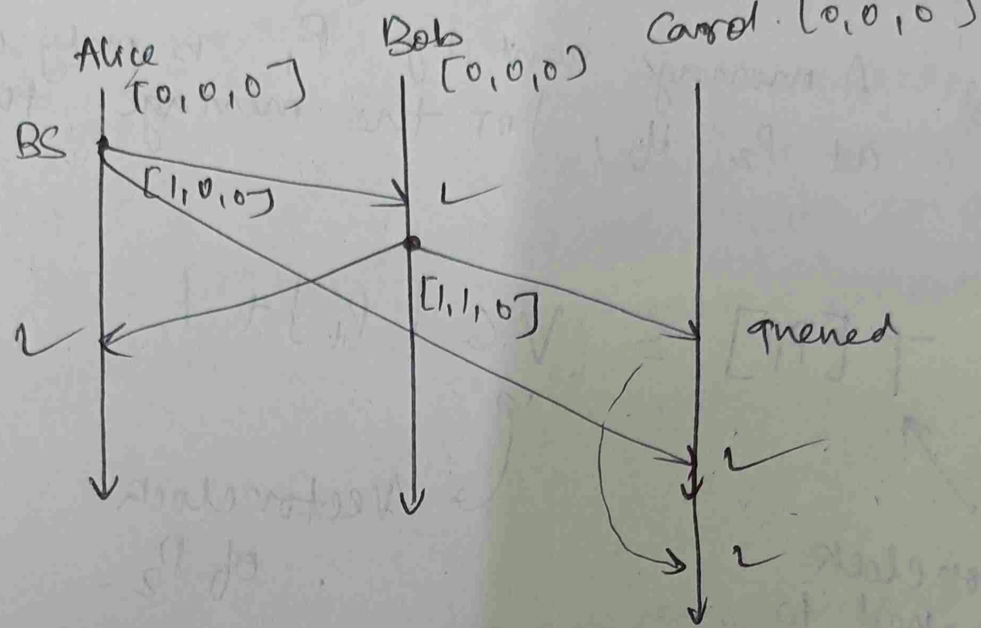
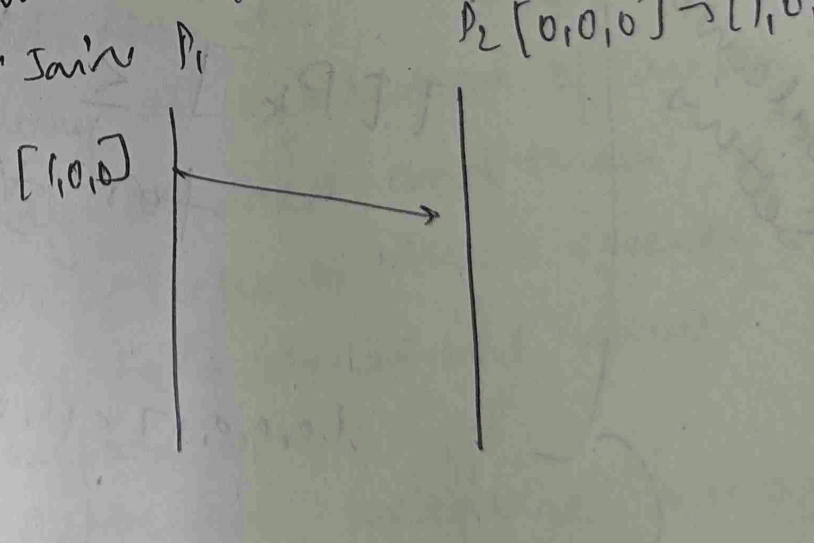
Lec 6:-

Causal broadcast Algorithm.

all messages sent to all processes in the broadcast.



Rule (1):- If a message sent by P1 (Process 1) is delivered at P_2, increment P_2's local local in the P_1 position.
↳ Jain P_1



Increment only happens at the delivery time.

Rule (2) :- If a message is sent by a process, first increment its own position in its local clock, and include the local clock along with the msg.

Rule (3) : A message sent by P_1 is only delivered at P_2 if, for the message timestamp T:

$$T[P_1] = VC[P_1] + 1$$

Vector clock attached to message from P_1

Vectorclock of P_2.

and

$$T[P_k] \leq VC[P_k]$$

for $\forall$ $k \neq 1$

other than the process sending the msg.

$[0,0,0,1] \leq [1,0,0,1]$

for broadcast systems

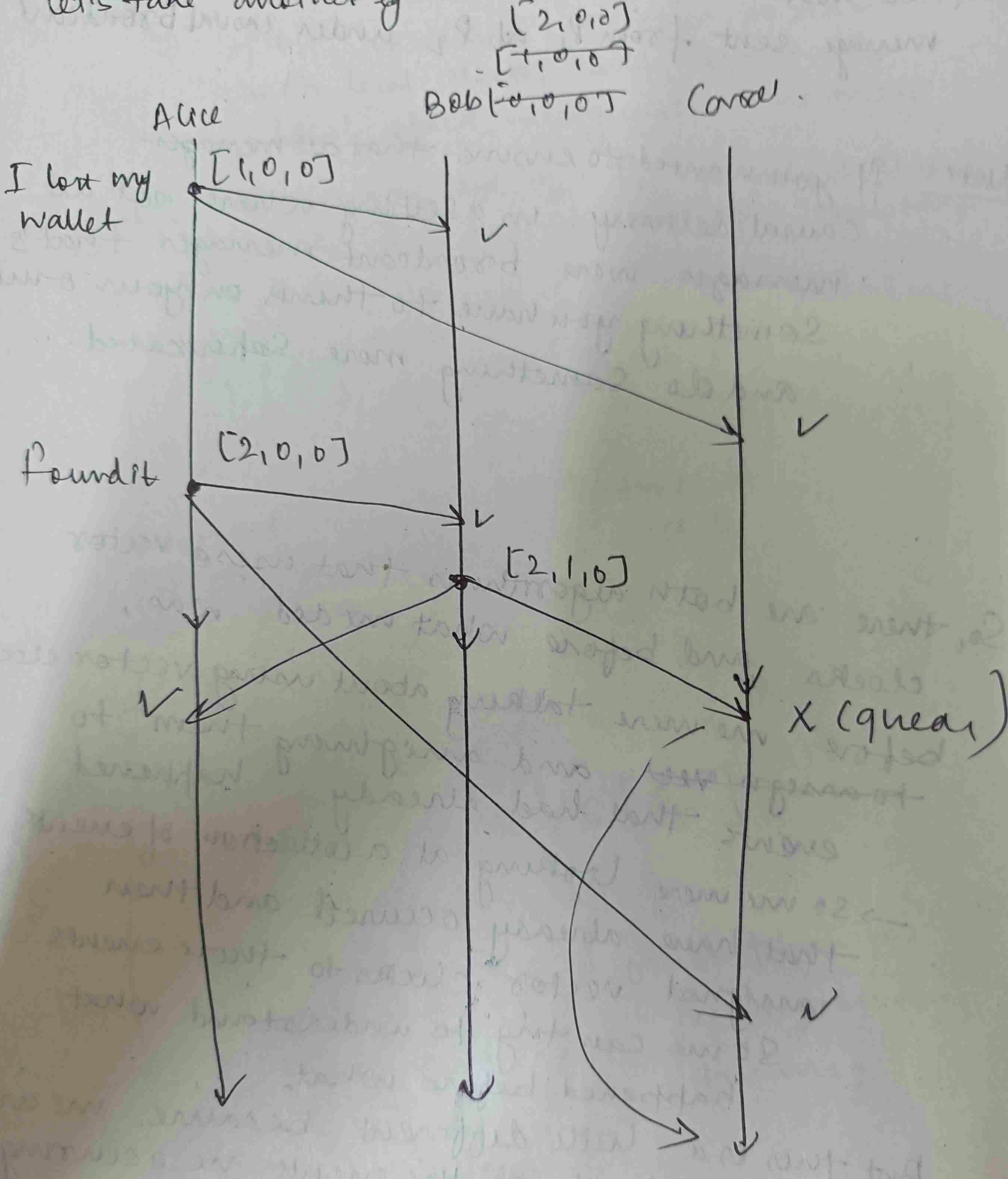
There should follow only when we are considering "sent" as an event and not receiving as event

So, there were the rules for when you can deliver a message sent from P_1 at P_2 under casual broadcast.

Note: If you wanted to ensure ~~that all messages~~ causal delivery in a setting where not all messages were broadcast messages that's something you have to think on your own and do something more sophisticated.

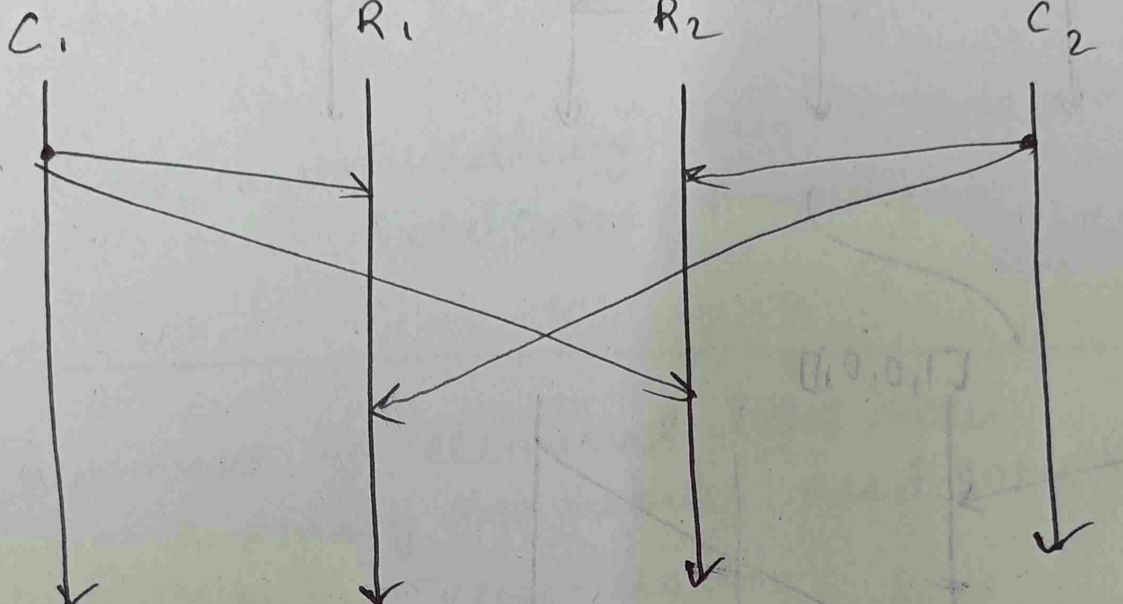
So, there are both algorithms that uses vector clocks and before what we did was, before we were talking about using vector clocks ~~to assign vect~~ and assigning them to events that had already happened
→ so we were looking at a collection of events that have already occured and then assigned vector clocks to those events so we can try to understand what happened before what.
But this is a little different because we are talking about as the events are occurring we are talking about "using vector clocks to decide whether or not to deliver messages as they are happening".

let's take another eg

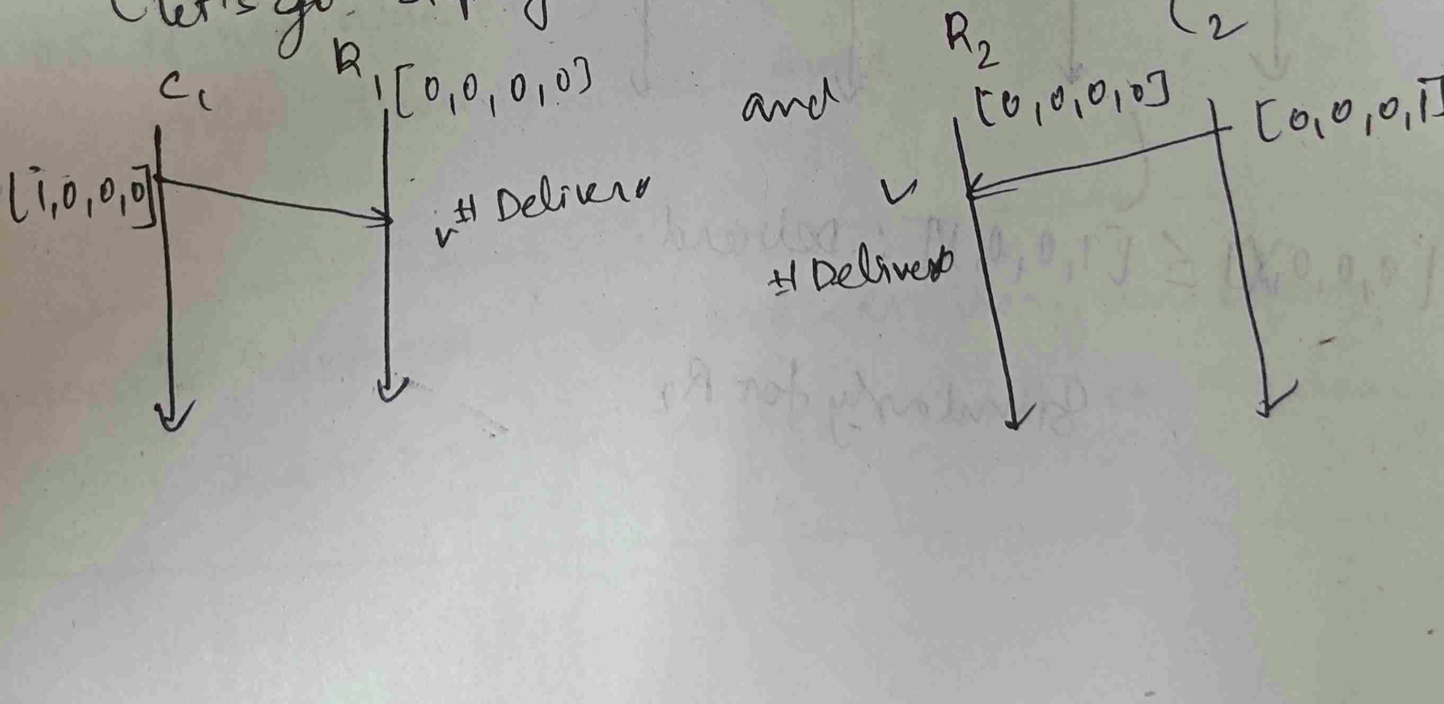


Can vector clocks rule out "Total Order" Anamoly??

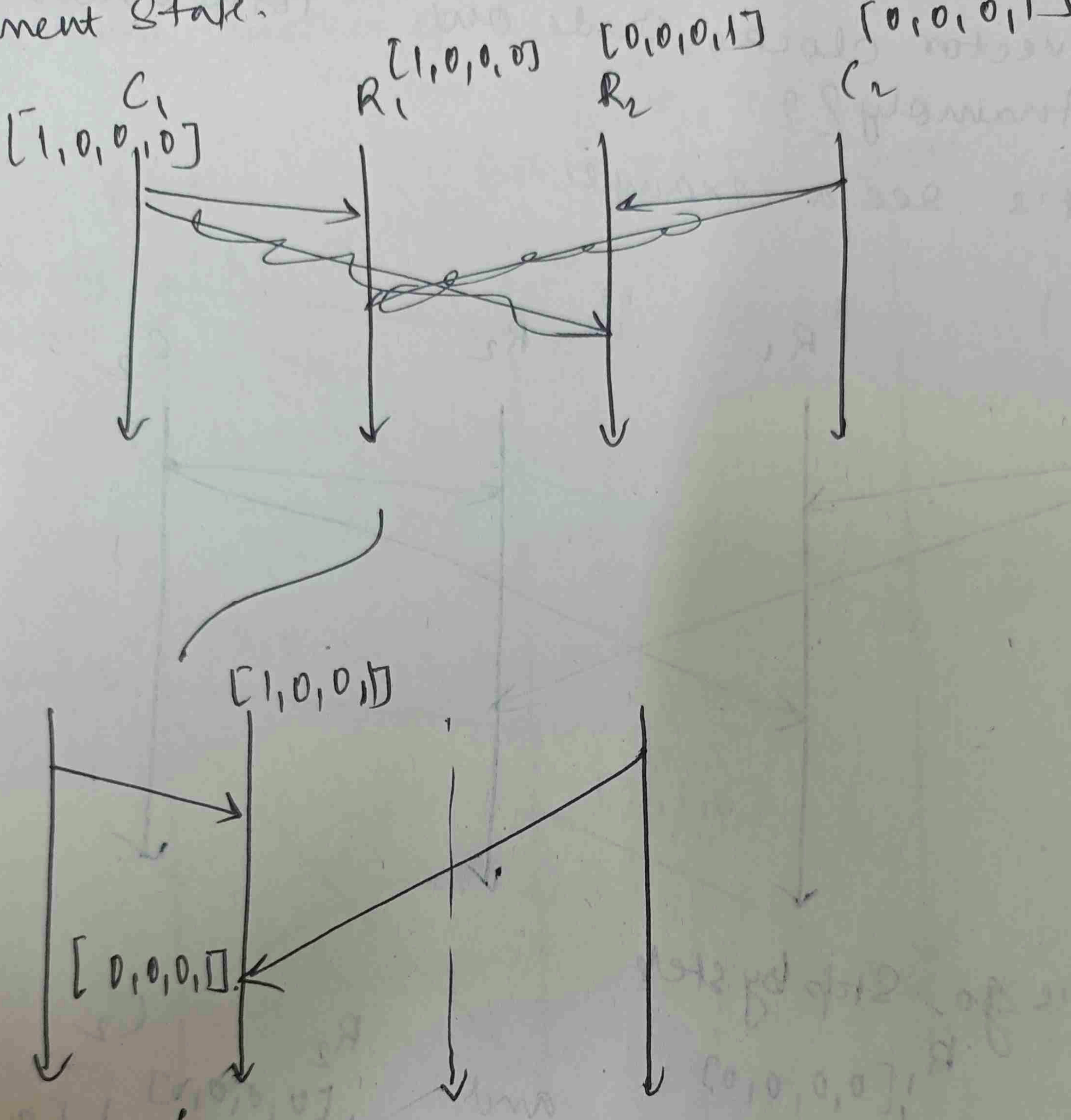
(Let's see a example.



(Let's go. Step by step

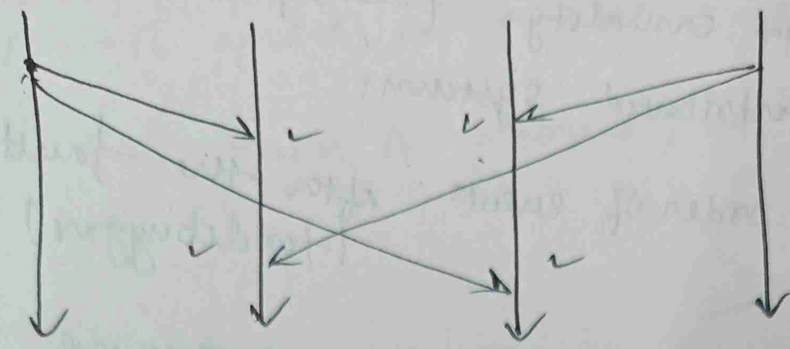


Current State.



$[0,0,0,\cancel{1}] \le [1,0,0,\cancel{1}]$ ∴ Delivered.

Similarly for R_2.



∴ So, casual delivery doesn't rule out Total Order Anamoly.

So, if we want to eliminate Total order Anamoly we would need something more than vector clocks.

or if we wanted both casual order and total order then we need more.

In general, Enforcing Total order is very annoying, So don't go for implying until its necessary.

"happens before"

ways that potential casualty ($\longrightarrow$) is used in distributed system!

→ determine order of events after the fault (for debugging)

- causal ordering of events as they are happening

- $\boxed{\text{consistent global snapshots}}$

There is a way of getting a picture of the global state of a distributed system and it's not trivial to do because each of these processes in a distributed system has its own state. and its own knowledge about other processes states

So there are some ways to try to take a global snapshots. Some ways that makes sense ~~to~~ and some ways that really don't make sense.

[if A → B and B is in the snapshot,
then A should be too.]

What it means to take a global snapshot of a distributed system.?

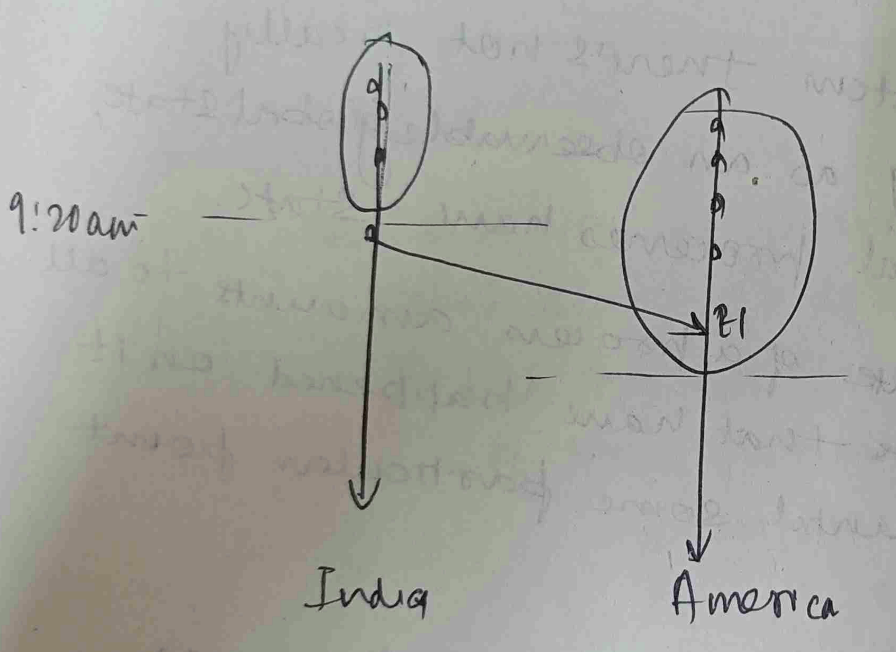
— In Distributed System there's not really such a thing as an observable global state, So individual processes have state.

and the state of a process amounts to all the events that have happened on it into until some particular point

But in general we have several processes and we want to be able to talk about the state of the entire System

So, if we had some sort of globally synchronized time of day clock, we could say "everybody take a snapshot of yourself at 9:20 am but we can't do it because my 9:20 am isn't necessarily the same as other"

So, this approach of telling everything that "take a snapshot of yourself at a particular time, this is prone to error because not everybody's clock can't be perfectly schronized"

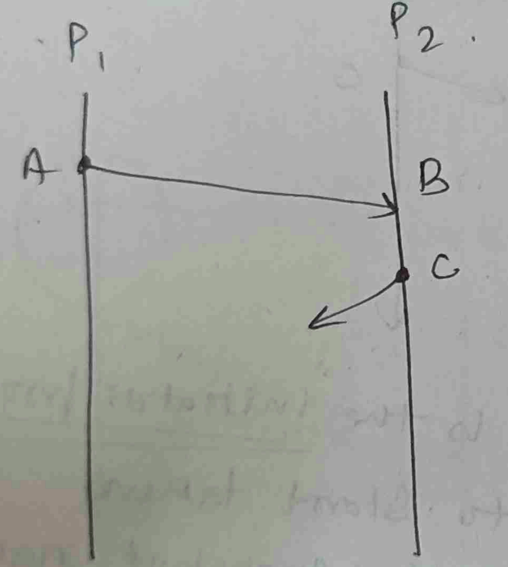


India snapshot E1 ming gaya na
aur American ke snapshot
mai E1 hi

So, what we need is some sort of algorithm that will allow us to take a global snapshot that actually makes sense.

Chandy-Lamport Algorithm

<u>Channel</u> → A connection from one process to another.



There are two ~~connect~~ channels / connection here.

C_{12}: channel from P_1 to P_2.

C_{21}: channel from P_2 to P_1

Yaha par A se msg gya aur B tak pahuch gaya to wo shi h

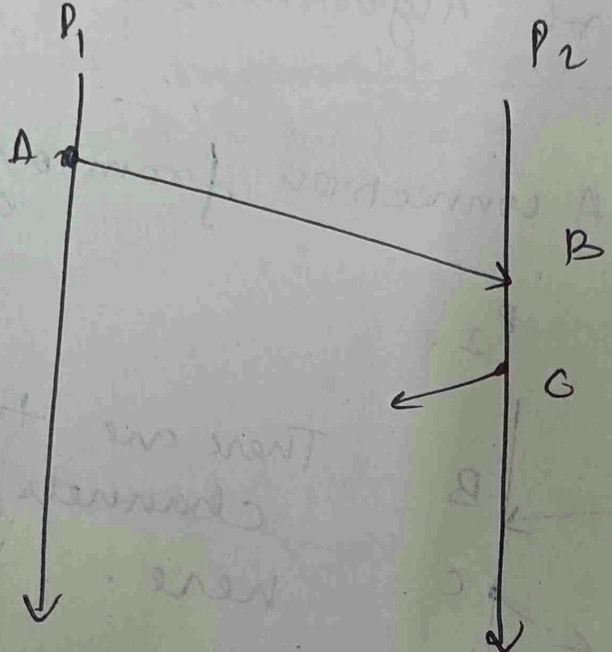
lekin C se msg gaya lekin wo pahucha n.

to we have "one msg in the channel C_{21}"

and "no msg in the channel C_{12}."

~~Cham~~

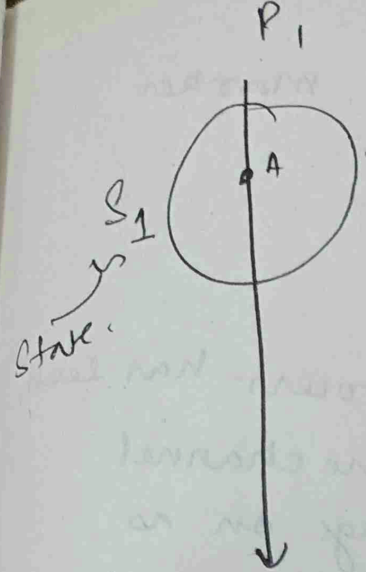
Channels acts like FIFO queues.



Let's say that P1 here is the "initiator process",
let's say P1 decides to start taking snapshots and take the snapshot right after message "A" is sent

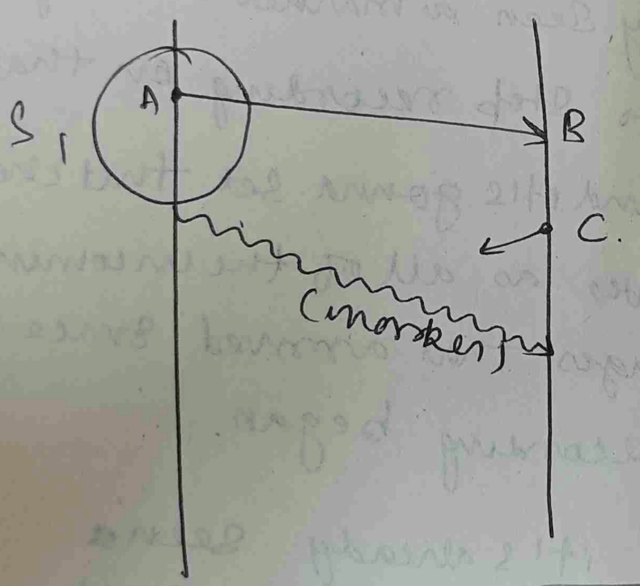
⇓

So it record it states just after sending A



after P1 records its States it sends what's called a "marker message"

and it sends that marker message out on all of its outgoing channels (in this case, it has only one outgoing channel, that is to P2)



P1 has to send marker message before it does anything else after snapshotting its own state.

after it sends that marker message it starts recording the messages that it receives on all of its incoming channels, (in this case it has only one incoming channel)

What happens when someone receives a marker message?

→ There are two cases! -

(i) If its the first marker that that process has seen, it records its state, it marks the channel that it got the marker message on as empty and it send its marker msg out on all of its outgoing channels.

(ii) If it has already seen a marker message before, then its going to stop recording on that channel and it's gonna set that channels final states as all of the incoming messages that arrived since recording began

(So, if it's already seen a marker msg and it receives another one then it stop recording incoming messages on that channel)