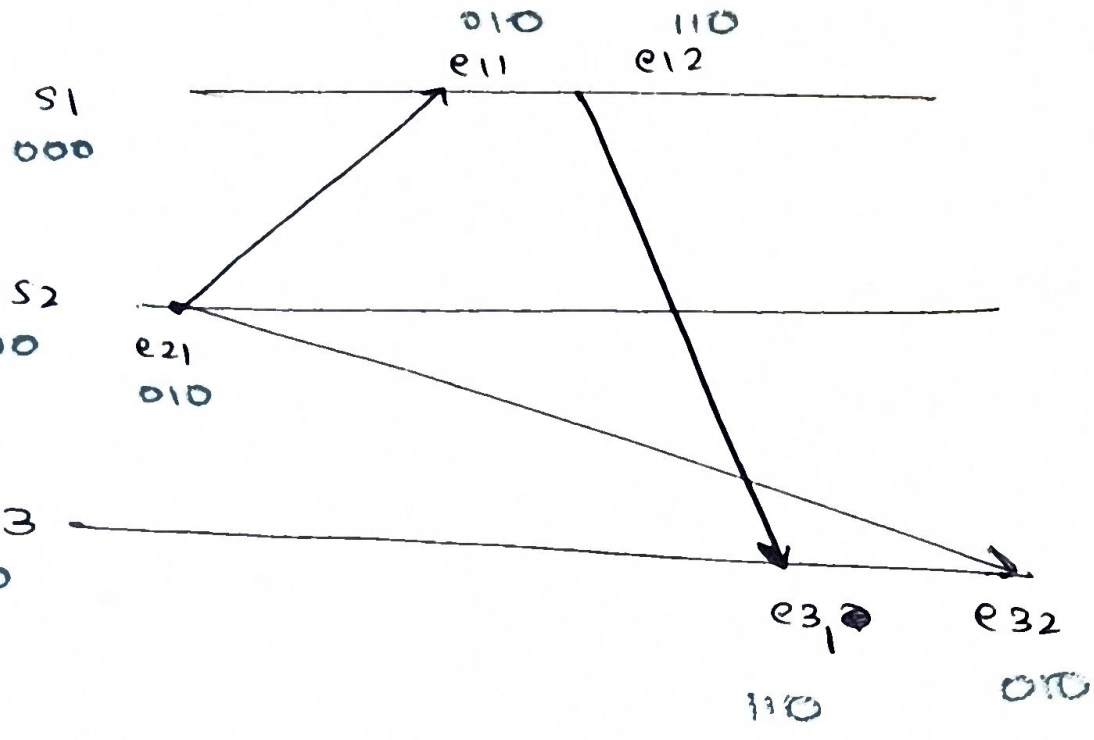
$c_1[1,2] \geq tm[1,2]$ ✓

⇒ deliver ⇒ $e_{12} = max(002, 003) = 003$

Example 8

Type: Case 5

Missed from other sender



(i) For e_{11} $c_1 = [000]$
$tm = [010]$

$c_1[0] == tm[2] - 1$ ✓

$c_1[1,3] \geq tm[1,3]$ ✓

$e_{11} = 010$

(ii) For $e_{12} = 110$

(iii) For e_{31} $c_3 = [000]$
$tm = [110]$

$c_3^{r,s}[1] == tm[1] - 1$ ✓

$c_1[2,3] \geq tm[2,3]$

$(0,0) \geq (1,0)$ ✗ → S_3 has missed a msg from a system that is not the current sender.

push $(1, 110)$ onto buffer

(iv) For e_{32} $c_3' = [000]$
$tm = [010]$

$c_3^{r,s}[2] == tm[2] - 1$ ✓ ⇒ $e_{31} = max(000, 010) = 010$

$c_3[1,3] \geq tm[1,3]$ ✓

(v) Now e_{31} B $c_3 = [010]$ ⇒ $e_{31} = max(010, 110)$
$tm = [110]$ $= 110$

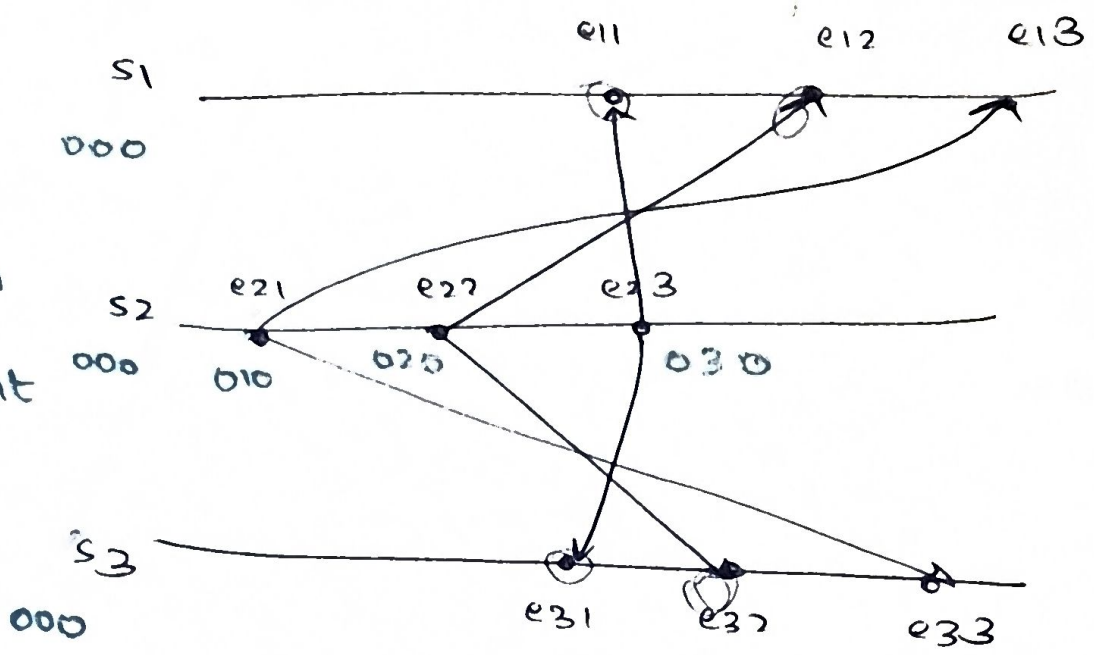
$c_3^{r}[1] == tm[1] - 1$ ✓

$c_3[2,3] \geq tm[2,3]$ $(1,0) \geq (1,0)$ ✓

Example 9

Type : Case 6

multiple elements in buffers of different systems



In this case e31 and e32 will fail since the message from e21 has not reached e33

push onto buffer of S3

(2,030)	(2,020)

e11 and e12 will fail because the message from e21 has not reached e13

push onto buffer of S1

(2,030)	(2,020)	

process e13 ⇒ 010

and e33 ⇒ 010

then dequeue each process from the queues of S1 & S3

* Causal Ordering of Messages - Multicast

→ uses an extra data structure : Vector Vi

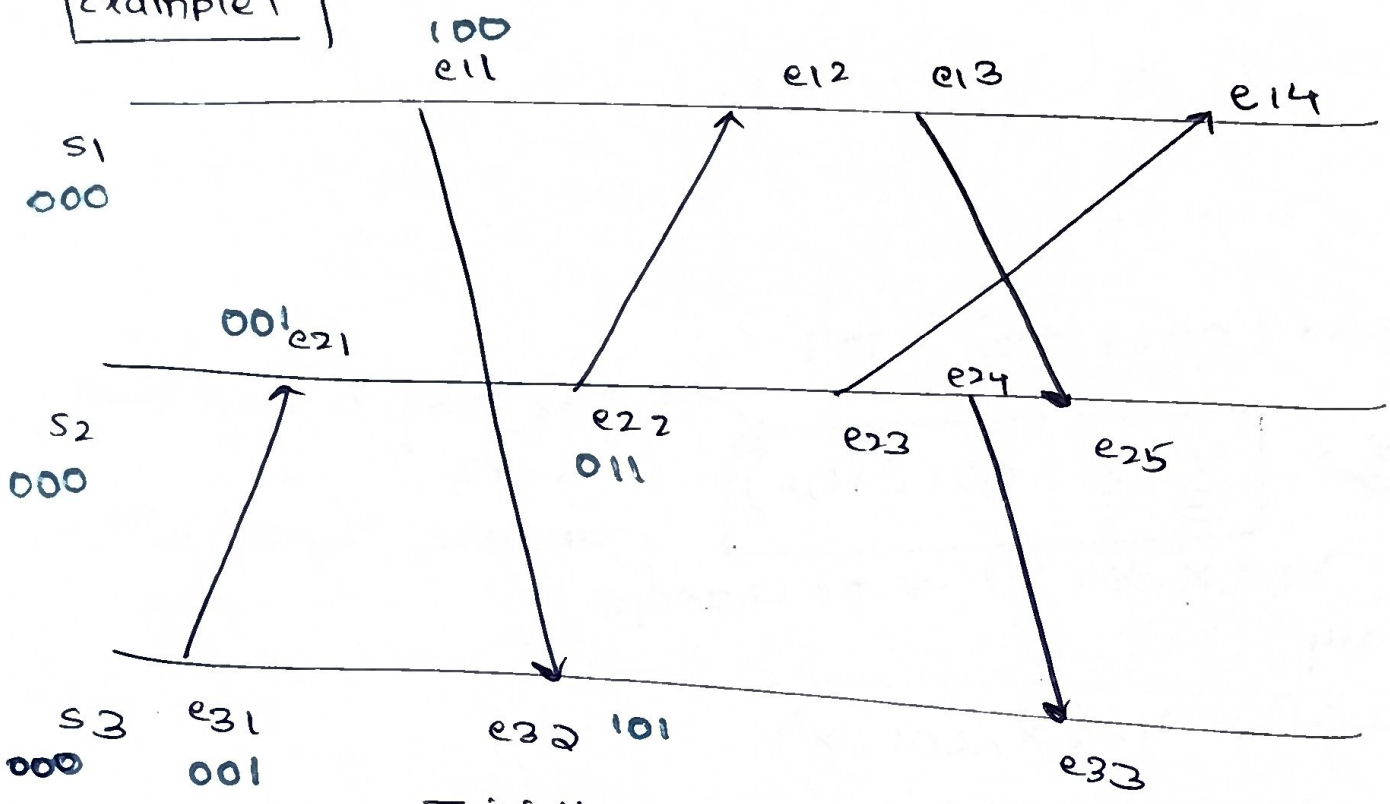
→ Algorithm for send

update clock Ci

send (Ci, Vi) to j

update Vi

Example 1



① at e31

(i) Inirially $C_3(0,0,0)$ $V_3(X,X,X)$

(ii) update clock : $C_3(0,0,1)$

Send (iii) send $C_3(0,0,1)$ $V_3(X,X,X)$ to S_2

(iv) update V_3 : $V_3(X, 001, X)$ → sent to S_2, update second pos

sent to S_2

② at e21

(i) Inirially $C_2(0,0,0)$, $V_2(X,X,X)$

recv (iii) Receive $C_3(0,0,1)$, $V_3(X,X,X)$ update received

(iii) $e_{21} = \max(000, 001) = 001$

update $V_2 = (X, 001, X)$ → clock value at receiver ind

(place received clock value)

own

ump

(X,X,X) &

③ at e11 : (i) Initially

$c_1(0,0,0)$ $v_1(X, X, X)$

(ii) update clock $c_1(1,0,0)$

send (iii) send c, v to e32

(iv) update $V_1 : (X, X, 100)$ → sent to S_3

The msg is sent to S_3, update 3rd pos

compare with

④ at e32

(i) Currently

$c_3(001)$ $V_3(X, 001, X)$

recv (ii) receive c, v from e11

(iii) $e_{32} = \max(001, 100) = 101$

(iv) update V_3 : $V_3(X, 001, 101)$

The msg is sent from 1 → 3, update V_3 at 3rd pos

compare with $V_1(X, X, X)$ ⇒ no change

⑤ at e22 (i) Currently

$c_2(001)$ $V_2(X, 001, X)$

send (ii) update clock : $c_2(011)$

(iii) send c_2, V_2 to S_1

(iv) update V_2 $(011, 001, X)$ → The msg is sent to from 2 → 1, update first pos

⑥ at e12 : (i) Currently

$c_1(100)$ $V_1(X, X, 100)$

recv (ii) receive c_2, V_2 from S_2

(iii) $e_{12} = \max(100, 011) = 111$

The message is sent from 2 → 1, update 1

(iv) update V_1 : ~~$V_1(111, X, 100)$~~

V_2 received : (X, 001, X)

update V_1 (111, 001, 100)

remember to add clock value, received V_1 old current V

The msg, was sent from $S_2 \rightarrow S_1$ → update S_1 with clock value

$S_2 \rightarrow S_4$ update pos 1

(7) at e23

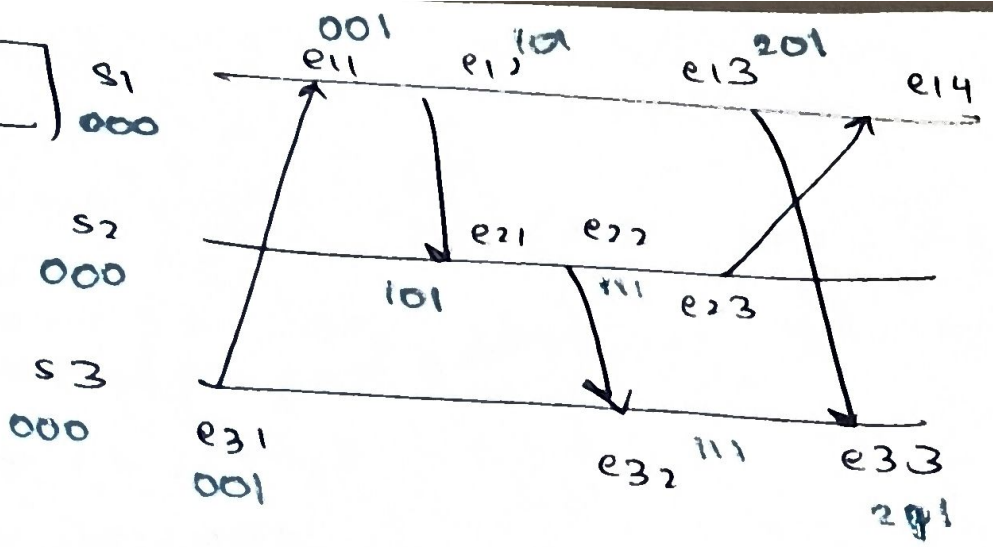
$C_2(011)$, V_2 (011, 001, X)

$C_2(021)$, V_2 (021, 001, X)

⋮

continue

Example2



@ e31

$C_3(0,0,0)$ $V_3(X,X,X)$

update C,V $C_3(0,0,1)$, $V_3(001,X,X)$

$S_3 \rightarrow S_1$

send

@ e11

$C_1(0,0,0)$ $V_1(X,X,X)$

receive here

recv

$C_1(0,0,1)$ $V_1(001,X,X)$

$S_3 \rightarrow S_1$

@ e12

$C_1(0,0,1)$ $V_1(001,X,X)$

send

update C,V $C_1(1,0,1)$ $V_1(001,101,X)$

$S_1 \rightarrow S_2$

@ e21

$C_2(0,0,0)$ $V_2(X,X,X)$

recv

receive here

$C_2(1,0,1)$ $V_2(001,101,X)$

$S_1 \rightarrow S_2$)

@ e13

$C_1(101)$ $V_1(001,101,X)$

$C_1(201)$ $V_1(001,101,201)$

send

$S_1 \rightarrow S_3$

@ e33 $C_3(\quad)$ $V_3(001, \quad, \quad)$

→ receive here

recv $C_3(\quad)$ $V_3(001, 101, \quad)$ $S_1 \rightarrow S_3$

@ e22 $C_2(101)$ $V_2(001, 101, X)$

send $C_2(\quad)$ $V_2(001, 101, \quad)$ $S_2 \rightarrow S_3$

@ e32 $C_2(001)$ $V_3(001, 101, 201)$

recv → receive here

$C_2(\quad)$ $V_3(001, 101, \quad)$ $S_2 \rightarrow S_3$

@e23 $C_2(1,1,1)$ $V_2(001, 101, 111)$

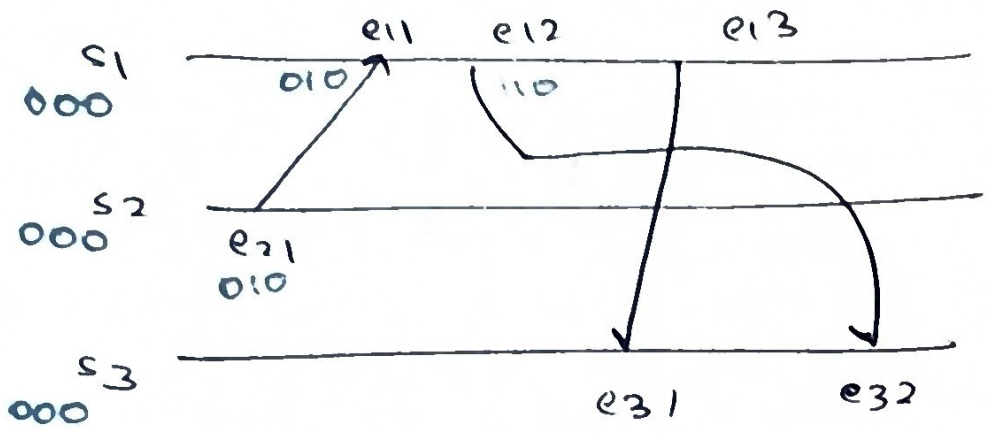
send $C_2(1,2,1)$ $V_2(12, 101, 111)$ $S_2 \rightarrow S_1$

@ e14 $C_1(201)$ $V_1(001, 101, 201)$

recv → receive here

$C_1(\quad)$ $V_1(\quad, 101, \quad)$ $S_2 \rightarrow S_1$

Example 3 – with conflict



@e21

$C_2(0,0,0)$ $V_2(X,X,X)$

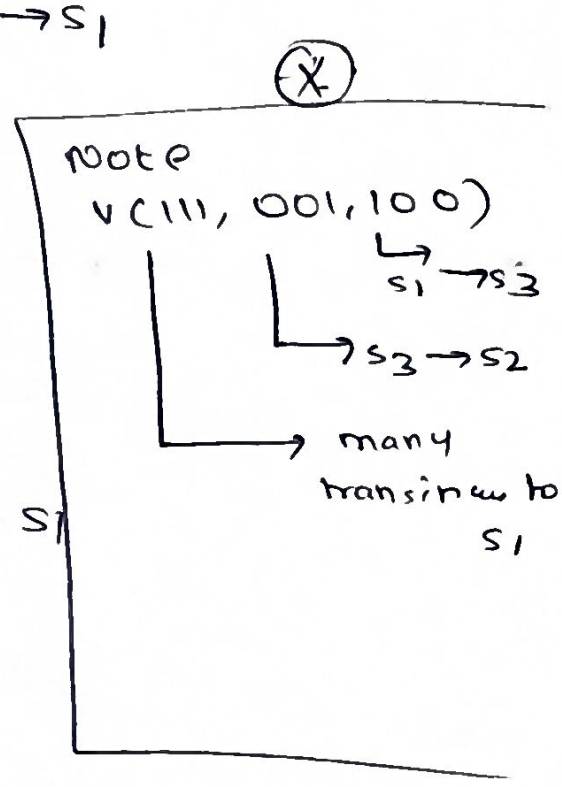
send $C_2(0,1,0)$ $V_2(010,X,X)$

@ e11 $C_1(0,0,0)$ $V_1(X,X,X)$

→ received here

recv $C_1(010)$ $V_1(010,X,X)$

$S_2 \rightarrow S_1$



$S_2 \rightarrow S_1$

@ e12 $C_1(010)$ $V_1(010,X,X)$

$C_1(110)$ $V_1(010,X,110)$

$S_1 \rightarrow S_3$

send → 2

@e32 ~~$C_3(000)$ $V_3(X,X,X)$~~

~~→ received here~~

~~recv~~

~~$C_3(1$~~ @e13 $C_1(110)$ $V_1(016,X,110)$ → 4

$C_1(216)$ $V_1(010,X,210)$ $S_1 \rightarrow S_3$

@ e31 $C_3(000)$ $V_3(XXX)$ → The receiver says nothing is received

→ receive here

but the received $V_1(010,X,110)$ ~~has~~ sent to S_3

∴ push 210, (010, X, 110) onto the buffer

now @e32 :

c_3 (000) v_3 (XXX)

↳ recv from 2 here

c_3 (110) | v_3 (010 X , 110) | $s_1 \rightarrow s_3$

now reattempt @e31 B

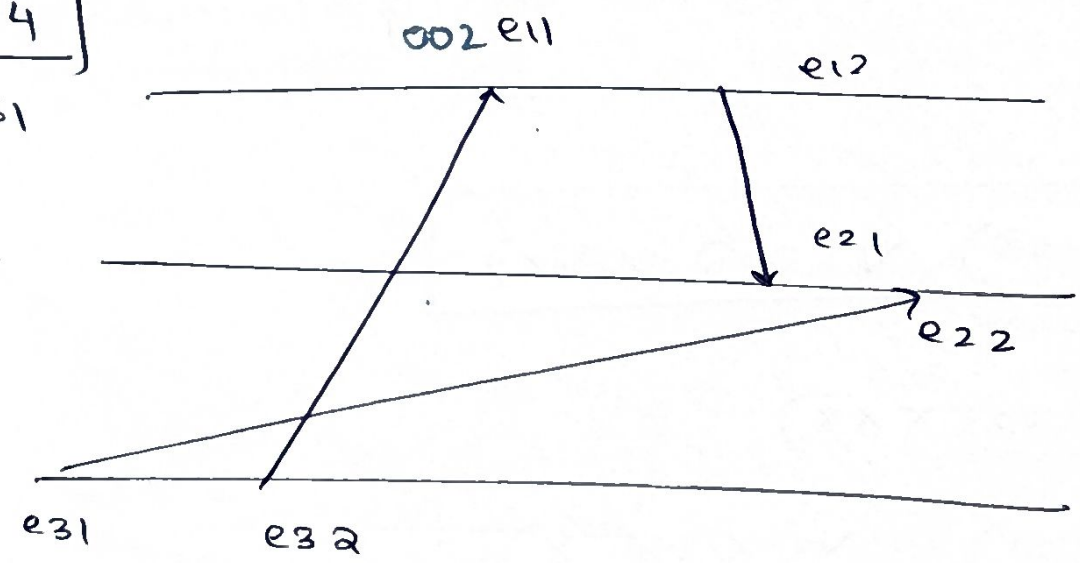
c_3 (110) v_3 (010, X, 110)

↳ receives from (Y) (from buffer)

$s_1 \rightarrow s_3$

c_3 (210), | v_3 (010, X, 210) |

Example



@ e31 : c_3(000) v_3(X X X)

c_3(001) | v_3(X,001, X) | $s_3 \rightarrow s_2$ → (1)

@ e32 c_3(001) v_3(X, 001, X)

c_3(002) v_3(002, 001, X) $s_3 \rightarrow s_1$

@ e_{11} $c(000)$ $v(xxx)$

→ received here

$s_3 \rightarrow s_1$

$c(002)$ $v(002, 001, x)$

@ e_{12}

$c(002)$ $v(002, 001, x)$

$c(102)$ $v(002, 102, x)$ $s_1 \rightarrow s_2$ → ②

@ e_{21} $c_2(000)$ $v_2(xxx)$

→ received here

Ther receiver says nothing is received by S_2 so far but $v(002, 001, x)$ ⇒ S_3 has sent a message to S_2 that has not reached.

⇒ buffer $c(102), v(002, 001, x)$

@ e_{22} $c_a(000)$ $v_2(xxx)$

from ① $c_a(001)$ $v_2(x, 001, x)$ $s_3 \rightarrow s_2$

now reattempt for e_{21} B

$c_2(001)$ $v_2(x, 001, x)$

from ② in buffer $s_3 \rightarrow s_2$

$c_2(102)$ $v_2(002, 102, x)$

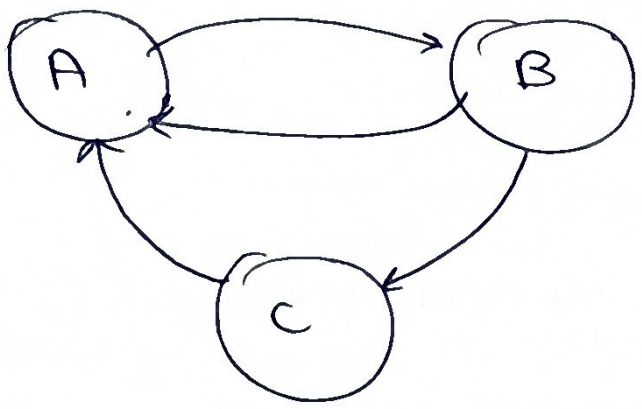
* Global States → consistent
  → strongly consistent
  → inconsistent

Consistent state – some sends are recorded, but are not received - they are still in transit - will be recorded in the next global state

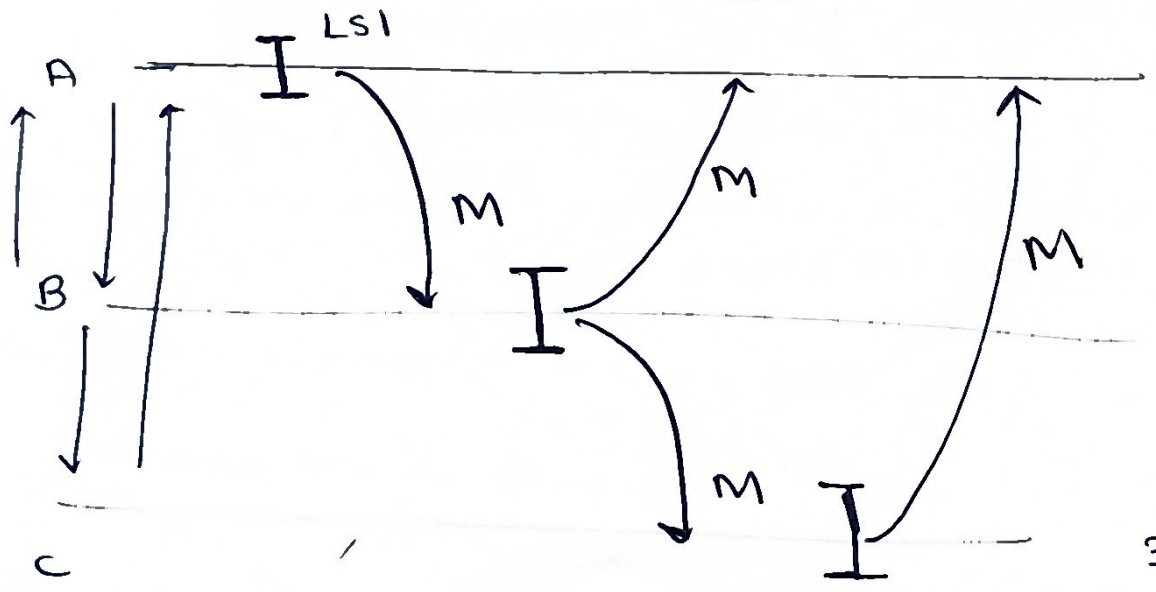
Strongly consistent – every send has a corresponding rec

* Global State Recording Algorithm (GSRA) - Chandy Lamport Algorithm

Consider the following topology



assume A starts the algorithm

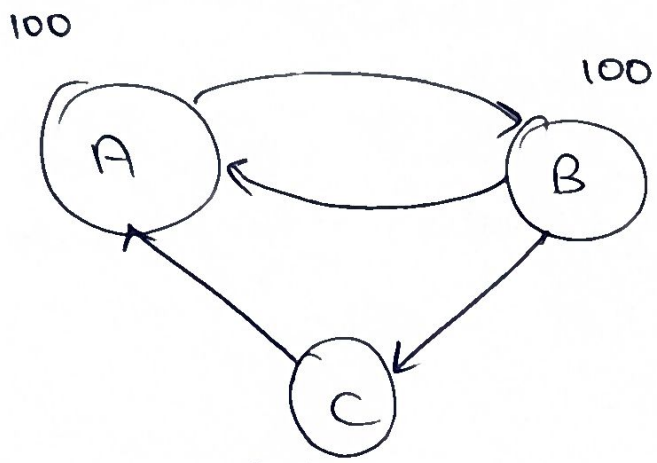


1. A records its local state
2. Immediately send a marker to its connections (here : B)
3. B takes a snapshot
4. Immediately send a marker to its connections

5. Now C takes a snapshot and sends a marker to its connection A.

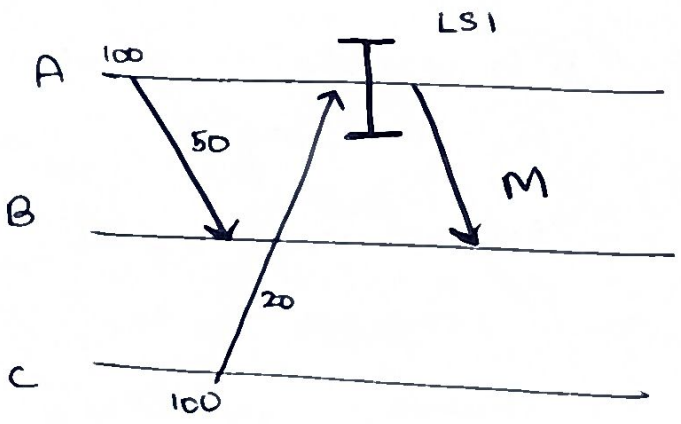
6. The algorithm ends when the initiator has received a marker from all the other connected states

Now, execution of the algorithm with values associated with it.



Start:

1. A sends 50 to B
2. C sends 20 to A
3. A takes a snapshot ⇒ LS1 = 50
4. Now it has to send markers to its connections (B)



current state

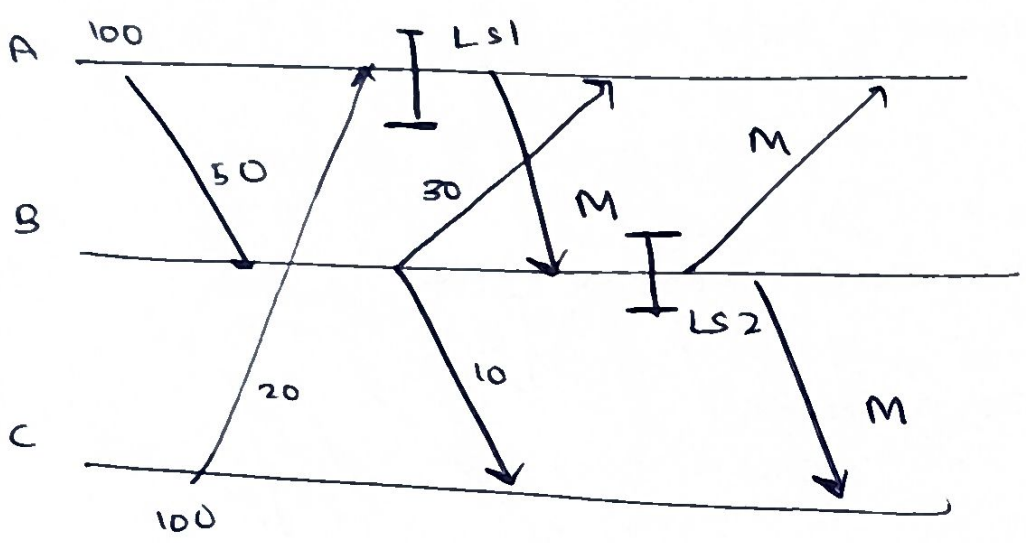
5. Now, before B receives M, it sends 30 to A and 10 to C.

(A)

6. Now it has received a marker

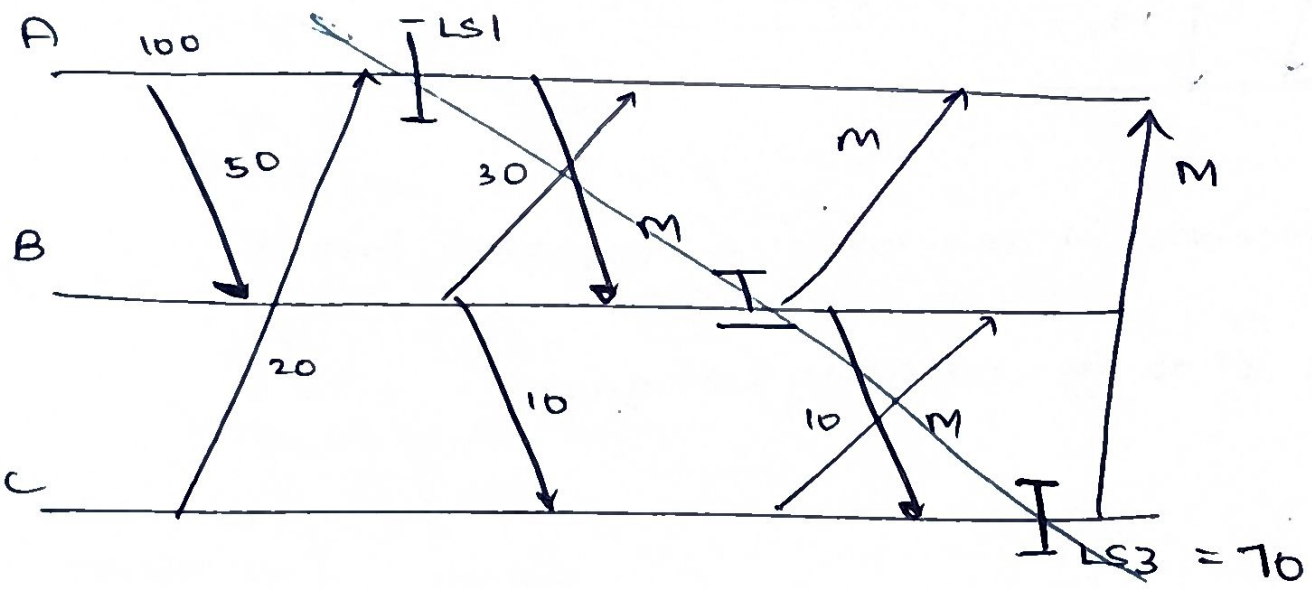
7. B Takes a snapshot LS2 : B = 60

8. Now B sends markers to its connections B & A



9. Before C receives its marker it sends 10 to B.

10. Then it takes a snapshot and sends a marker to its connections (A)

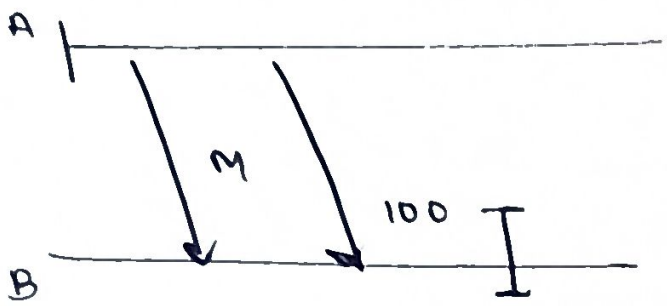


11. A has received markers from all its connected processes.

The algorithm is complete

* ~~Examples for Snapshot consistent~~,

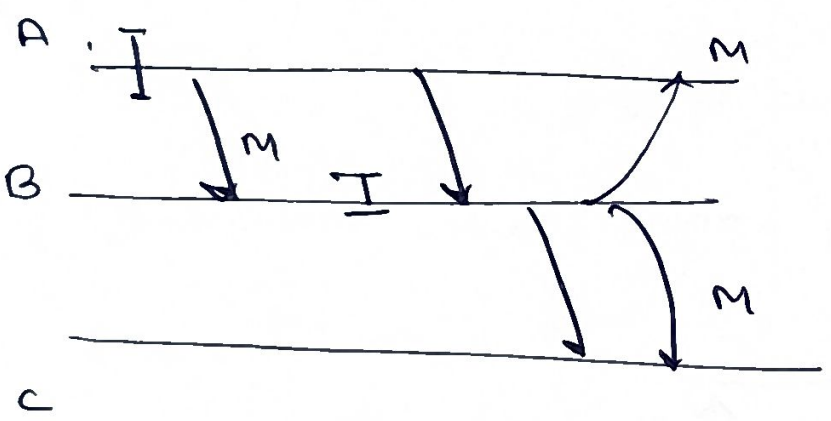
* Avoiding Inconsistency in GSRA Algorithm



→ B receives a marker, but doesn't take a snapshot (i.e it postpones) until it receives from A.

→ Hence, in this case, the global state shows that B has received, but the send is not recorded

→ This is inconsistent



→ In this case B receives M and takes a snapshot, but it postpones the send of M to its outgoing channels

Basically

(i) receive M

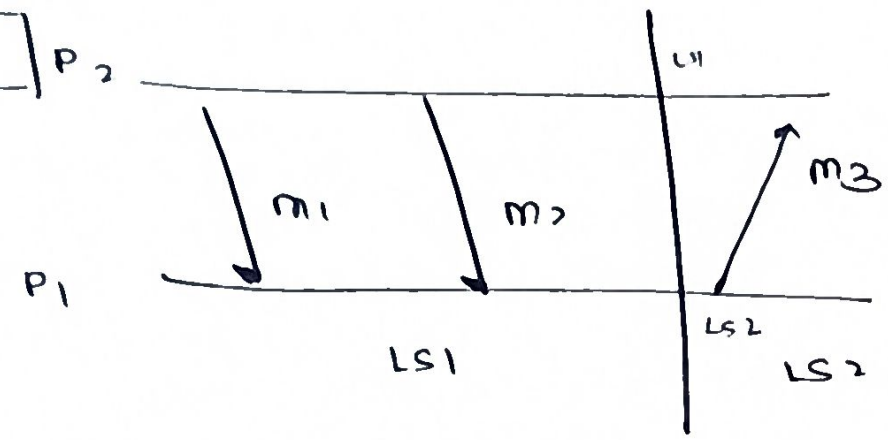
(ii) Take snapshot

(iii) send M

⇒ There should be no other sends / receives after an M is received and until all Ms are sent out
a snapshot is taken

* Examples for strongly consistent, concistent and inconsistent

Example1



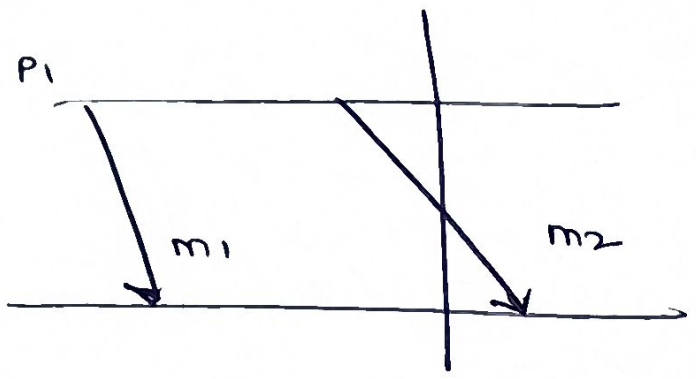
GS = LS1 ∪ LS2

LS1 = { send(m1) , send(m2) }

LS2 = { rec(m1), rec(m2) }

=> strongly consistent

Example2

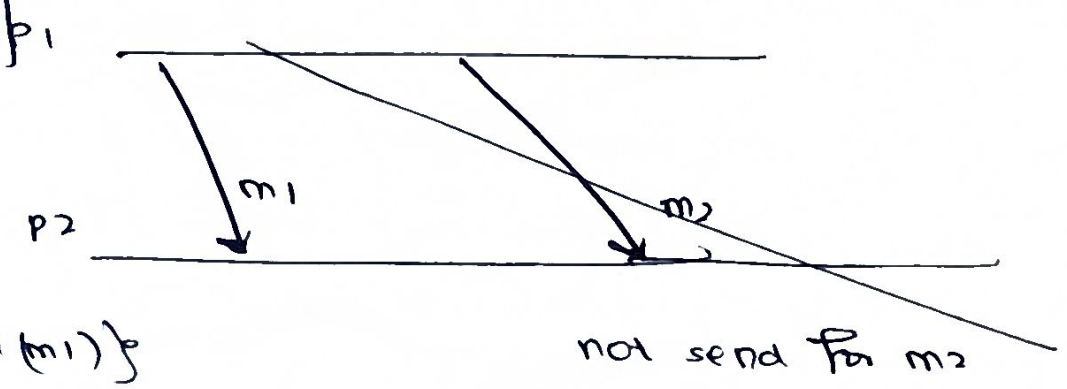


LS1 = { send(m1), send m2 }

LS2 = { rec(m1 }

=> consistent

Example 3



LS1 = { send(m1) }

LS2 = { rec(m1), rec(m2) }

not send for m2

=> Inconsistent