The Need for Propagating Acknowledgments

- Can I tell that all processes are aware that D has progressed beyond time t.

  Nodes need this information to determine when it is safe to discard/stabilize updates.

Matrix Time

- In the system of matrix clocks, the time is represented by a set of n x n matrices of non-negative integers.

- A process p_i maintains a matrix $mt_i[1..n, 1..n]$ where, $mt_i[i, i]$ denotes the local logical clock of p_i and tracks the progress of the computation at process p_i.

- $mt_i[i, j]$ denotes the latest knowledge that process p_i has about the local logical clock, $mt_j[j, j]$, of process p_j. Mt[i,*] is vector vt_i

- $mt_i[j, k]$ represents the knowledge that process p_i has about the latest knowledge that p_j has about the local logical clock, $mt_k[k, k]$, of p_k .

- The entire matrix mt_i denotes p_i's local view of the global logical time.

Matrix Time

Process p_i uses below rules to update its clock:

- Rule 1 : Before executing an event, process p_i updates its local logical time as follows:

$mt_i[i, i] := mt_i[i, i] + d$

Matrix Time

- Rule 2: Each message m is piggybacked with matrix time mt. When p_i receives such a message (m,mt) from a process p_j, then p_i executes the below sequence:
  1. Update its global logical time as follows:
     (a) $1 \leq k \leq n : mt_i[i, k] := ?$

     (b) $1 \leq k, l \leq n : mt_i[k, l] := \max(mt_i[k, l], mt[k, l])$
  2. Execute Rule 1.
  3. Deliver message m

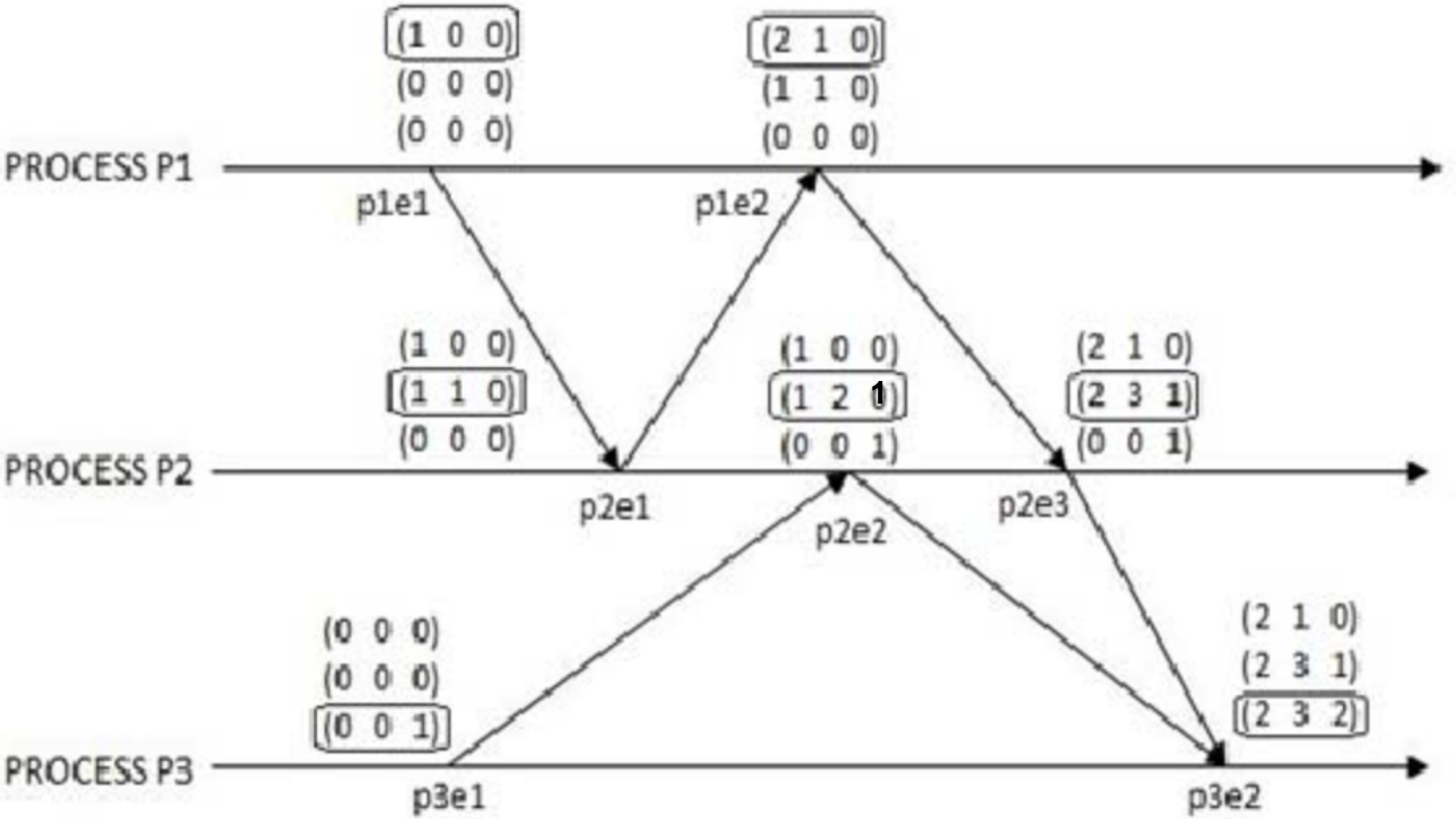
Matrix Time

Process p_i uses below rules to update its clock:

- Rule 1 : Before executing an event, process p_i updates its local logical time as follows:

  $mt_i[i, i] := mt_i[i, i] + d \ (d > 0)$

- Rule 2: Each message m is piggybacked with matrix time mt. When p_i receives such a message (m,mt) from a process p_j, then p_i executes the below sequence:
  1. Update its global logical time as follows:
     (a) $1 \leq k \leq n : mt_i[i, k] := \max(mt_i[i, k], mt[j, k])$ (That is, update its row $mt_i[i,*]$ with the p_j's row in the received timestamp, mt.)
     (b) $1 \leq k, l \leq n : mt_i[k, l] := \max(mt_i[k, l], mt[k, l])$
  2. Execute Rule 1.
  3. Deliver message m


(1 0 0)
(0 0 0)
(0 0 0)
(2 1 0)
(1 1 0)
(0 0 0)
PROCESS P1
p1e1
p1e2
(1 0 0)
(1 1 0)
(0 0 0)
(1 0 0)
(1 2 1)
(0 0 1)
(2 1 0)
(2 3 1)
(0 0 1)
PROCESS P2
p2e1
p2e2
p2e3
(0 0 0)
(0 0 0)
(0 0 1)
(2 1 0)
(2 3 1)
(2 3 2)
PROCESS P3
p3e1
p3e2

Matrix Time

Current update is:

$1 \le k \le n : mt_i[i, k] := \max(mt_i[i, k], mt[j, k])$ (That is, update its row $mt_i[i,*]$ with the p_j's row in the received timestamp, mt.)

Suppose process j sent msg with mt[p,k]=10 but mt[j,k]=5 while $mt_i[i, k]=3$
then $mt_i[i, k]$ becomes only 5 and not 10.
Should it be $\max(mt_i[i, k] , \max(mt[*,k]))$?

Matrix Time

Current update is:

$1 \le k \le n : mt_i[i, k] := \max(mt_i[i, k], mt[j, k])$ (That is, update its row $mt_i[i,*]$ with the p_j's row in the received timestamp, mt.)

Look at this: mt[p,k]=10 but mt[j,k]=5. Is this possible?
When process j received matrix from p then matrix of j would have got updated as
$mt_j[j, k] := \max(mt_j[j, k], mt[p, k])$ and so $mt_j[j, k] = 10$.
So it cannot happen that mt[p,k]=10 but mt[j,k]=5

what if mt[p,k] was not updated due to no direct msg by p?

Basic Properties of Matrix Time

- Vector $mt_i[i, .]$ contains all the properties of vector clocks.

- In addition, matrix clocks have the following property:
  - $\min_k (mt_i[k, i]) \geq t \Rightarrow$ process p_i knows that every other process p_k knows that p_i's local time has progressed till t.
- If this is true, it is clear that process p_i knows that all other processes has received p_i 's msg sent at t and hence can discard the msg.

- In many applications, this implies that processes will no longer require from p_l certain information (it might be holding information until all processes are updated about some msg from p_l)and can use this fact to discard obsolete information.

Matrix Time Vs. Vector Time

- Matrix time is useful in settings such as maintaining a distributed dictionary or a distributed database.
- Updates are streamed continuously across the sites.
- Each site has to know when an update message needs to be no longer stored.
- Matrix time can tell local sites when to discard old messages.

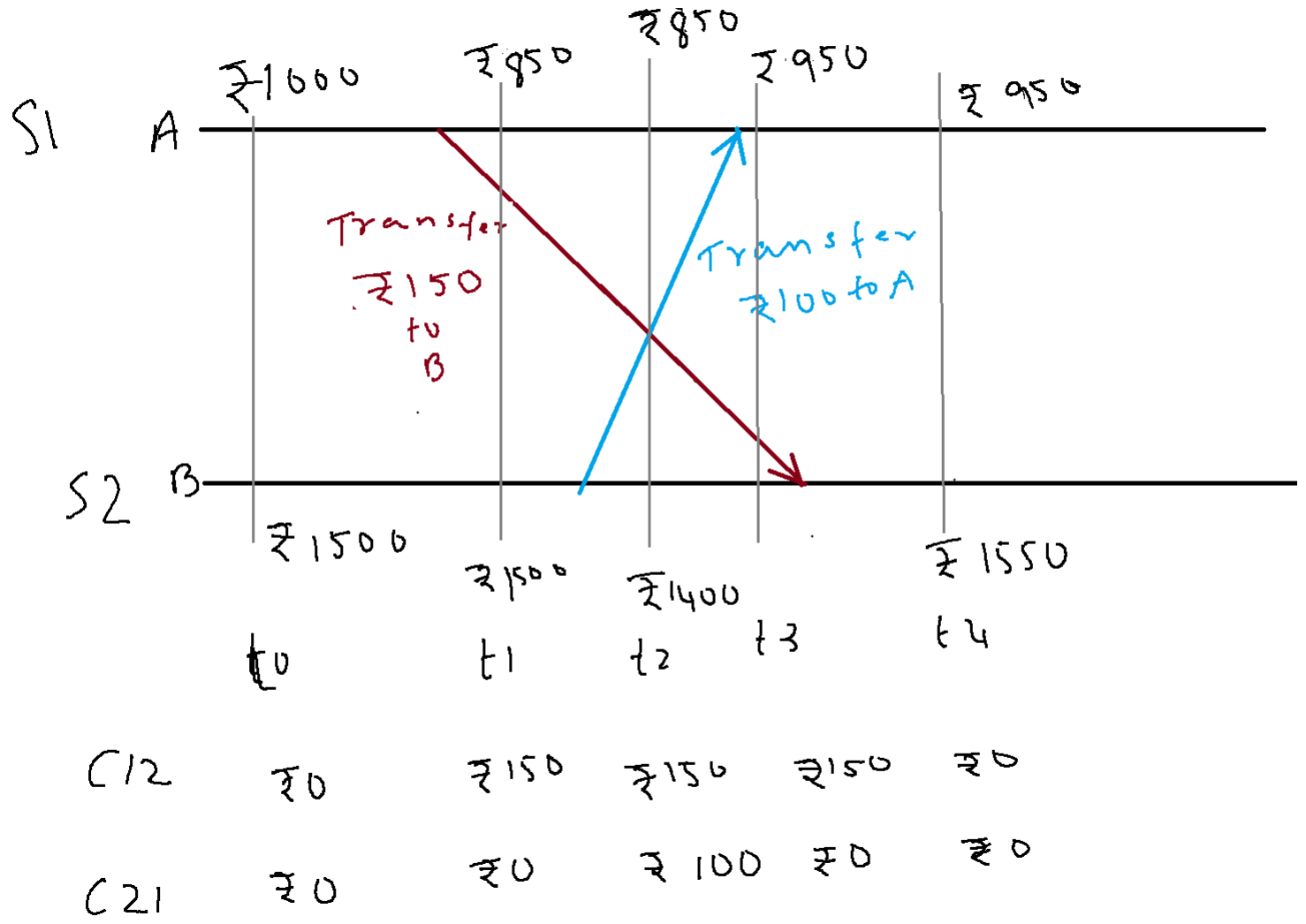
A Distributed Program in Execution

- How can causal order among messages be guaranteed? Read about it from **An Efficient Causal Order Algorithm for Message Delivery in Distributed System, Jangt, Park, Cho, and Yoon**

Global State of a Distributed System

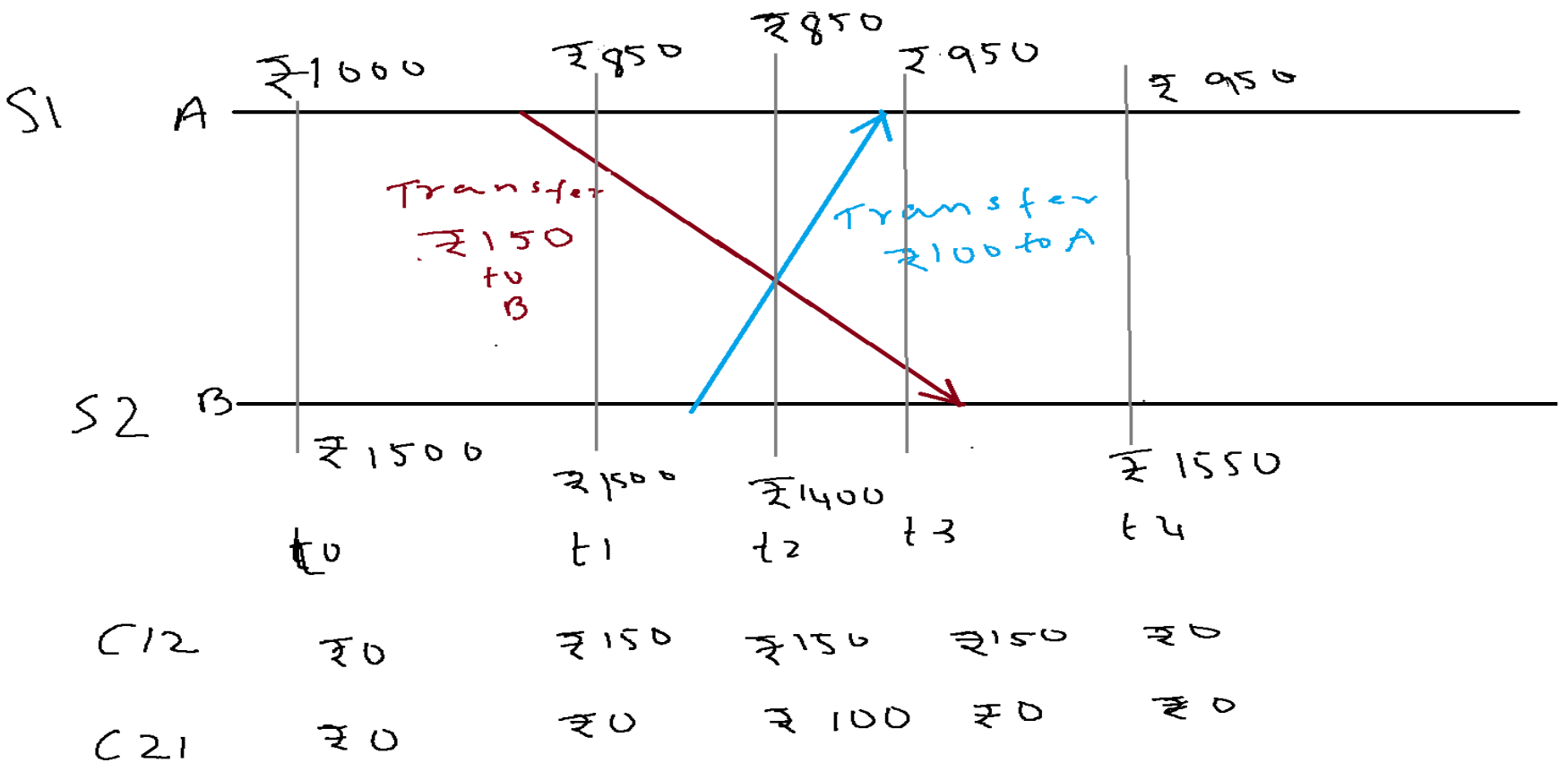
- Cant stop the process and see the global snapshot and neither can we synchronise all the systems and ask them to take a snapshot at a particular instant.
- Not necessary a state the system has visited but is one state that the system could have visited
- Application: Check Pointing Recovery, Deadlock needs understanding current state to measure if a true deadlock.
- Issues: We need to know both node and channel states as some msgs might be in transit
- Assumption: System can not be stopped to take global state. Also no global clock to do the same.

Global States and Snapshots



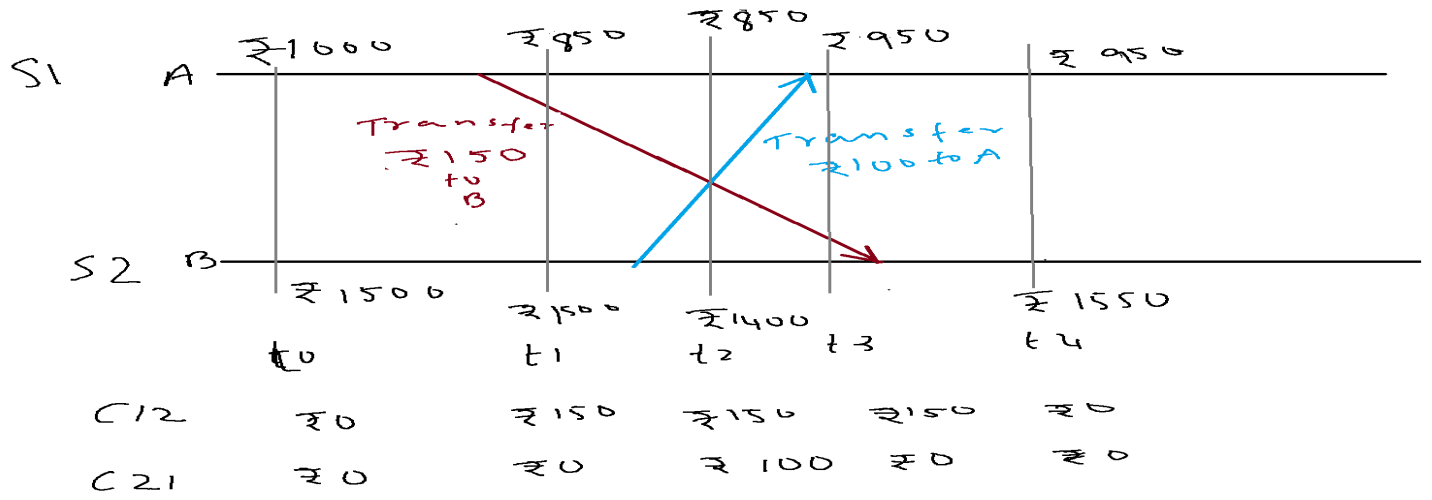
- Consider the distributed execution shown above.

Global States and Snapshots



- If the state of the account A is recorded at time t0 and the state of account B and that of the channels C12 and C21 are recorded at t2, what is the total money in the system?

Global States and Snapshots



- Initially 1000+1500=2500
- At check point t0 for A and t2 for B we have 1000(A) +1400(B)+150(C_{12})+100(C_{21})= 2650 An extra 150 Rs !!!.
- If this state is used for checkpointing, then there is trouble!

Global State of a Distributed System

- One can think of the collection of local states as the global state.
- Recall that the local state of a process, P_i, is the contents of the processor registers, stack, local memory, etc.
- The state of a channel is the set of messages in transit in the channel.
- Events lead to changes in the state of local process(es).

Global States and Snapshots

- Recording the global state of a distributed system on-the-fly is an important paradigm.
- The lack of globally shared memory, global clock and unpredictable message delays in a distributed system make this problem non-trivial.
- Building on the definition consistent global states we discuss issues to be addressed to obtain consistent distributed snapshots.
- Then several algorithms to determine on-the-fly such snapshots are presented for several types of networks.

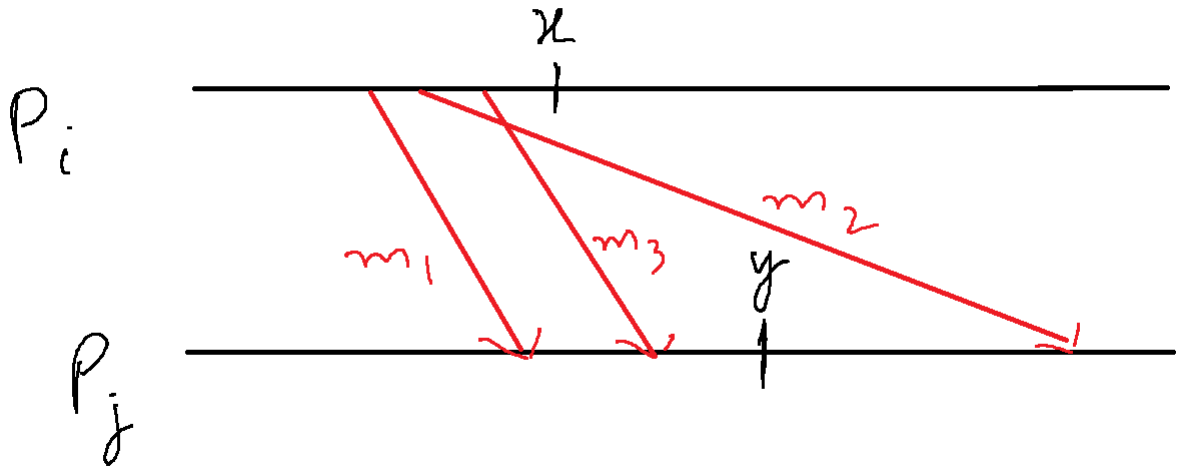
Global State of a Distributed System

- At any instant the state of a process Pi denoted by LS_i, is a result of the sequence of all events executed by p_i upto that instant. State of a Channel C_{ij} is denoted as SC_{ij}

  The msgs in transit on the channel C_{ij} are denoted as $SC_{ij} = \{m_{ij} \mid send(m_{ij}) \in LS_i \text{ and } rec(m_{ij}) \notin LS_j\}$

- To formalize, let LS^x_i denote the local state of process P_i after the occurrence of all events until event e^x_i
  - LS^0_i is the initial state of P_i

Global State of a Distributed System



- We use the above to capture the state of a channel C_{ij}, denoted $SC^{x,y}_{i,j}$ as follows.

$$SC^{x,y}_{i,j} = \{m_{ij} \mid send(m_{ij}) <= LS^{x}_{i} \text{ and } recv(m_{ij}) \not\leq LS^{y}_{j}\}$$

- In words, the state of $SC^{x,y}_{i,j}$ denotes all messages that have been sent by P_i up to event e^{x}_{i} and not received by P_j up to event e^{y}_{j}.

Conditions for Consistent Global States

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state GS is defined as,
  - $GS = \{ \cup_i LS_i , \cup_{i,j} SC_{ij} \}$
- A global state GS is a consistent global state iff it satisfies the following two conditions :

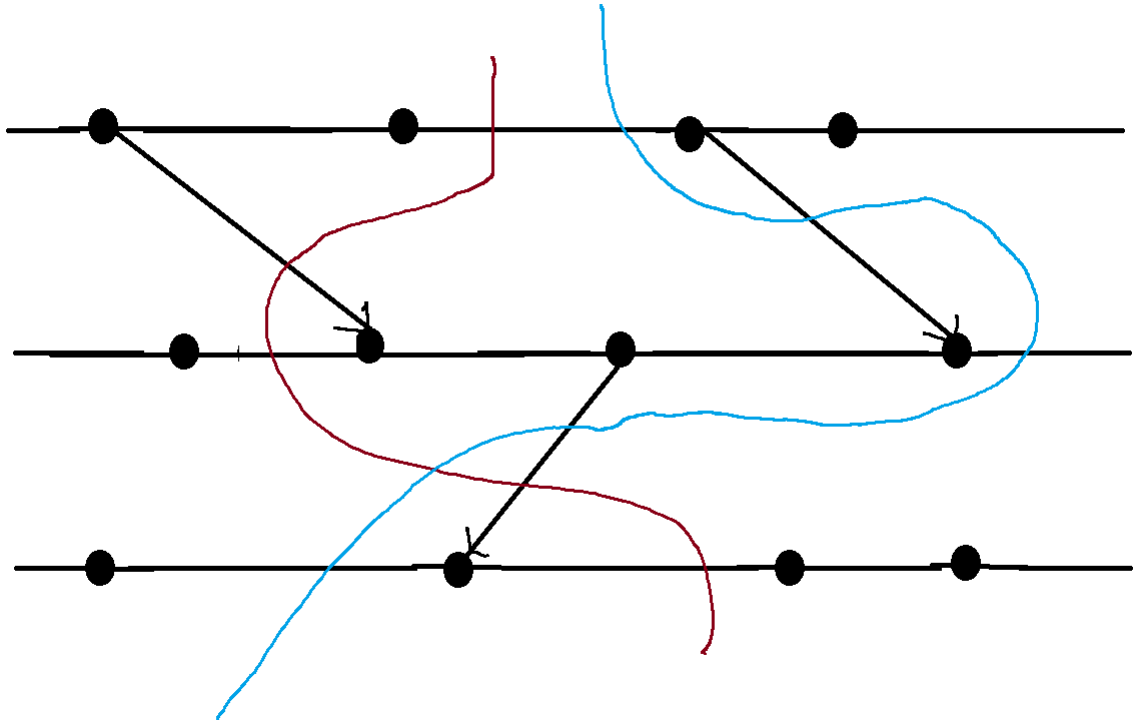
C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$. ($\oplus$ is the Exclusive-OR operator)

C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$.

Conditions for Consistent Global States

- A global state GS is a consistent global state iff it satisfies the following two conditions :
- C1: send(m_{ij})$\in$ LS_i $\Rightarrow$ m_{ij} $\in$ SC_{ij} $\oplus$ rec(m_{ij})$\in$$LS_j$. ($\oplus$ is the Exclusive-OR operator)
- C1 states the law of conservation of messages.
  - Every message that is recorded as sent in the local state of some process is either captured in the state of the channel or is captured in the local state of the receiver.
- C2: send(m_{ij})$\notin$ LS_i $\Rightarrow$ m_{ij} $\notin$ SC_{ij} $\wedge$ rec(m_{ij}) $\notin$ LS_j.
- C2 states that for every cause there is an effect.
  - If a message is not recorded as sent in the local state of a process P_i, then the message cannot be included in the state of the channel C_{ij} or be captured as received by P_j.

Interpretation in terms of cuts



- A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.
- Such a cut is known as a consistent cut.

Interpretation in terms of cuts – Example

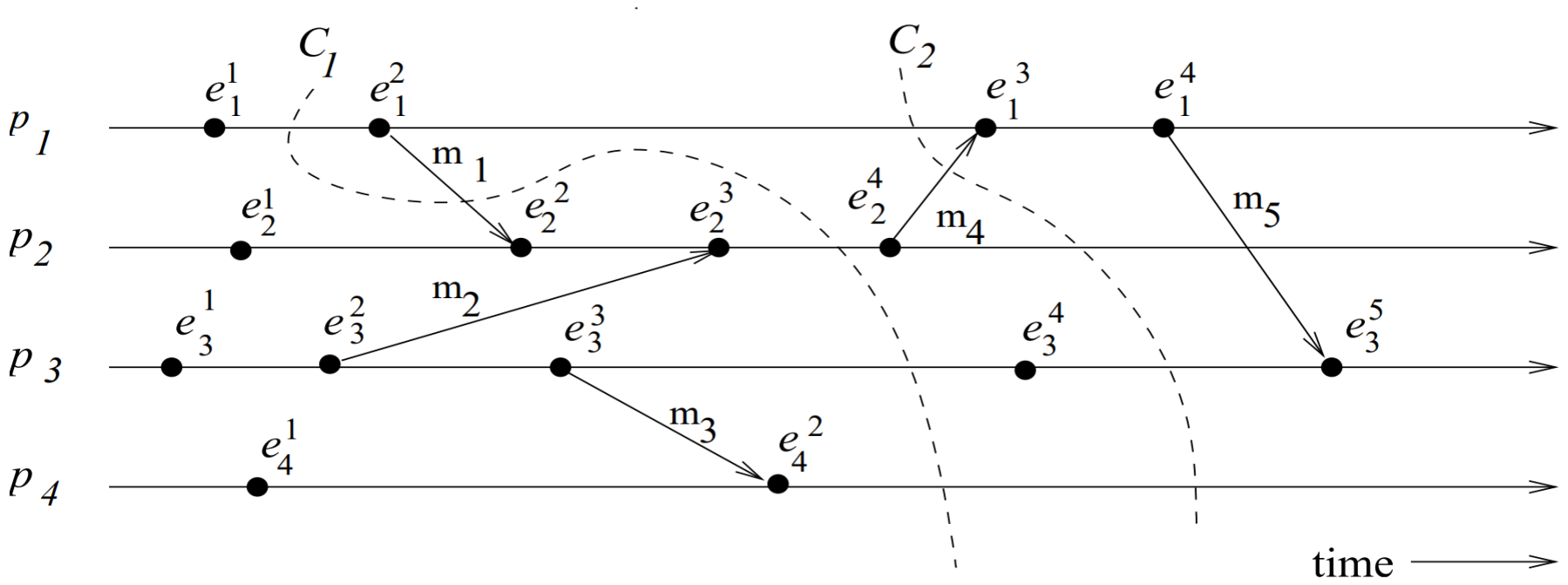
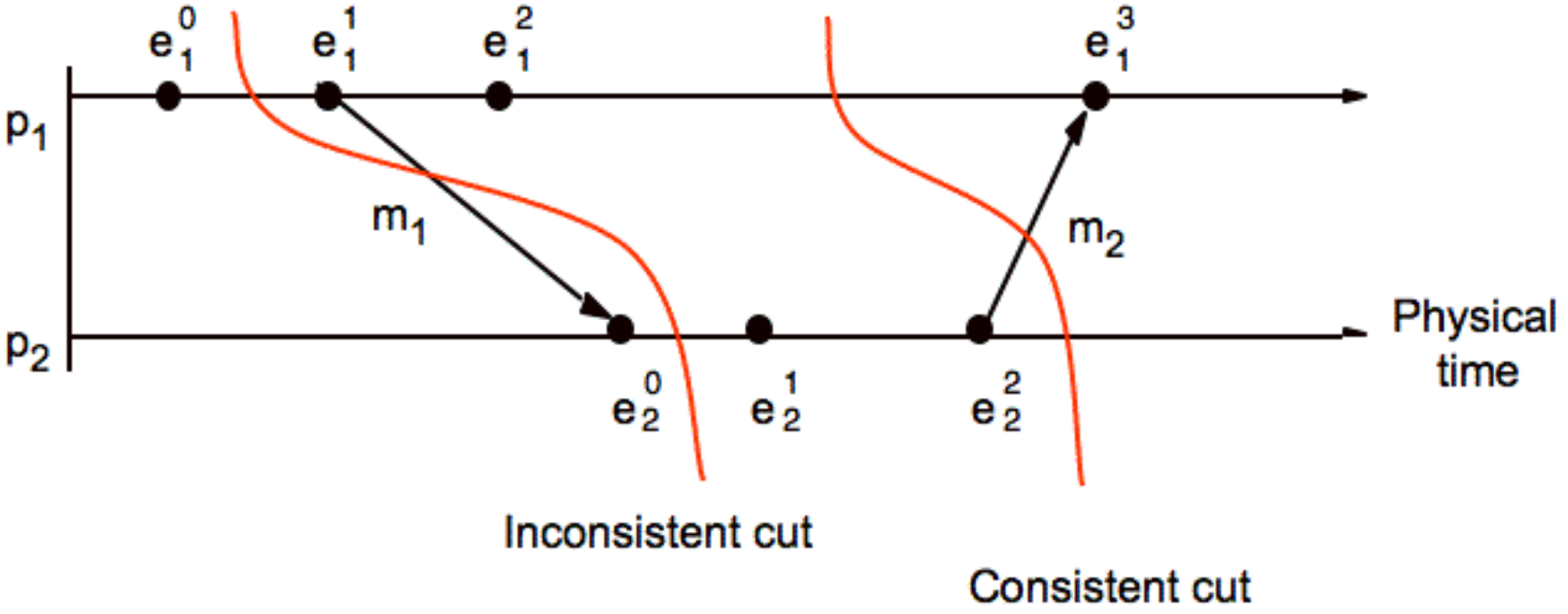


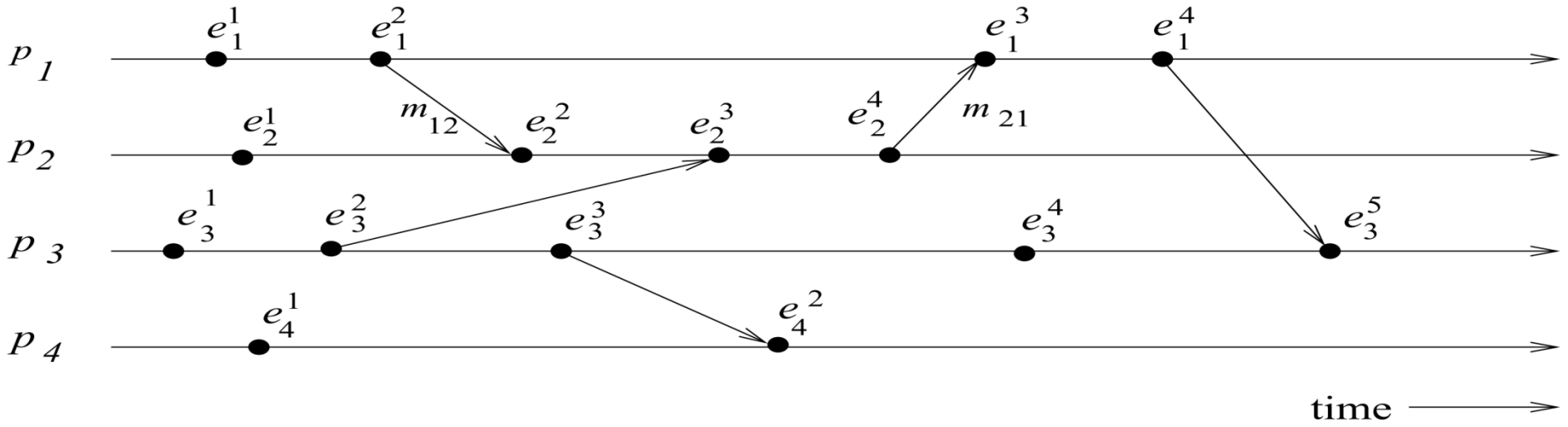
Figure 4.1: An Interpretation in Terms of a Cut.

- Cut C_1 is inconsistent because message m_1 is flowing from the FUTURE to the PAST.
- Cut C_2 is consistent and message m_4 must be captured in the state of channel C_{21}.

Consistent and Inconsistent Cut



Example

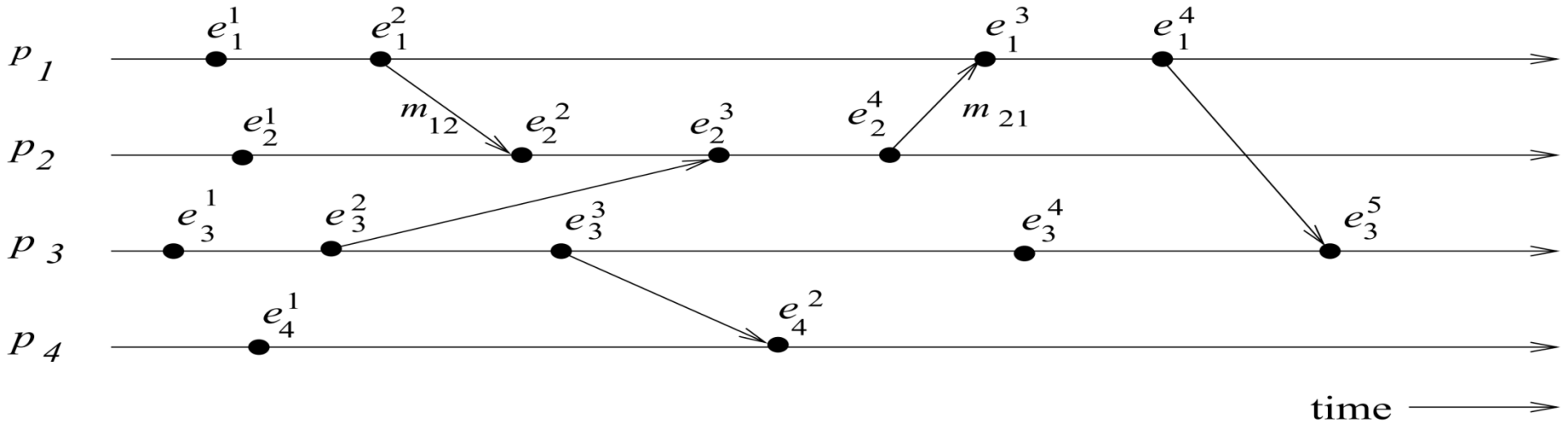


- Which global states are consistent?

1) $\{LS^1_1, LS^3_2, LS^3_3, LS^2_4\}$
2) $\{LS^2_1, LS^4_2, LS^4_3, LS^2_4\}$
3) $\{LS^2_1, LS^1_2, SC^{22}_{12}, LS^3_3, LS^2_4 , SC^{23}_{32}\}$

$SC^{x,y}_{i,j} = \{m_{ij} \mid send(m_{ij}) <= LS^x_i \text{ and } recv(m_{ij}) \not\leq LS^y_j\}$

Example



- Which global states are consistent?

1) $\{LS^1_1, LS^3_2, LS^3_3, LS^2_4\}$ -inconsistent
2) $\{LS^2_1, LS^4_2, LS^4_3, LS^2_4\}$ - consistent
3) $\{LS^2_1, LS^1_2, SC^{22}_{12}, LS^3_3, LS^2_4, SC^{23}_{32}\}$ - consistent

$SC^{x,y}_{i,j} = \{m_{ij} \mid send(m_{ij}) <= LS^x_i \text{ and } recv(m_{ij}) \not< LS^y_j\}$

Difficulty of Taking a Snapshot

- Issue 1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.
  - Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from C1).
  - Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from C2).
- Issue 2: How to determine the instant when a process takes its snapshot.
    - A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

A Distributed Program in Execution

- Let us now model message delivery.
- Several ways are possible
  - Non-FIFO : Message can be reordered in transit
  - FIFO : Messages are NOT reordered in transit.
  - Causal : Slightly stronger than FIFO. Messages destined for the same destination are not reordered. In symbols,

    for any two messages m_{ij} and m_{kj},
    if send(m_{ij}) $\rightarrow$ send(m_{kj}) then recv(m_{ij}) $\rightarrow$ recv(m_{kj}).

  - Causal degenerates to FIFO when sent from same process.

    FIFO $\subset$ Non-FIFO.

Global Snapshots for FIFO channels

– Chandy lamports Algorithm

Note:
A process knows which channel it has received a msg.

A process has an incoming and outgoing msg channel to every other process

Chandy-Lamport's Algorithm

- Uses special marker messages.
- One process acts as initiator, starts the state collection by following the marker sending rule below.

- Marker sending rule for process P:
  - P records its state and
  - For each outgoing channel C from P **on which a marker has not been sent already**, P sends a marker along C before any further message is sent on C

Chandy Lamport's Algorithm contd..

- **When Q receives a marker along a channel C:**
  - **If Q has not recorded its state then Q records its state and also records the state of C as empty; Q then follows the marker sending rule**
  - **If Q has already recorded its state, it records the state of C as the sequence of messages received along C after Q's state was recorded and before Q received the marker along C**

Chandy Lamport's Algorithm contd..

- **If Q has already recorded its state, it records the state of C as the sequence of messages received along C after Q's state was recorded and before Q received the marker along C**

  Say a marker is received by Pj from say Pi where Pj has already recorded its state. Now there can be no messages sent from Pi to Pj from when Pi records the state to when Pi sends out the markers. Hence this msg received was sent before Pi marked its state and received after Pj marked its state. Thus, this msg belongs to the transit messages in Channel (Pi, Pj)

Notable Points

- Markers sent on a channel distinguish messages sent on the channel before the sender recorded its states and the messages sent after the sender recorded its state

- The state collected may not be any state that actually happened in reality, rather a state that “could have” happened

Chandy-Lamport algorithm – FIFO Channels

- This algorithm presented by Mani Chandy and Leslie Lamport works for FIFO channels.

- Why wont it work for non FIFO msgs

Summary

- The Chandy-Lamport algorithm uses a control message, called a ***marker*** whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a marker, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

Summary

- The algorithm can be initiated by any process by executing the “Marker Sending Rule” by which it records its local state and sends a marker on each outgoing channel.
- A process executes the “Marker Receiving Rule” on receiving a marker.
- If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the “Marker Sending Rule” to record its local state.

Termination Condition

- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Correctness and Complexity

- Correctness
- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition C2 is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij} , it acts as follows: If process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij} . Thus, condition C1 is satisfied.
- Complexity
- The recording part of a single instance of the algorithm requires O(e) messages.

Properties of the recorded global state

- The recorded global state may not correspond to any of the global states that occurred during the computation.
- This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.
  - But the system could have passed through the recorded global states in some equivalent executions.
  - The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.
  - Therefore, a recorded global state is useful in detecting stable properties.

HW

- Why will the discussed algorithm not work for non FIFO msgs
- Can you modify or design a new algorithm for non FIFO msgs
- How can we design the algorithm if multiple processes can initiate the algorithm concurrently- Read page 94 (H.W)