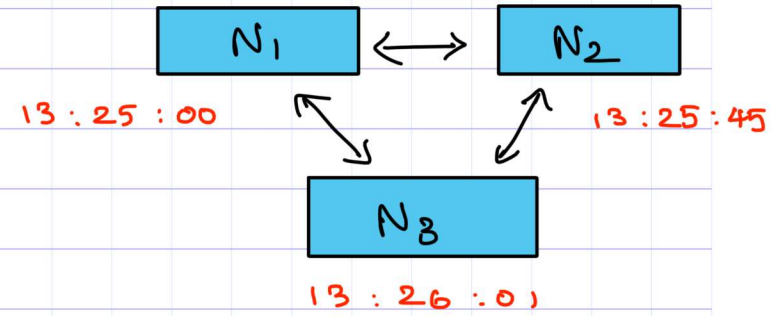
Synchronization (Absence of global clock)

$N_3 \rightarrow N_1, N_2$: 01

$N_1 \rightarrow N_3$: 01

$N_2 \rightarrow N_3$: 46



$N_2 \rightarrow N_1, N_3$: 47

$N_1 \rightarrow N_2$: 03

$N_3 \rightarrow N_2$: 04

Ordering of msgs w/ clock order (of N_1, say) leads to wrong ordering!

- Clock syncing isn't practical.
- Physical clocks aren't possible.
- Geographically distributed nodes ⇒ different timezones
- Cannot have a common clock.

Assumptions:

→ Comm. channel is error free.

→ Comm. channel is FIFO.

Clock syncing with message passing: [rcvd. timestamp + RTT/2] — based on assumptions

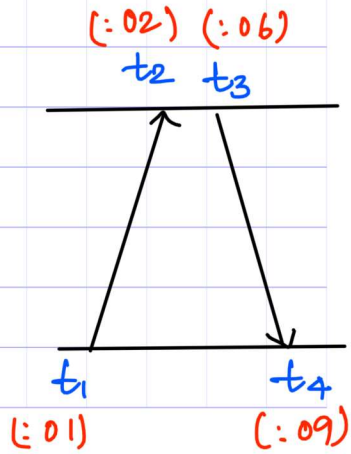
In case of delay & not FIFO:

$(t_4 - t_1) - (t_3 - t_2) = 8 - 4 = 4$

$(t_4 - t_1)$ ↓ RTT

$(t_3 - t_2)$ ↓ processing overhead

4 ↓ true RTT



true RTT/2 = 4/2 = 2

↓

almost accurate.

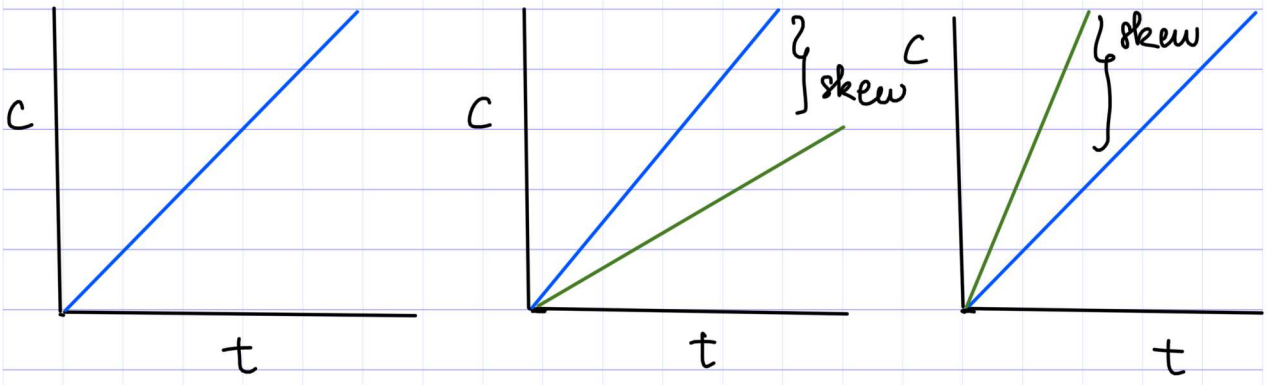
Delay is unpredictable.

∴ Cannot sync 2 physical clocks.

Clock Synchronization

Clock drift : clocks tick @ diff. rates.
quartz clock : 10^{-6} seconds
diff. of 1 sec / 11.6 days

clock skew : diff. between 2 clocks @ one point in time



$\frac{dC}{dt} = 1$

$\frac{dC}{dt} < 1$

make clock run faster till sync

$\frac{dC}{dt} > 1$

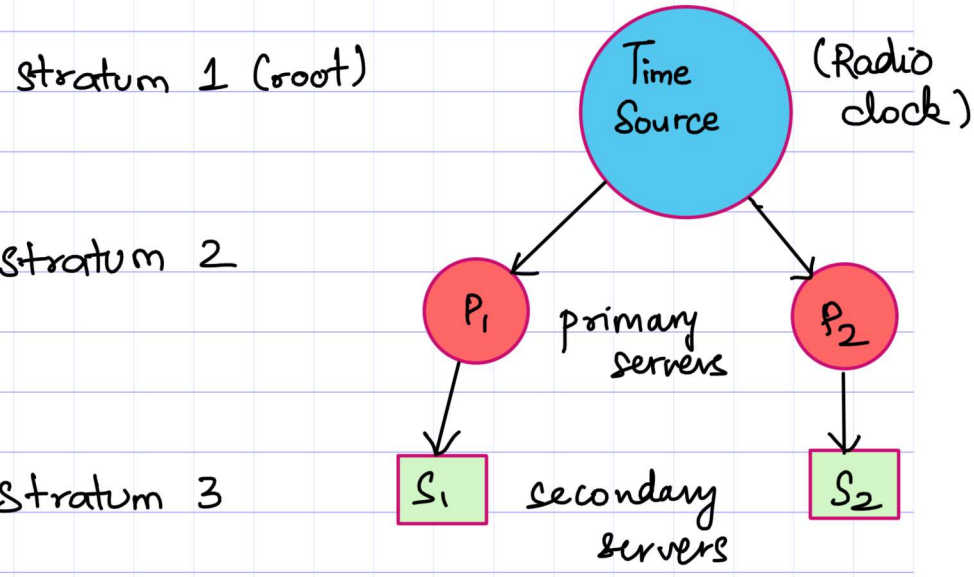
make clock run slower till sync

Mechanisms to sync :

Cristian's Algorithm ($T_{new} = T_{server} + RTT/2$)
Berkeley Algorithm (Fault tolerant avg. timestamping)
Network Time Protocol

Network Time Protocol :

Clients across Internet sync accurately to UTC despite message delays



Modes :

multicast mode - sends time to all other nodes.

procedure call mode - similar to Cristian's algorithm

symmetric mode - for master servers pair of servers exchange messages.

UDP protocol.

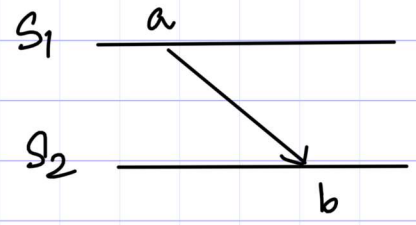
Logical Clocks (causality: as per chronology)

By Leslie Lamport

"happened before" relationship → causally affects.

- If 'a' happened before 'b', then
  $Clock_a < Clock_b$.
  $a \longrightarrow b$

- If two processes interact with 'a' in one to 'b' in another, then
  $send(m) \longrightarrow recv(m)$
  $a \longrightarrow b$.
  $C_a < C_b$

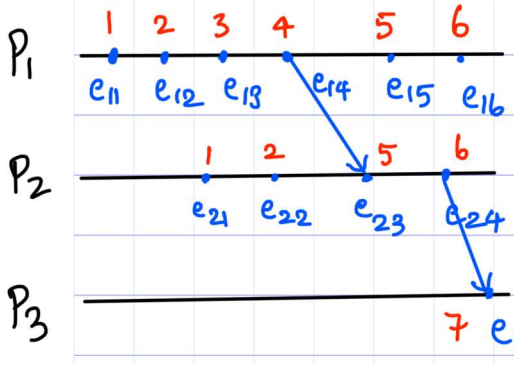


- Rules for incrementing:

$C_i = C_i + d$ for processing events ($d = 1$)

$C_{j+1} = \max(C_j, t_m) + 1$ for communication events.
↓
(message timestamp)

Partial Orders



- Transitive $e_{11} \longrightarrow e_{22}$
- Antisymmetric $e_{11} \not\longrightarrow e_{21}$
- Irreflexive $e_{11} \longrightarrow e_{11}$

Cons: if $a \rightarrow b$; $C_a \leq C_b$.

but,

(not strongly consistent) if $C_a \leq C_b$; $a \rightarrow b$? FALSE.

i.e. cannot determine chronological order given any two clock values.

Vector Clocks

- Rules for incrementing

$C_i[i] = C_i[i] + 1$ for processing events.

$\forall_k \; C_j[k] = \max(E_m[k], C_j[k] + 1)$

↓

Increment self clock on receive.

For some clock values: t_a, t_b

$t_a == t_b \Rightarrow \forall_k \; t_a[k] == t_b[k]$

$t_a != t_b \Rightarrow \exists_k \; t_a[k] != t_b[k]$.

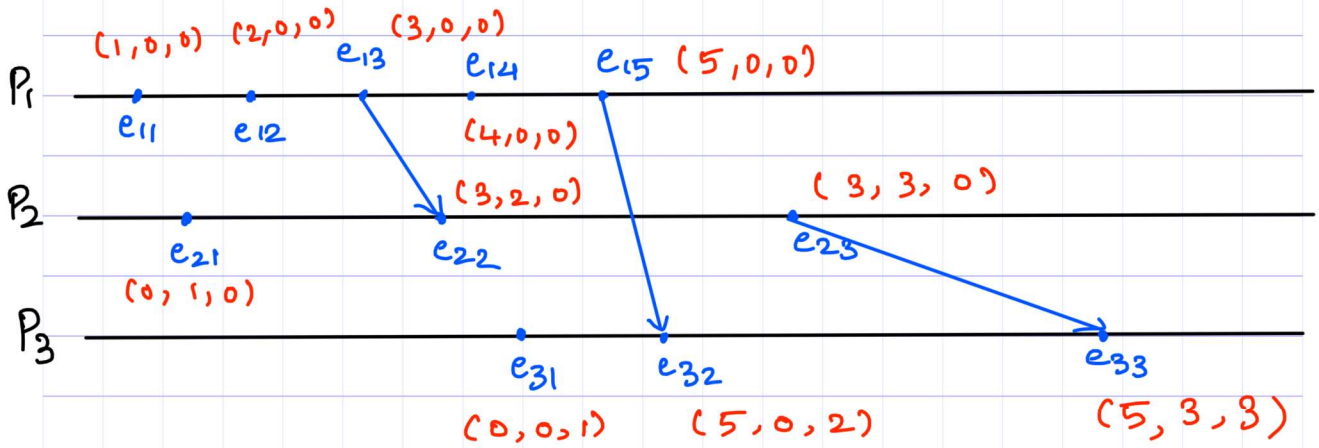
Cons:

Overlooks concurrent events.

Cannot determine which event occurred in which process, in what order.

Need more memory.

No. of processes may not be known in advance.



Applications :

- distributed debugging
- causal distributed shared memory
- global breakpoints
- consistency of checkpoints in optimistic recovery.

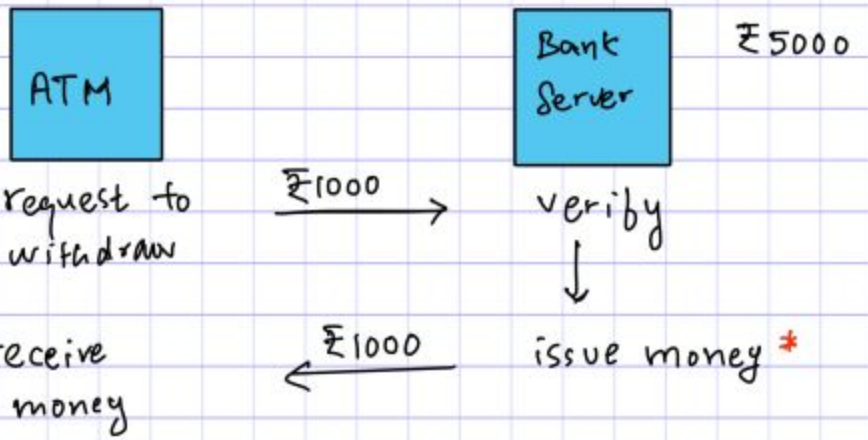
Vector clocks are strongly consistent.

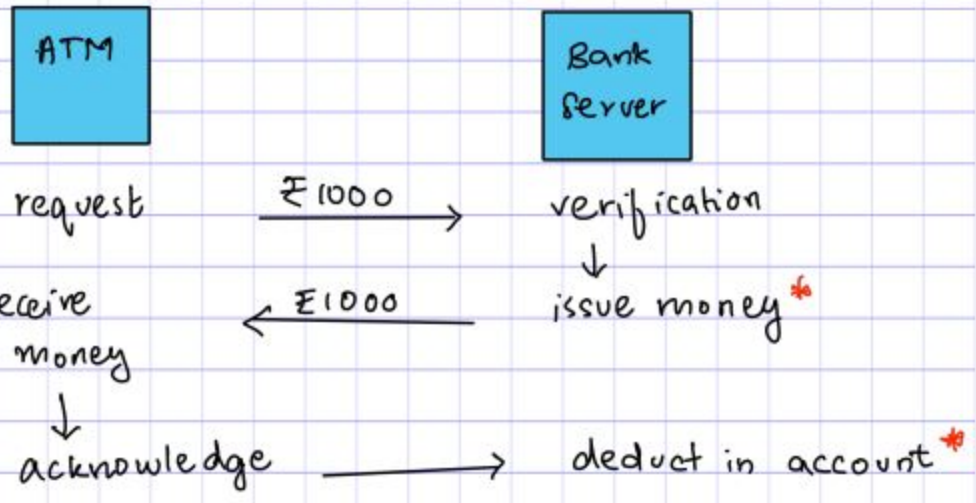
i.e. if $t_a < t_b \Rightarrow a \rightarrow b$.

↓

solves the Lamport clock problem.

Absence of shared memory problem





* If error happens in bank side, (in both cases) the account balance becomes inconsistent & bank loses money.

Causal Ordering of Messages

if $send(m_1) \rightarrow send(m_2)$
then $receive(m_1) \rightarrow receive(m_2)$.

Assumptions:

- Group communication (broadcast)
- No loss in communication channel
- Customized vector clocks
  - → Communication events only
  - → On receive, no increment

Algorithm:

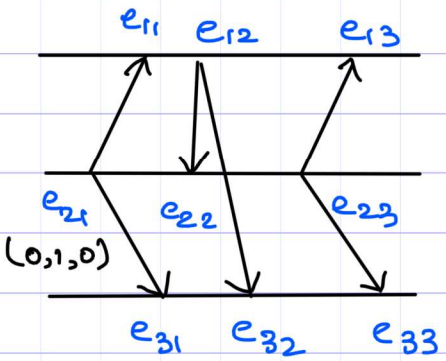
Sender's rule:
Increment i^{th} value & send timestamp.

Receiver's rule: (i - sender, j - receiver)
→ $C_j[i] == tm[i] - 1$
→ $C_j[k] >= tm[k]$, $k \neq i$
(> because send events can be concurrent)

Case (i) (Basic Case)

e_{21} : (0, 1, 0) → timestamp

e_{11} : (i = 2, j = 1) C_1 [0, 0, 0] (current clock)



(i) $C_1[2] == tm[2] - 1$? true
(ii) $C_1[1, 3] >= tm[1, 3]$? true.
∴ Accept message (∵ in causal order)
P_1 clock : (0, 1, 0)

e_{31} : $(i = 2, j = 3)$ $C_3[0,0,0]$ $tm[0,1,0]$

(i) $C_3[2] == tm[2] - 1$? true

(ii) $C_3[1,3] >= tm[1,3]$? true

∴ Accept message

P_3 clock : $(0,1,0)$

e_{12} : Sender's rule. P_1 clock : $(1,1,0)$

e_{22} : $C_2[0,1,0]$ $tm[1,1,0]$

$(i = 1, j = 2)$

(i) $C_2[1] == tm[1] - 1$? true

(ii) $C_2[2,3] >= tm[2,3]$? true

∴ Accept message

P_2 clock : $(1,1,0)$

e_{32} : $C_3[0,1,0]$ $tm[1,1,0]$

$(i = 1, j = 3)$

(i) $C_3[1] == tm[1] - 1$? true

(ii) $C_3[2,3] >= tm[2,3]$? true

∴ Accept message

P_3 clock : $(1,1,0)$

e_{23} : Sender's rule. P_2 clock : $(1,2,0)$

e_{13} : $C_1[1,1,0]$ $tm[1,2,0]$

$(i = 2, j = 1)$

(i) $C_1[2] == tm[2] - 1$? true

(ii) $C_1[1,3] >= tm[1,3]$? true

∴ Accept message

P_1 clock : $(1,2,0)$

e_{33} : $C_3[1,1,0]$ $tm[1,2,0]$

$(i=2, j=3)$

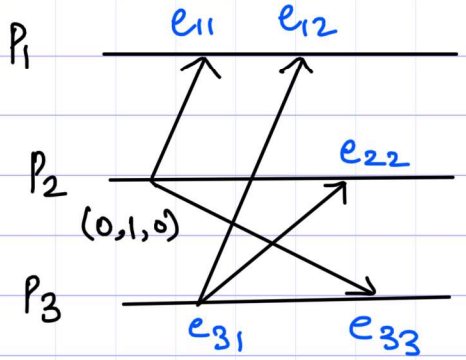
(i) $C_3[2] == tm[2]-1$? true

(ii) $C_3[1,3] >= tm[1,3]$? true

$\therefore$ Accept message

P_3 clock : $(1,2,0)$

Case (ii)



e_{11} : $C_1(0,0,0)$ $tm(0,1,0)$

$(i=2, j=1)$

(i) $C_1[2] == tm[2]-1$?

true

(ii) $C_1[1,3] >= tm[1,3]$? true

$\therefore$ Accept message

P_1 clock : $(0,1,0)$

e_{31} : Sender's rule . P_3 clock : $(0,0,1)$

e_{33} : $C_3(0,0,1)$ $tm(0,1,0)$

$(i=2, j=3)$

(i) $C_3[2] == tm[2]-1$? true

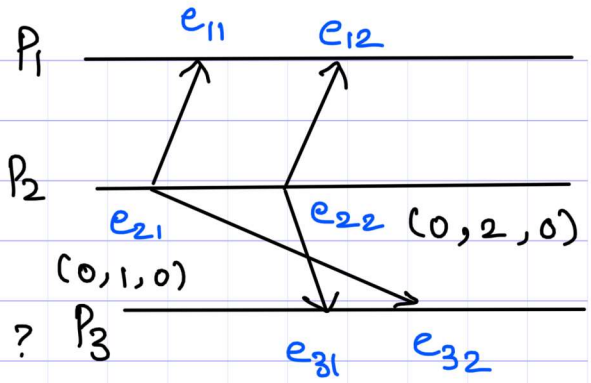
(ii) $C_3[1,3] >= tm[1,3]$? true

$\therefore$ Accept message.

P_3 clock : $(0,1,1)$

Similarly, e_{12} & e_{22} are accepted.

Case (iii) (not in causal order)



e_{11} : $C_1(0,0,0)$ $tm(0,1,0)$
$(i = 2, j = 1)$

(i) $C_1[2] == tm[2] - 1$? true

(ii) $C_1[1,3] >= tm[1,3]$? true

∴ Accept message

P_1 clock : (0,1,0)

→ 2nd msg delivered first!

e_{31} : $C_3(0,0,0)$ $tm(0,2,0)$
$(i = 2, j = 3)$

(i) $C_3[2] == tm[2] - 1$? false

There is a gap! P_3 missed a message from P_2.

∴ Do not accept the message, buffer it.

P_3 buffer : $(0,2,0)_{e_{31}}$

e_{32} : $C_3(0,0,0)$ $tm(0,1,0)$
$(i = 2, j = 3)$

(i) & (ii) are true.

∴ Accept the message.

P_3 clock : (0,1,0)

Retrieve message from buffer & retry delivery.

e_{31}' : $C_3(0,1,0)$ $tm(0,2,0)$

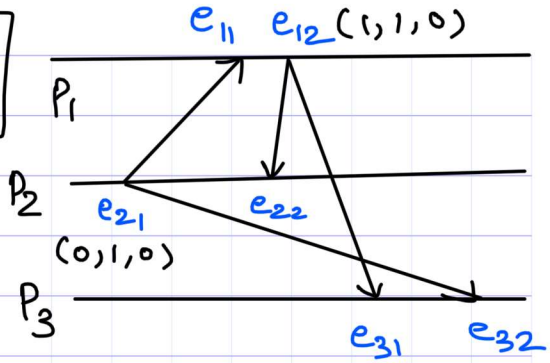
(i) & (ii) are true now.

∴ Accept message.

P_3 clock : (0,2,0).

e_{12} will be accepted.

Case (iv) [if send(m_1) → send(m_2) then recv(m_1) → recv(m_2)]



e_{11} : c_1 (0,0,0) , tm (0,1,0)
($i = 2, j = 1$)
(i) & (ii) are true
∴ Accept message
C_1 clock : (0,1,0)

e_{22} : will be accepted similarly.
C_2 clock : (1,1,0)

e_{31} : c_3 (0,0,0) tm (1,1,0) ($i = 1, j = 3$)

(i) $C_3[1] == tm[1] - 1$?
true.

[Conveys to P_3 that P_1 recvd. a msg. from P_2 which P_3 hasn't yet recvd.]

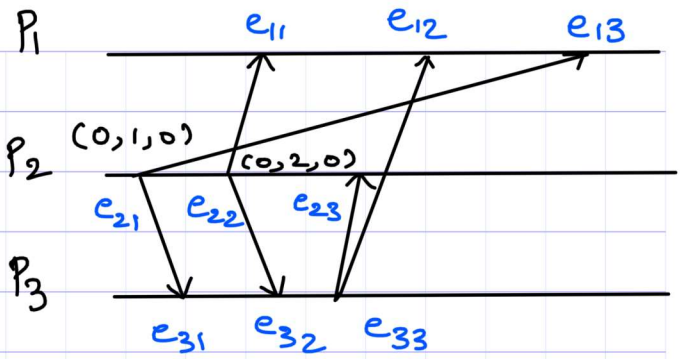
(ii) $C_3[2,3] >= tm[2,3]$? false.
Buffer e_{31} in P_3. Do not accept.

e_{32} : c_3 (0,0,0) tm (0,1,0)
($i = 2, j = 3$)
(i) & (ii) are true.
∴ Accept message.
P_3 clock : (0,1,0)

Attempt redelivery.

e_{31}' : c_3 (0,1,0) tm (1,1,0)
($i = 1, j = 3$)
(i) & (ii) are true.
∴ Accept message.
P_3 clock : (1,1,0)

Case (v)



e_{31} : C_3 (0,0,0) tm (0,1,0) ($i = 2, j = 3$)

(i) & (ii) are true.

∴ Accept message.

C_3 clock : (0,1,0)

e_{32} : C_3 (0,1,0) tm (0,2,0) ($i = 2, j = 3$)

(i) & (ii) are true.

∴ Accept message.

C_3 clock : (0,2,0)

e_{33} : Sender's rule. C_3 clock : (0,2,1)

e_{23} : will be accepted.

C_2 clock : (0,2,1)

e_{11} : C_1 (0,0,0) tm (0,2,0) ($i = 3, j = 1$)

(i) false.

Buffer the message.

P_1 buffer : $(0,2,0)_{e_{11}}$

e_{12} : C_1 (0,0,0) tm (0,2,1) ($i = 3, j = 1$)

(i) true

(ii) false

Buffer the message

P_1 buffer : e_{11}, $(0,2,1)_{e_{12}}$

e_{13} : $C_1(0,0,0)$ $tm(0,1,0)$ $(i=2, j=1)$

Rule (i) & (ii) are true.

∴ Accept message.

C_1 clock : $(0,1,0)$.

Attempt redelivery.

e_{11}' : $C_1(0,1,0)$ $tm(0,2,0)$

(i) & (ii) are true.

∴ Accept message.

C_1 clock : $(0,2,0)$.

e_{12}' : C_1 $(0,2,0)$ $tm(0,2,1)$

(i) & (ii) are true.

∴ Accept message.

C_1 clock : $(0,2,1)$

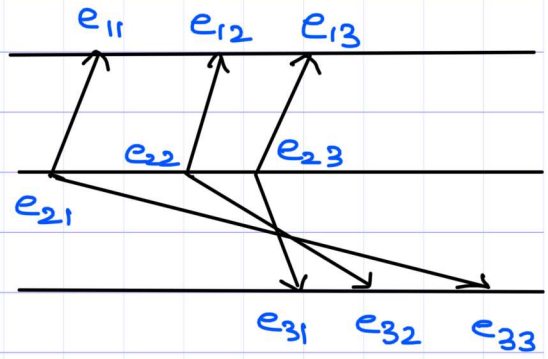
Case (vi)

e_{31} : $C_3(0,0,0)$

$tm(0,3,0)$

$(i=2, j=3)$

(i) false



Buffer e_{31} in P_3.

P_3 Buffer : $(0,3,0)e_{31}$

e_{32} : $C_3(0,0,0)$ $tm(0,2,0)$ $(i=2, j=3)$

(i) false.

P_3 Buffer : $e_{31}, (0,2,0)e_{32}$

e_{33} : $C_3(0,0,0)$ tm $(0,1,0)$ $(i=2, j=3)$

(i) & (ii) are true.

∴ Accept message.

C_3 clock : $(0,1,0)$.

Attempt redelivery.

e_{31}' : $C_3(0,1,0)$ tm$(0,3,0)$ $(i=2, j=3)$

(i) false

Buffer e_{31}' again!

P_3 Buffer : $e_{32}, (0,3,0)_{e_{31}'}$

e_{32}' : $C_3(0,1,0)$ tm $(0,2,0)$ $(i=2, j=3)$

(i) & (ii) are true.

Accept message.

C_3 clock : $(0,2,0)$

e_{33}'' : $C_3(0,2,0)$ tm $(0,3,0)$ $(i=2, j=3)$

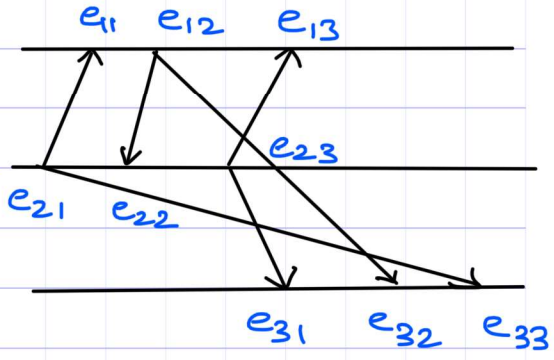
(i) & (ii) are true.

Accept message.

C_3 clock : $(0,3,0)$.

Case (vii)

(Re-entrance, but not from the same sender)



e_{12} : C_2 : $(0,1,0)$ (send)

e_{11} : Accept. C_1 : $(0,1,0)$

e_{12} : C_1 : $(1,1,0)$ (send)

e_{22} : Accept. C_2 : $(1,1,0)$

e_{23} : C_2 : (1, 2, 0) (send)

e_{13} : Accept. C_1 : (1, 2, 0)

e_{31} : C_3 (0, 0, 0) tm (1, 2, 0) (i = 2, j = 3)

(i) false.

P_3 Buffer : $(1, 2, 0)_{e_{31}}$

e_{32} : C_3 (0, 0, 0) tm (1, 1, 0) (i = 1, j = 3)

(i) true

(ii) false

P_3 Buffer : e_{31}, $(1, 1, 0)_{e_{32}}$

e_{33} : C_3 (0, 0, 0) tm (0, 1, 0) (i = 2, j = 3)

(i) & (ii) are true.

∴ Accept. P_3 clock : (0, 1, 0)

Attempt redelivery.

e_{31}' : C_3 (0, 1, 0) tm (1, 2, 0) (i = 2, j = 3)

(i) $C_3[2] == tm[2] - 1$? true.

(ii) $C_3[1,3] >= tm[1, 3]$? false.

P_3 Buffer : e_{32}, $(1, 2, 0)_{e_{31}'}$

e_{32}' : C_3 (0, 1, 0) tm (1, 1, 0) (i = 1, j = 3)

(i) $C_3[1] == tm[1] - 1$? true.

(ii) $C_3[2, 3] >= tm[2, 3]$? true.

∴ Accept message.

C_3 clock : (1, 1, 0).

e_{33}'' : C_3 (1, 1, 0) tm (1, 2, 0) (i = 2, j = 3)

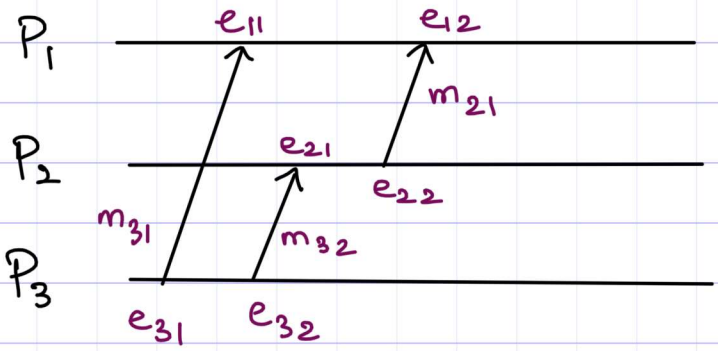
(i) & (ii) are true.

∴ Accept the message.

C_3 clock : (1, 2, 0).

If 2 send events causally affect each other, then the corresponding 2 receive events must also causally affect each other.

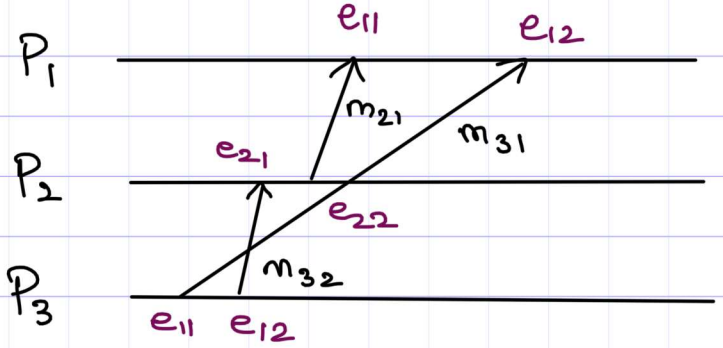
Causal Ordering of Messages - Non Broadcast



send (m_{31}) → send (m_{32})
send (m_{32}) → rec (m_{32})
rec (m_{32}) → send (m_{21})
send (m_{21}) → rec (m_{21})

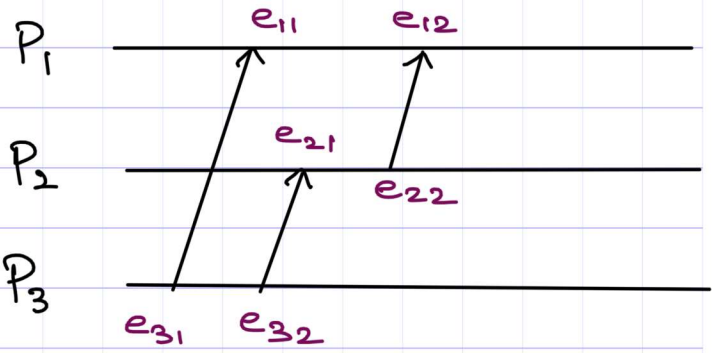
Conclusion } ⇒ rec(m_{31}) → rec(m_{21})

Delivery is causal.



Non-causal delivery.
m_{21} should be delivered after m_{31}.

Schipper - Eggli - Sandoz Protocol



e_{31}

$C(0,0,1)$

$V_3(x,x,x)$

Send t, V_3.

Update V_3.

$V_3((0,0,1), x, x)$

t - timestamp

V_m - message vector

e_{32}

$C(0,0,2)$ $V_3((0,0,1), x, x)$

Send t, V_3.

Update V_3.

$V_3((0,0,1), (0,0,2), x)$

e_{11}

$C(0,0,0)$ $V_1(x,x,x)$

$t(0,0,1)$ $V_m(x,x,x)$ ← received.

↳ Both null.

⇒ nothing has happened yet.

Accept the message.

$V_1(x, x, (0,0,1))$

copied here because message received from P_3.

Merge the vectors & copy the timestamp to the vector.

$C(0,0,1)$

e_{21}

$C(0,0,0)$ $V_2(x, x, x)$ — no conflict

$t(0,0,2)$ $V_m((0,0,1), x, x)$

$V_2((0,0,1), x, (0,0,2))$ $C(0,0,2)$

e_{22}

$C(0,1,2)$ $V_2((0,0,1), x, (0,0,2))$

Send C, V_2.

Update V_2.

$V_2((0,1,2), x, (0,0,2))$

e_{12}

$C(0,0,1)$ $V_1(x, x, (0,0,1))$

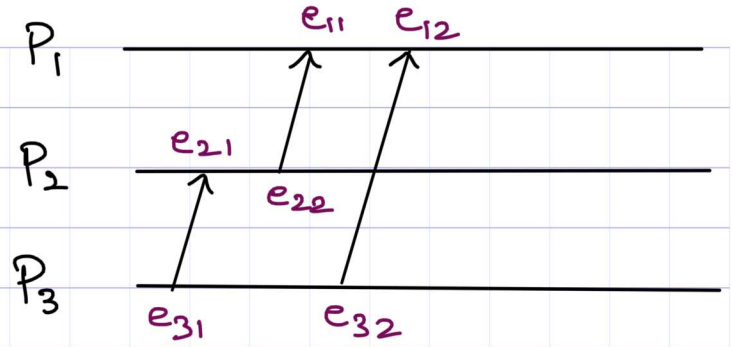
$t(0,1,2)$ $V_m((0,0,1), x, (0,0,2))$

e_{31}

Send event

$C_3 (0, 0, 1)$

$V_3 (x, x, x)$



Send C_3, V_3.

Update V_3.

$V_3 (x, (0, 0, 1), x)$.

e_{32}

Send event

$C_3 (0, 0, 2)$

$V_3 ((0, 0, 2), (0, 0, 1), x)$

Send C_3, V_3.

Update V_3.

$V_3 ((0, 0, 2), (0, 0, 1), x)$.

e_{21}

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

Receive

$C_m (0, 0, 1)$ & $V_m (x, x, x)$. ⇒ First msg. from sender.

Accept the message.

Update V_2.

$V_2 (x, x, (0, 0, 1))$

Merge Clock Values.

$C_2 (0, 0, 1)$

e_{22}

Send event

$C_2 (0, 1, 1)$

$V_2 (x, x, (0, 0, 1))$

Send C_2 & V_2.

Update V_2.

$V_2 ((0, 1, 1), x, (0, 0, 1))$

e_{11}

$C_1 (0, 0, 0)$ $V_1 (x, x, x)$

→ no conflict.

Receive

$C_m (0, 1, 1)$ $V_m ((x), x, (0, 0, 1))$

Accept the message.

Update C_1. $C_1 (0, 1, 1)$

Update V_1. $V_1 [x, (0, 1, 1), (0, 0, 1))$

↓ timestamp copied to sender's location.
+ merged V_m & V_1.

now P_1 is aware that P_3 has sent a msg. to P_2.

e_{12}

$C_1 (0, 1, 1)$ $V_1 [(x), (0, 1, 1), (0, 0, 1)]$

no conflict

Receive

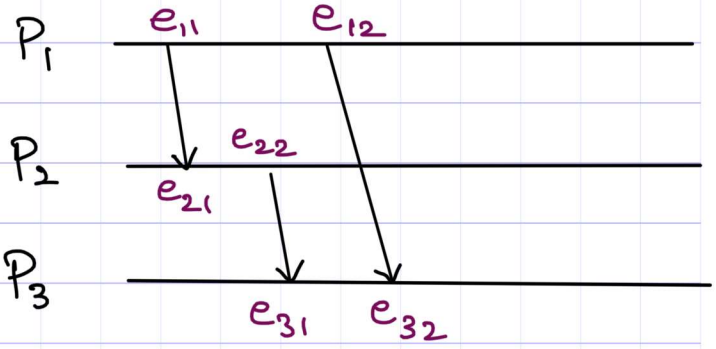
$C_m (0, 0, 2)$ $V_m [(x), (0, 0, 1), x)$

Accept the message.

Update/merge C_1 & V_1.

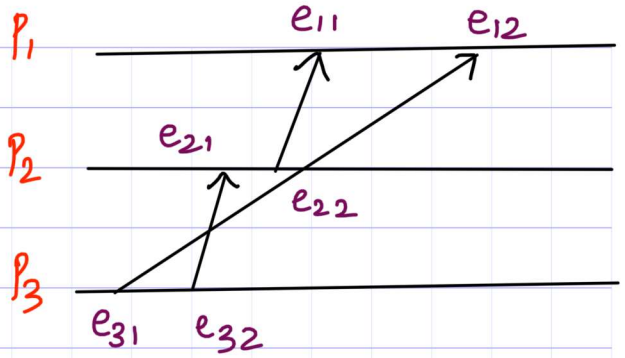
$V_1 (x, (0, 1, 1), (0, 0, 2))$

$C_1 (0, 1, 2)$



Try.

Example



e_{31}

C_3 [0 0 0]
V_3 [x x x]

send C_3 [0 0 1] ; V_3 [x x x] to e_{12}.
update C_3 [0 0 1] ; V_3 [(0 0 1) x x]

e_{32}

C_3 [0 0 1]
V_3 [(0 0 1) x x]

send C_3 [0 0 2] ; V_3 [(0 0 1) x x] to e_{21}.
update C_3 [0 0 2] ; V_3 [(0 0 1) (0 0 2) x]

e_{21}

C_2 [0 0 0] V_2 [x x x]
recv [0 0 2] ; V_m [(0 0 1) x x]
update C_2 [0 0 2] ; V_2 [(0 0 1) (0 0 2) x]

e_{22}

C_2 [0 0 2] ; V_2 [(0 0 1) (0 0 2) x]
send C_2 [0 1 2] ; V_2 [(0 0 1) (0 0 2) x]
update C_2 [0 1 2] ; V_2 [(0 1 2) (0 0 2) x]

e_{11}

C_1 [0 0 0] V_1 [x x x]
recv C_1 [0 1 2] V_m [(0 0 1) (0 0 2) x]
send C_1 [0 1 2] V_m [(0 0 1) (0 0 2) x]
to buffer.

e_{12}

C_1 [0 0 0] V_1 [x x x]

recv. [0 0 1] V_m [x x x]

update C_1 [0 0 1] V_1 [(0 0 1) x x]

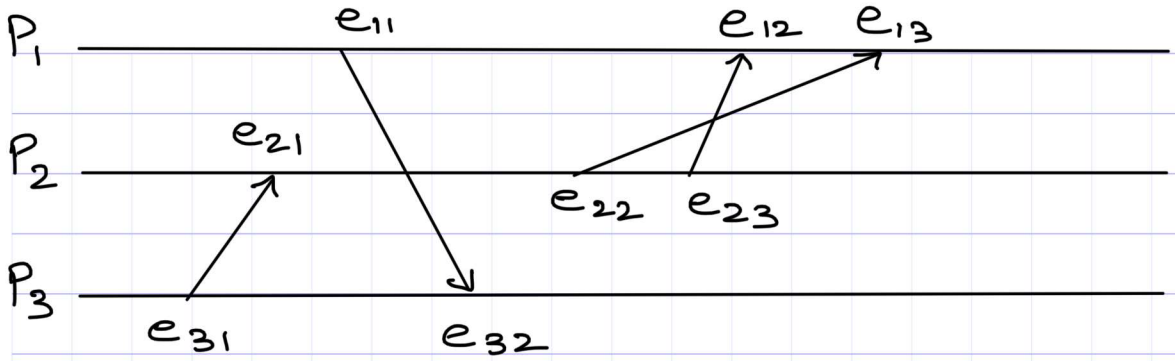
since clock C_1 is updated, check buffer.

e_{11}'

C_1 [0 0 1] V_1 [(0 0 1) x x]

recv. [0 1 2] V_m [(0 0 1) (0 0 2) x]

update C_1 [0 1 2] V_1 [(0 0 1) (0 0 2) x]



e_{31} (send)

$C_3[0,0,0]$ $V_3[x,x,x]$

$C_3[0,0,1]$ $V_3[x,x,x]$

send to e_{21}

$V_3[x, [0,0,1], x]$

e_{21} (receive)

$C_2[0,0,0]$ $V_2[x, x, x]$

→ $C_m[0,0,1]$ $V_m[x, x, x]$

unset

Accept delivery.

$C_2[0,0,1]$ $C_2[x, [0,0,1], x]$

e_{11} (send)

$C_1[0,0,0]$ $V_1[x,x,x]$

$C_1[1,0,0]$ $V_1[x,x,x]$

send to e32

$V_1[x, x, [1,0,0]]$

e_{32} (receive)

$C_3[0,0,1]$ $V_3[x,[0,0,1],x]$

$C_m[1,0,0]$ $V_m[x,x,x]$

unset

Accept delivery

$C_3[1,0,1]$

$V_3[x,[0,0,1],[1,0,0]]$

e_{22} (send)

$C_2[0,0,1]$ $V_2[x,[0,0,1],x]$

$C_2[0,1,1]$ $V_2[x,[0,0,1],x]$

send to e_{13}

$V_2[[0,1,1],[0,0,1],x]$

e_{23} (send)

$C_2[0,2,1]$ $V_2[[0,1,1],[0,0,1],x]$

send to e_{12}

$V_2[[0,2,1],[0,0,1],x]$

e_{12} (receive)

$C_1[1,0,0]$ $V_1[x,x,[1,0,0]]$

$C_m[0,2,1]$ $V_m[[0,1,1],[0,0,1],x]$

buffer it.

e_{13} (receive)

$C_1[1,0,0]$ $V_1[x,x,[1,0,0]]$

$C_m[0,1,1]$ $V_m[x,[0,0,1],x]$

accept delivery

$C_1[1,1,1]$ $V_1[[0,1,1],x,[1,0,0]]$

e_{12}' (redelivery)

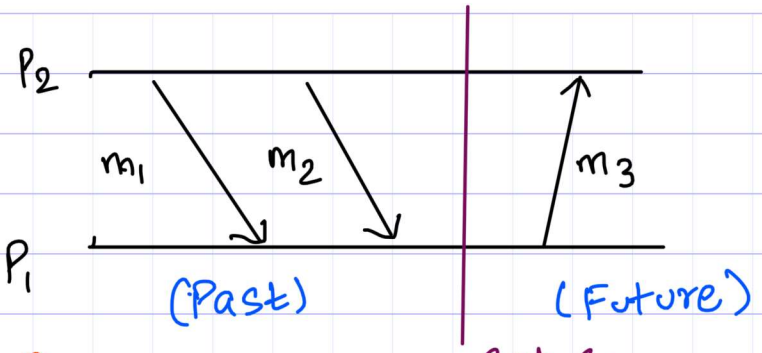
C_1 [1, 1, 1] V_1 [[0, 1, 1], x , [1, 0, 0]]

C_m [0, 2, 1] V_m [[0, 1, 1], x , x]

accept delivery

C_1 [1, 2, 1] V_1 [[0, 2, 1], x, [1, 0, 0]]

Global States



Global state = $\bigcup_i^n$ Local states;

[Cut is a graphical line splitting the time-space graph into 2 parts]

Here,

$GS = LS_1 \cup LS_2$.

$LS_1 = \{ send(m_1), send(m_2) \}$

$LS_2 = \{ receive(m_1), receive(m_2) \}$

$GS = \{ send(m_1), send(m_2), receive(m_1), receive(m_2) \}$ strongly consistent

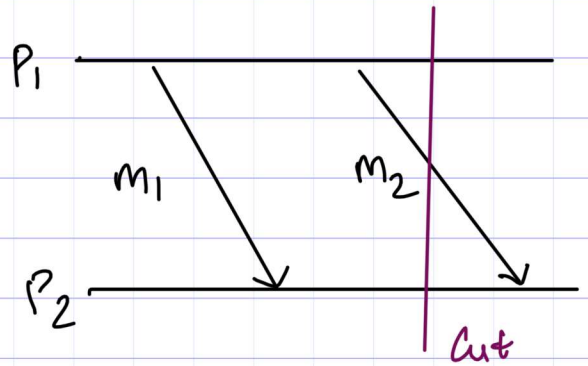
Strongly consistent global state

1) $\forall$ send event in GS, corresponding receive event will be present

2) & no(send) = no(receive)

$LS_1 = \{ send(m_1), send(m_2) \}$
$LS_2 = \{ recv(m_1) \}$

$GS = \{ send(m_1), send(m_2), recv(m_1) \}$



not strongly consistent, but consistent.

m_2 is in transit because only send(m_2) is present.

Consistent Global state.

$\forall$ receive event in GS, corresponding send events are present.

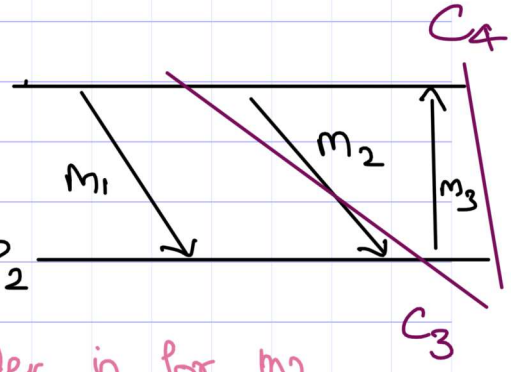
Transit would be there in consistent, but not in strongly consistent.

$LS_1 = \{send(m_1)\}$

$LS_2 = \{rec(m_1), rec(m_2)\}$

$GS_1 = LS_1 \cup LS_2$

Inconsistent global state.



P_2 cannot know who the sender is for m_2 from the global state.

$GS_2 = \{send(m_2), send(m_3), rec(m_3)\}$

$GS_1 \longrightarrow GS_2$ (causal ordering)

But causality fails here because in GS_1 we have a $recv(m_2)$ before the $send(m_2)$ in GS_2.

But $GS_1 \rightarrow GS_2 \Rightarrow$ causal ordering is inconsistent.

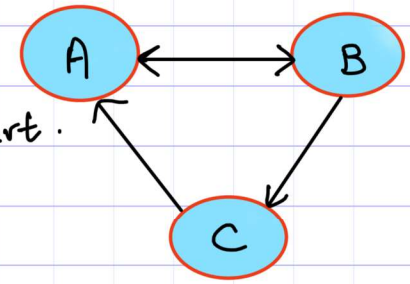
Maintaining strongly consistent states is practically impossible because of performance lag.

(Need to freeze, snapshot, resume operation each time event occurs)

Consistent Global State Recording Protocol

Lamport & Chandy

Any node can start the algorithm.
(Assume here A starts)



Send :

(i) integrate msg. on channel & start.
(ii) Sends a marker message M thro' all outgoing channels.

(Starting the algorithm is done by a leader election method)

Receive :

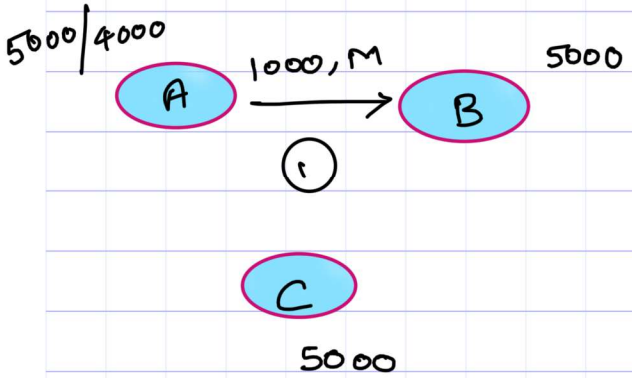
(i) Record States.
(ii) Follow send rule.

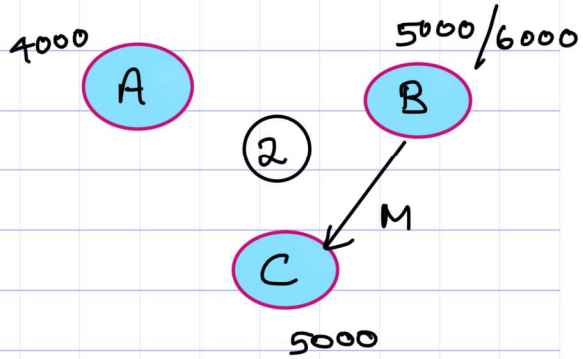
Termination :

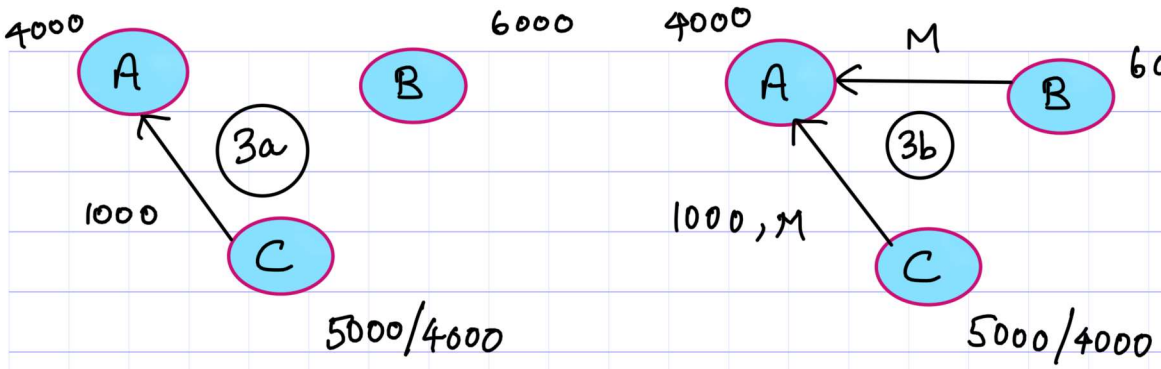
Once all incoming nodes receive the marker message M.

Assumptions :

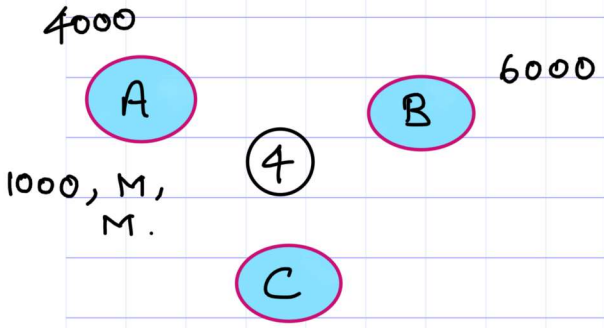
Reliable channel.
Ordering is FIFO (inherently causal order).



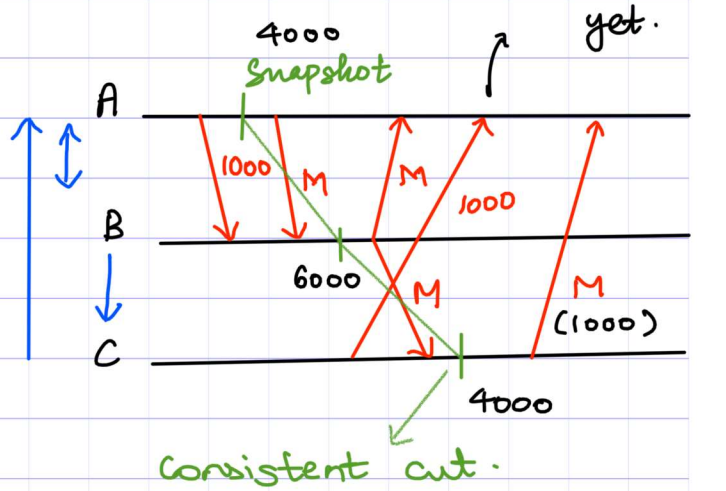




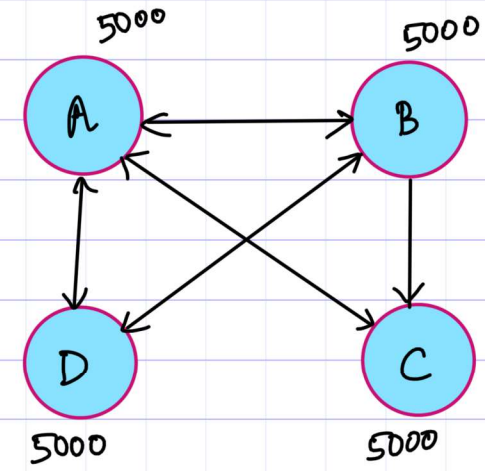
3a & 3b are concurrent events (happens parallelly)

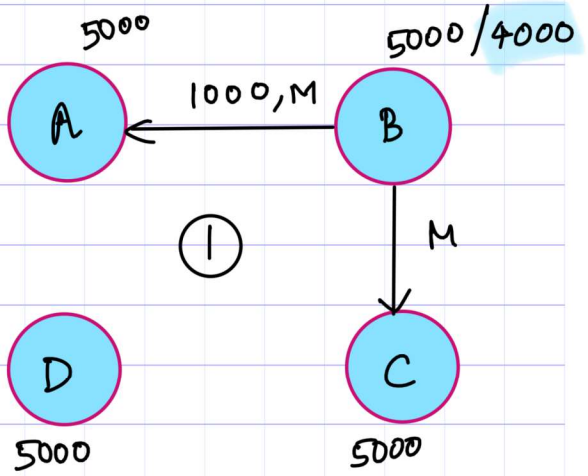


this 1000 has not been recorded yet.

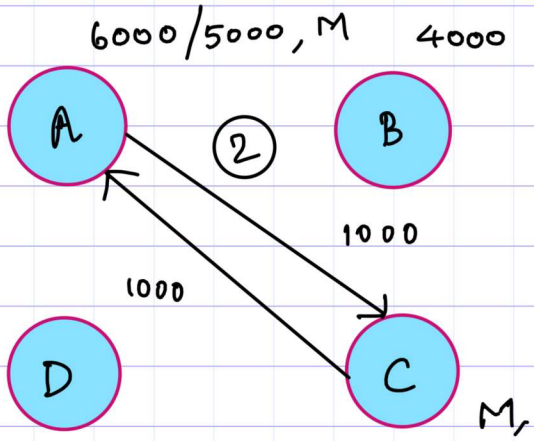


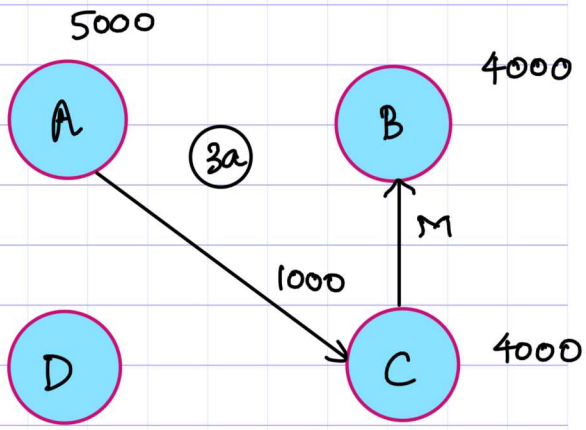
Example:



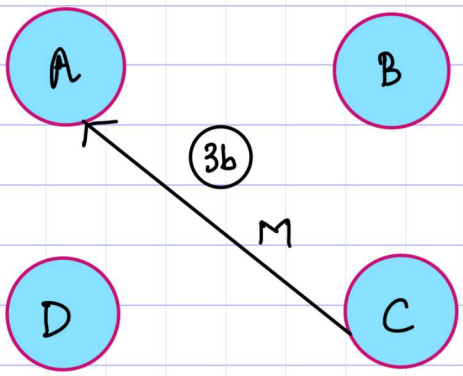


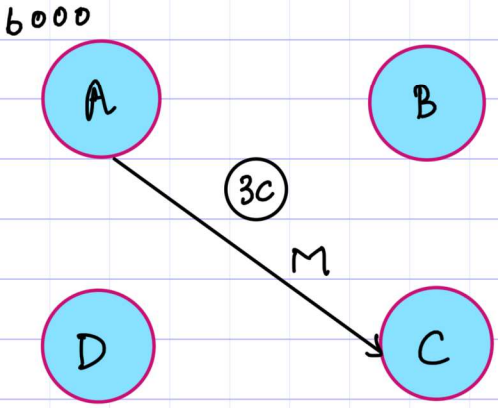
Leader was chosen to be B.
B starts the algorithm

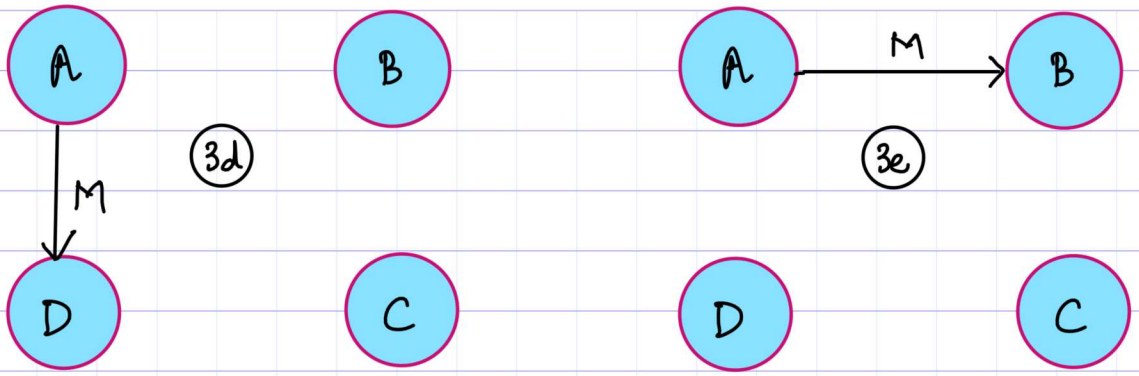




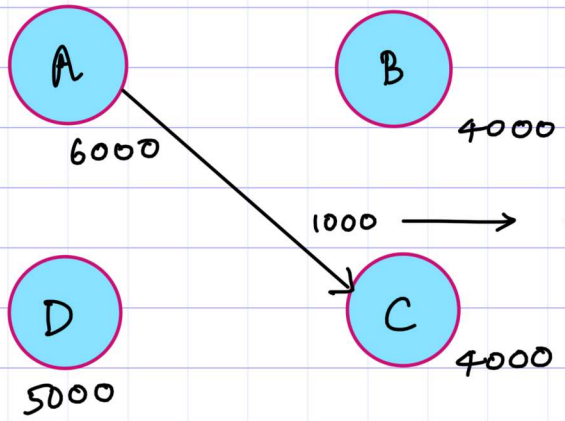
(C sends 1000 by the time it receives M from B)







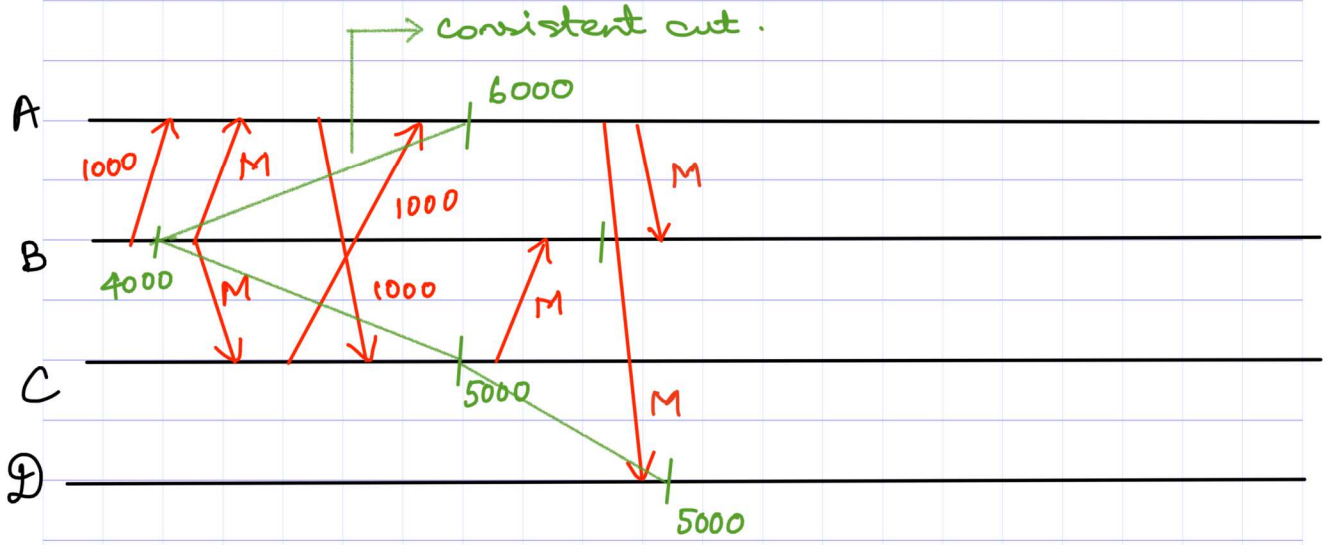
Sender receives marker M from all the incoming nodes, state is consistent.



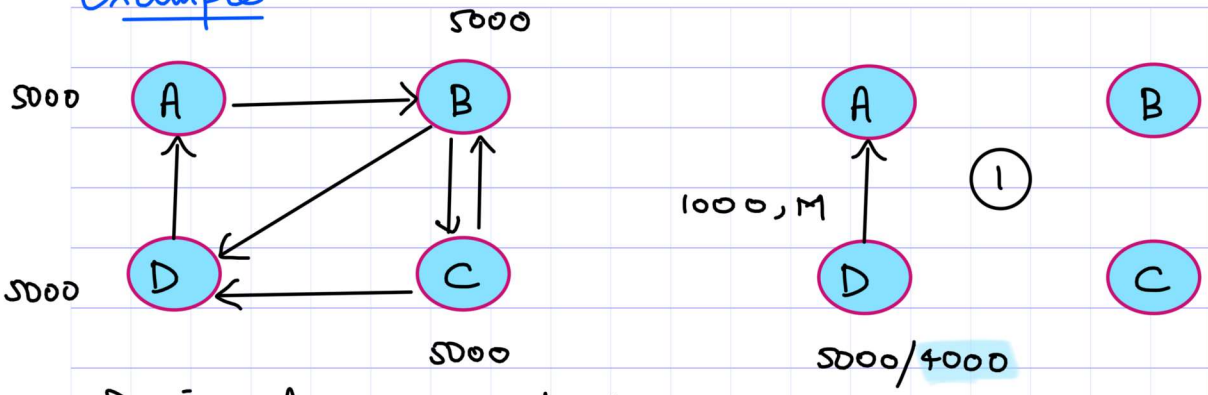
Final snapshot

1000 → still in the channel, not yet accepted.
(because the marker M was received before this 1000 in C)

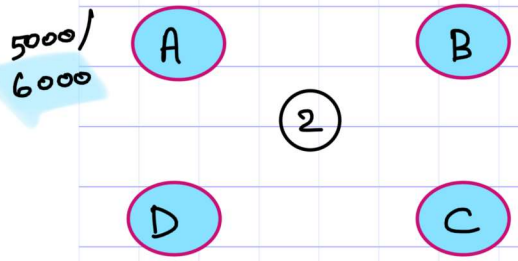
This 1000 can be a part of the next global snapshot.



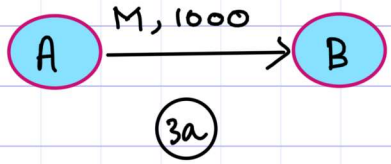
Example



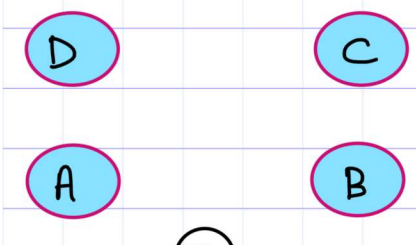
D is chosen as leader.
1000, Marker is sent in order by D.
Since D sends M, D records state as well.



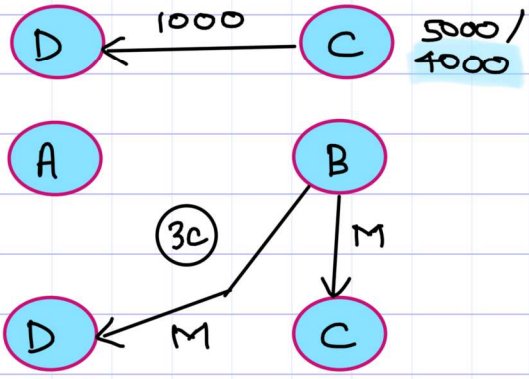
Marker rcvd. by A, state recorded.



B rcvs. M, B snapshots @ 5000, then rcvs. 1000 from A.

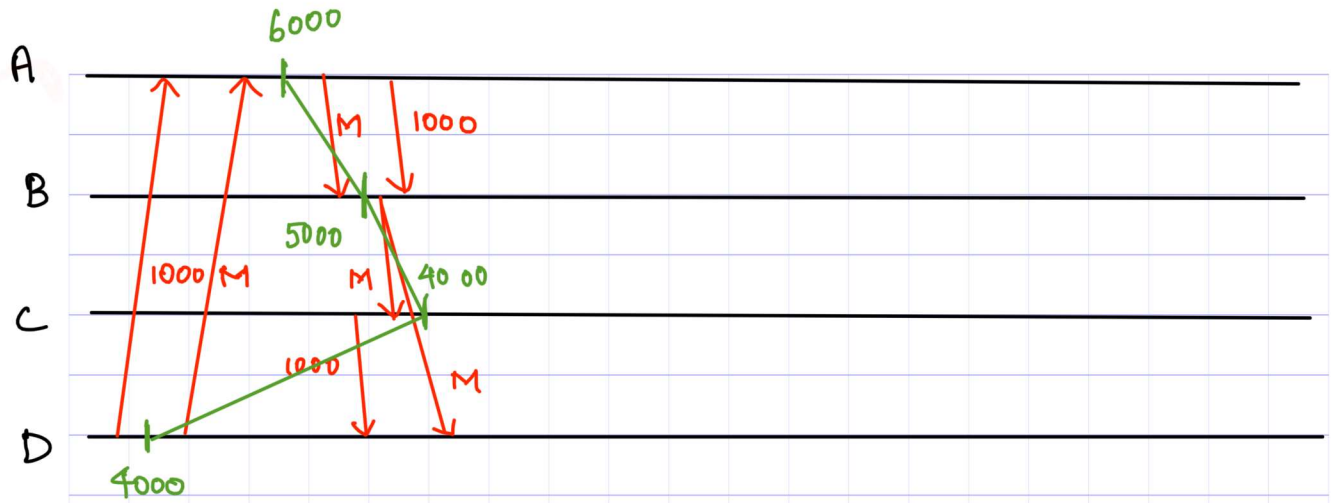


C concurrently sends 1000 to D before getting M from B.

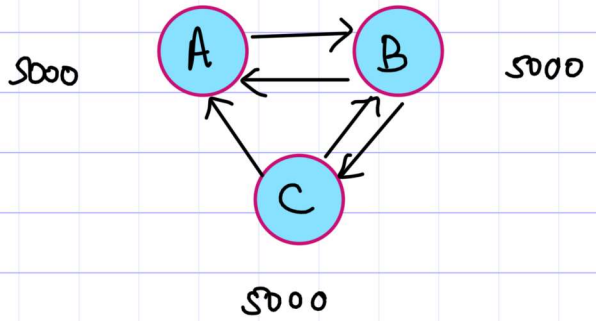


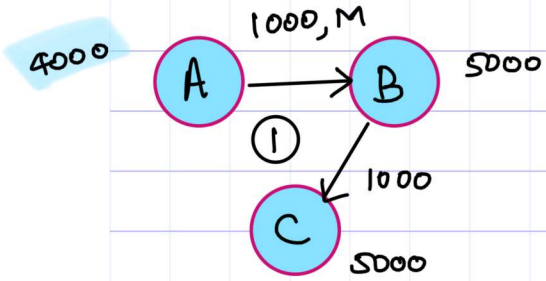
D receives 1000 followed by M from C. But D doesn't snapshot because it is the initiator.

1000 from C→D is waiting in channel.

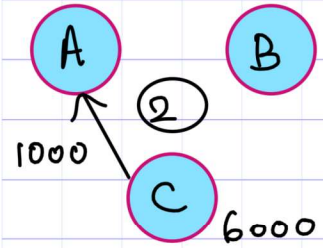


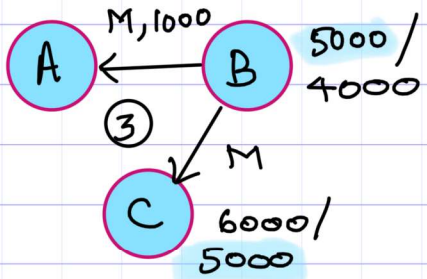
Example :

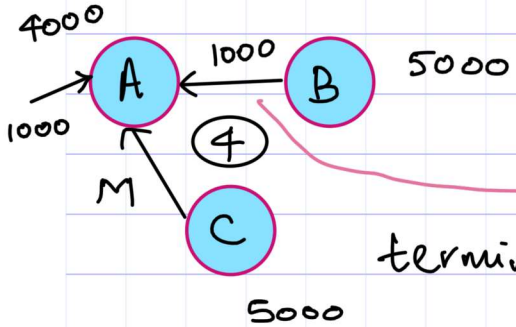




A sends 1000, M to B.
B concurrently sends 1000 to C.

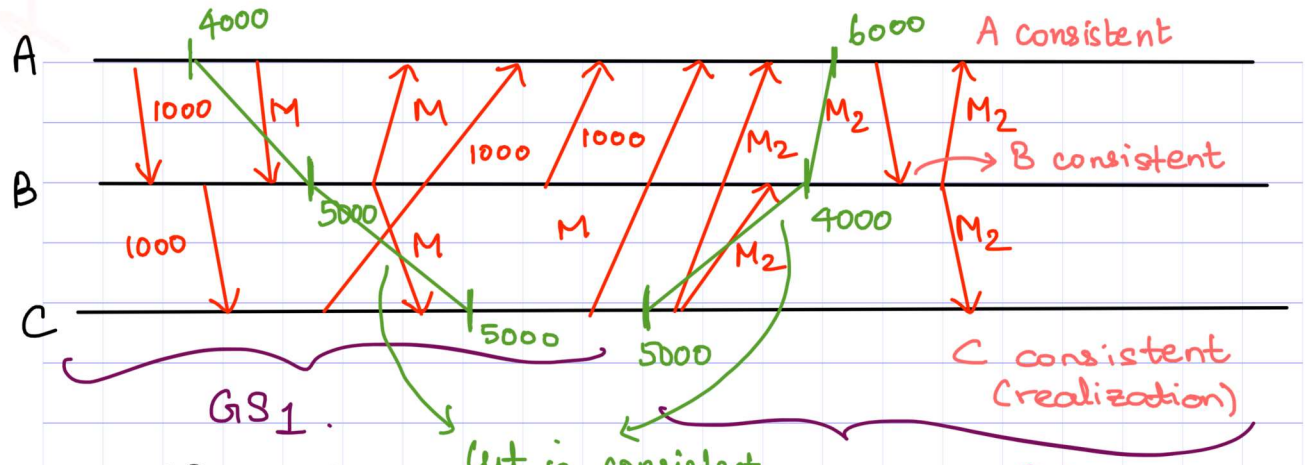


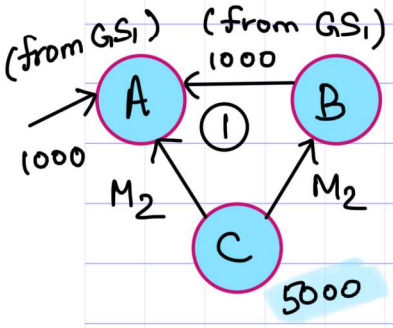




terminate.

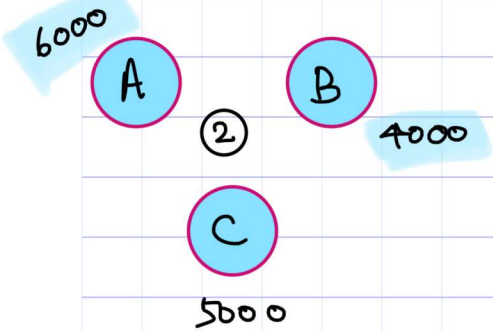
both send & receive are not recorded in snapshot yet because B already sent the marker.





C initiates Marker.

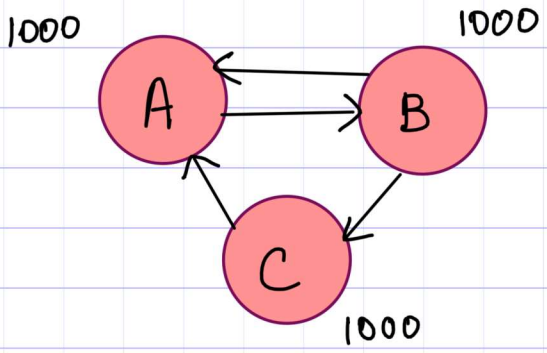
AS SOON AS MARKER IS RECEIVED TAKE A SNAPSHOT!

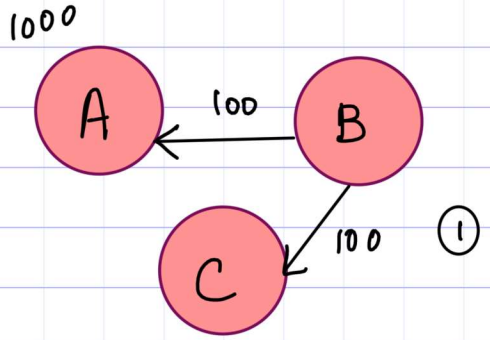


Both cuts are not strongly consistent because the receive & send of 1000 from C → A are recorded in 2 separate snapshots.

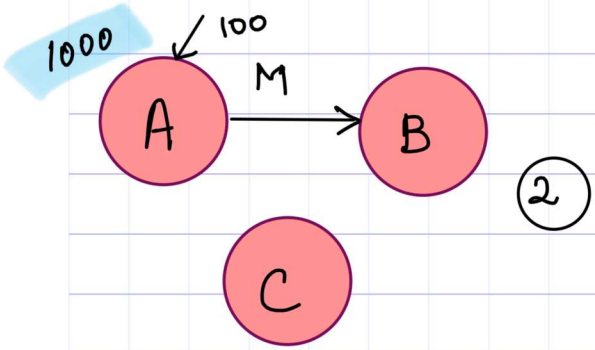
$send_B(1000)$ to A and $receive_A(1000)$ from B will be recorded as part of GS_2.

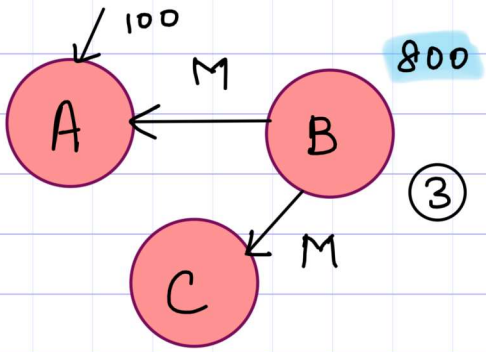
Global State Recording Algorithm

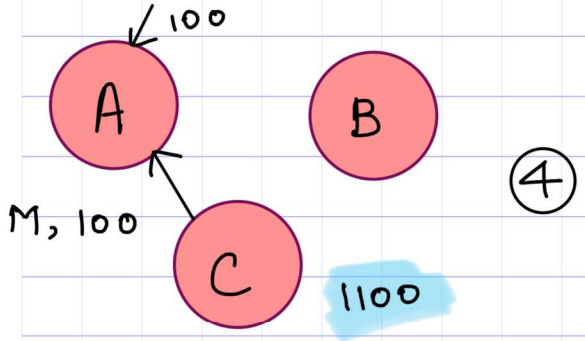


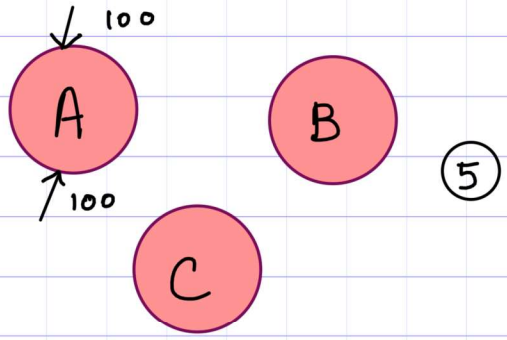


Transactions $B \rightarrow A$
$B \rightarrow C$.



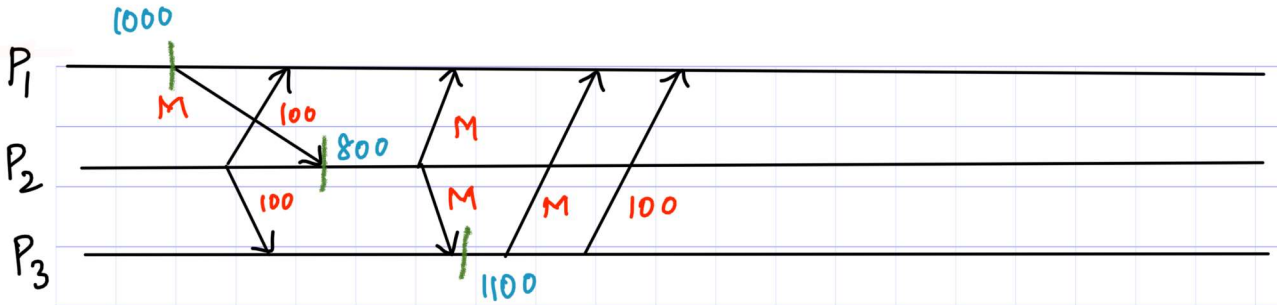






Recording complete.

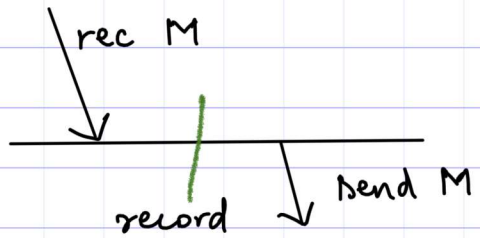
State is consistent.



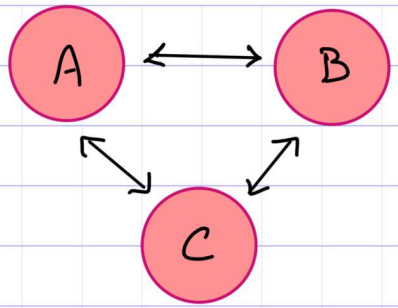
If P_1 initiates the algorithm again,

P_1's state is recorded as 1200

(1000 + 100 from P_2 + 100 from P_3).



general correct procedure.



A initiates algorithm.

5000