Theoretical Foundations

Logical Time and Global States

Introduction

- A DS is a collection of computers that are spatially separated and do not share a common memory
- Processes executing on these computers communicate with one another by exchanging messages over communication channels
- The messages are delivered after an arbitrary transmission delay

Inherent Limitations of a DS

- Absence of Global clock
  - No system-wide common clock
  - Solution
    1. system wide common clock
       - Message transmission delay causes two system to have different time for the same event.
    2. synchronized clocks
       - Physical clock can drift from physical time and drift rate may vary from clock to clock

- Impact of the absence of global time
  - Process scheduling request arrival time is important i.e. temporal ordering of event is important
  - It is difficult to reason about the temporal order of events

Absence of Shared Memory

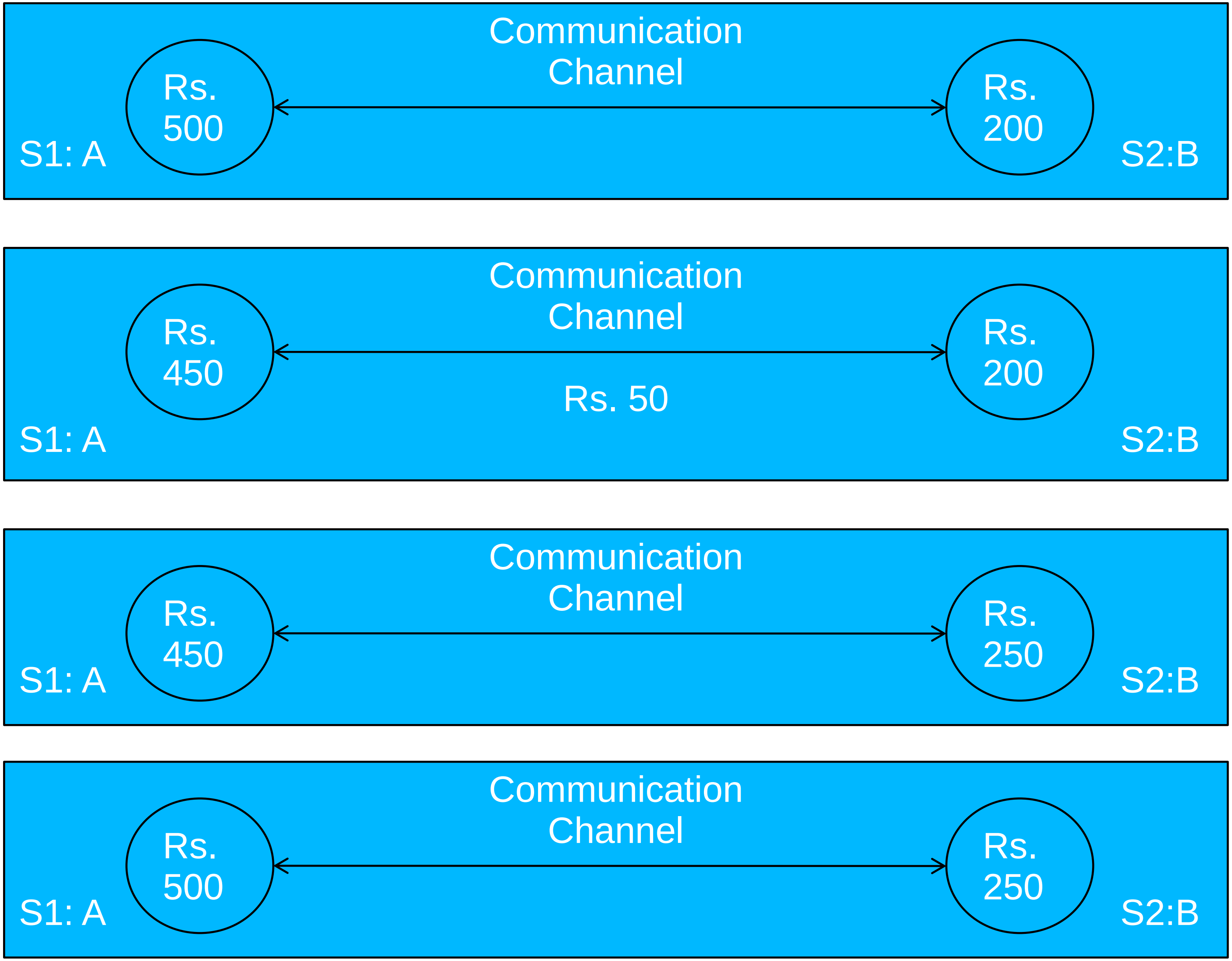
- Up-to-date state of the entire system is not available to individual process
- It is important for
  - Deciding system’s behavior
  - Debugging
  - Recovery from failures etc.
- Obtaining a coherent global state of the system is difficult

Definitions

- **Coherent View:** A view of a system is said to be coherent if all the observations of different processes are made at the same physical time

- **Complete View / Global State:** A complete view encompasses the local views (local states) at all the computers and any messages that are in transit in the DS

A DS with two sites



Lamport’s Logical Clock

- Execution of processes is characterized by a sequence of events
- Execution of procedure could be one event or execution of instruction could be one event
- Sending a message constitutes one event and receiving a message constitutes one event

Lamport's Happened Before Relation (→)

Captures the casual dependencies between events

- a→ b, if **a** and **b** are events in the same process
- a→ b, if **a** is the event of sending a message **m** in a process and **b** is the event of receipt of the same message **m** by another process
- If a→ b, and b→ c, then a→ c

  (→ is a transitive relation)
- Past events influence future events and this influence among causally related events is referred to as causal affects

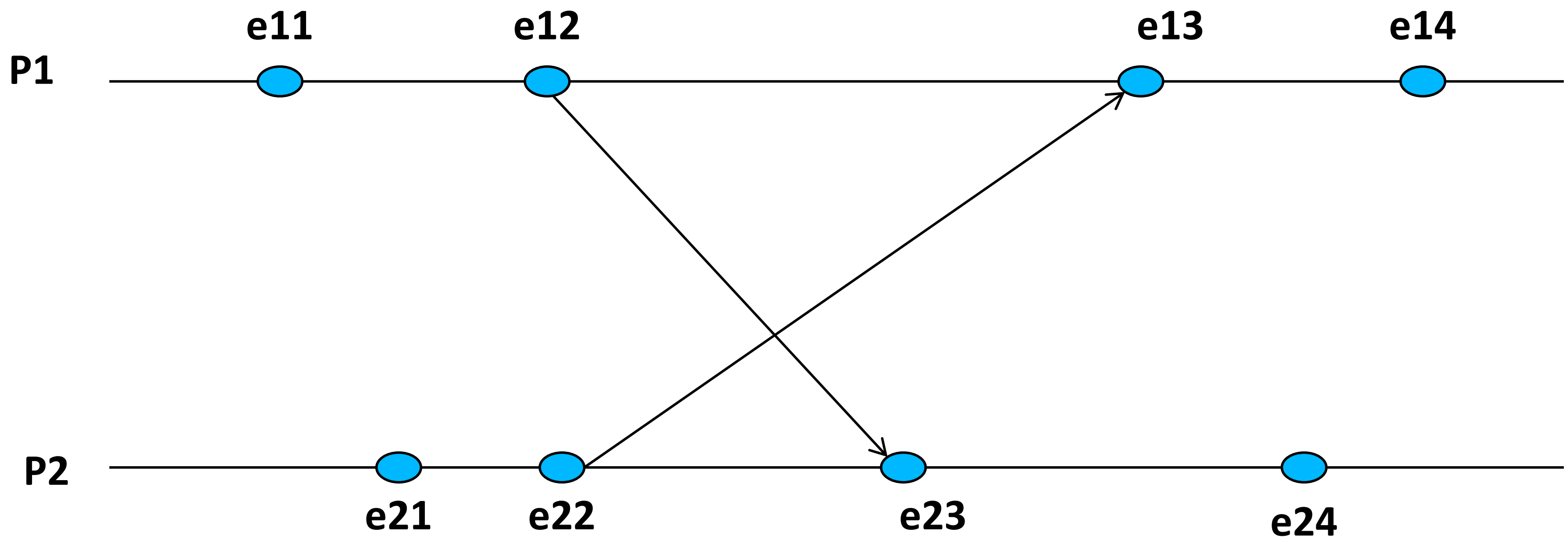
Causally Related Events

Event **a** causally affects event **b** if a→b

Concurrent Events

- Two distinct event **a** and **b** are said to be concurrent if a→b and b→a are false
- denoted by **a || b**
- For any two events **a** and **b** in the system, either a→b, b→a or a || b

Example (space-time diagram)



e11→e12, e11→e13, e11→e14, e11→e23, e11→e24,

e12→e13,e12→e14 ,e12→e23, e12→e24,

e13→e14

e21→e22, e21→e23, e21→e24, e21→e13, e21→e14,

e22→e23,e22→e24 ,e22→e13, e22→e14,

e23→e24

e11 || e21, e14 || e24

Lamport's Logical Clocks

- There is a clock C_i at each process P_i in the system
- The clock C_i can be thought of as a function that assigns a number C_i (a) to any event **a**, called the timestamp of event **a** at P_i
- No relation with physical time
- Monotonically increasing values

Condition satisfied by the system of clocks

- If a→b, then $C(a) < C(b)$
- "→" relation can be realized by using the logical clocks if the following two conditions are met

[C1] For any two events **a** and **b** in a process P_i, if a occurs before b, then

$$C_i (a) < C_i (b)$$

[C2] If **a** is the event of sending a message **m** in process P_i and **b** is the event of receiving the same message **m** at process P_j, then

$$C_i (a) < C_j (b)$$

Implementation Rules for Lamport's Logical clock

Following rules for the clocks guarantee that the clocks satisfy the correctness conditions C1 and C2

[IR1] Clock C_i is incremented between any two successive events in process P_i :

$$C_i := C_i + d \quad (d>0)$$

Usually **d** has the value **1**

[IR2] If **a** is the event of sending a message **m** in process P_i , then message **m** is assigned a timestamp $t_m = C_i$ (a). On receiving the same message **m** by process P_j , C_j is set by following rule:

$$C_j := \max(C_j, t_m + d) \quad (d>0)$$

Lamport's happened before relation

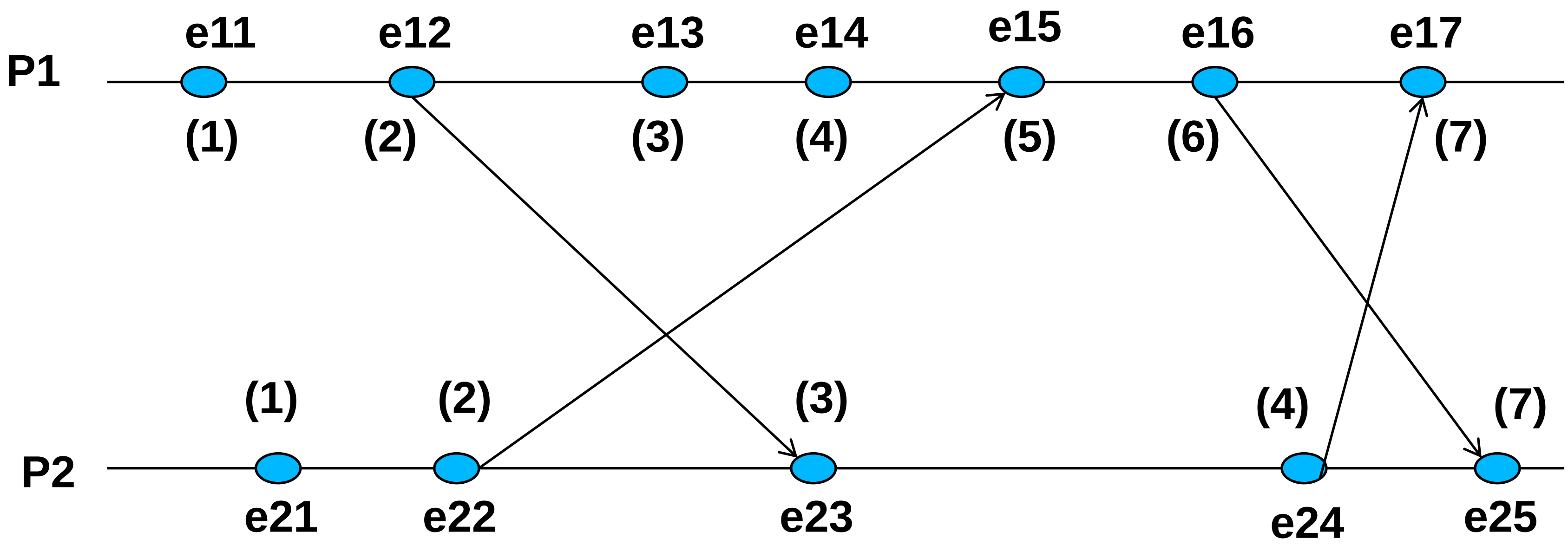
- irreflexive partial order among the events
- Total ordering is possible (denoted by =>)
- If a is any event at process Pi and b is any event at process Pj then a=>b if any only if either:

C_i (a) < C_j (b) or

C_i (a) = C_j (b) and P_i « P_j

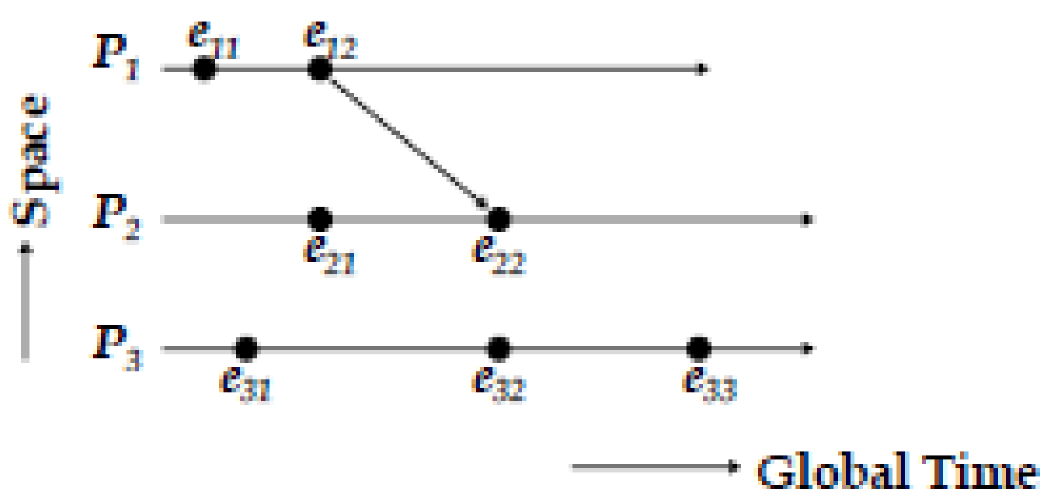
where « is any arbitrary relation that totally orders the processes to break ties.

Lamport's Logical Clock Example



Limitations of Lamport's Logical Clock

- If $a \rightarrow b$ then $C(a)<C(b)$
- However, the reverse is not necessarily true
- Events **a** and **b** may or may not be causally related
- $C(e_{11})<C(e_{22})$ and $C(e_{11}) < C(e_{32})$.
- $e_{11} \rightarrow e_{22}$ but $e_{11} \rightarrow e_{32}$ is not true



Vector Clocks

- Each process P_i is equipped with a clock C_i, which is an integer vector of length **n**.
- The clock C_i can be thought of as a function that assigns a vector $C_i(a)$ to any event **a**
- $C_i(a)$ is referred to as the timestamp of event **a** at P_i
- $C_i[i]$ corresponds to Pi's own logical clock time
- $C_i[j]$ $(j \neq i)$ is P_i's best guess of the logical time at P_j's own logical time

Implementation Rules for Vector Clocks

[IR1] Clock C_i is incremented between any two successive events in process P_i :

$$C_i [i] := C_i [i] + d \quad (d>0)$$

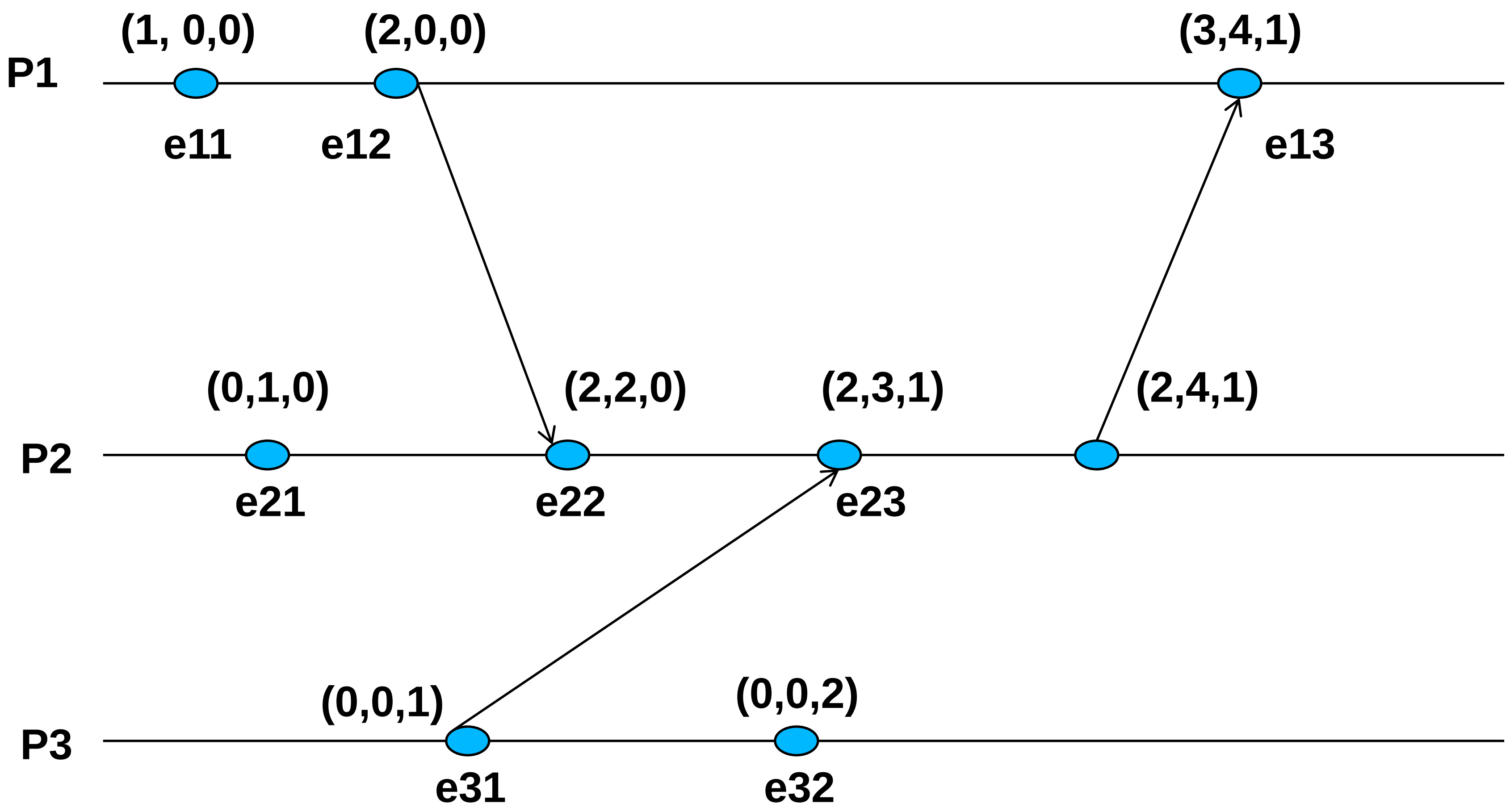
Usually **d** has the value **1**

[IR2] If **a** is the event of sending a message **m** in process P_i , then message **m** is assigned a vector timestamp $t_m = C_i$ (a); on receiving the same message **m** by process P_j , C_j is updated by following rule:

$$\forall k, C_j[k] := \max(C_j[k], t_m[k])$$

On receipt of messages, a process learns about the more recent clock values of the rest of the processes in the system.

Vector Clock Example



Vector Timestamps Comparison

For any two vector timestamps t^a and t^b of events **a** and **b** respectively

Equal:	$t^a = t^b$	iff	$\forall i,$	$t^a[i] = t^b[i]$
Not Equal:	$t^a \neq t^b$	iff	$\exists i,$	$t^a[i] \neq t^b[i]$
Less than of Equal:	$t^a \leq t^b$	iff	$\forall i,$	$t^a[i] \leq t^b[i]$
Not Less than or Equal to:	$t^a \nleq t^b$	iff	$\exists i,$	$t^a[i] > t^b[i]$
Less than:	$t^a < t^b$	iff	$(t^a \leq t^b \ \wedge \ t^a \neq t^b)$	
Not less than:	$t^a \nless t^b$	iff	$\text{not}(t^a \leq t^b \ \wedge \ t^a \neq t^b)$	
Concurrent:	$t^a \mid\mid t^b$	iff	$(\ t^a \nless t^b \ \wedge \ t^b \nless t^a\ )$	

Causally Related Events

- Events a and b are causally related, if $t^a < t^b$ or $t^b < t^a$, otherwise these events are concurrent
- In the system of vector clocks:

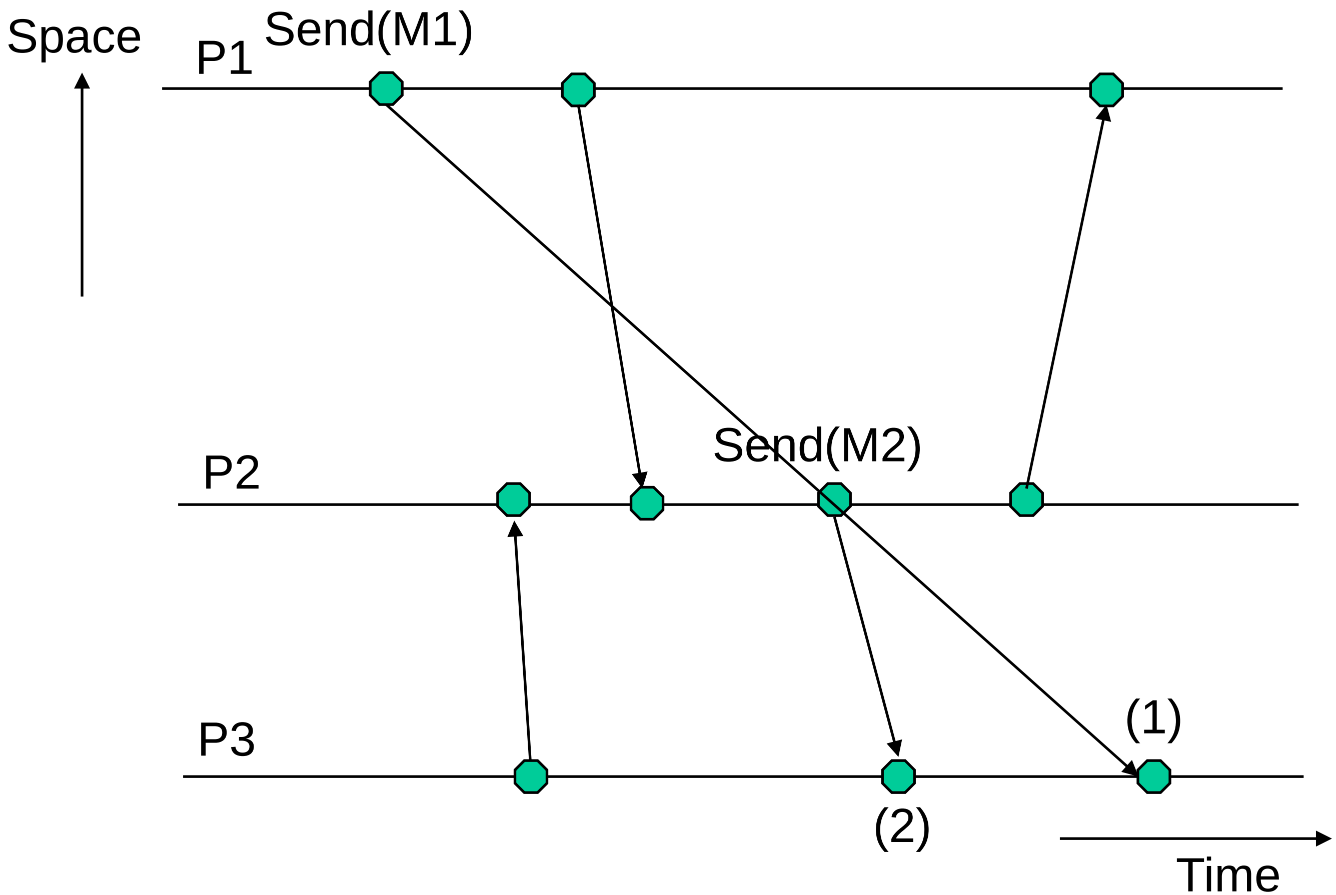
$$a \rightarrow b \quad \text{iff} \quad t^a < t^b$$

- An event **a** can causally affect another event **b** if there exists a path that propagates the (local) time knowledge of event **a** to event **b**

Causal Ordering of Messages

- If Send(M_1) → Send(M_2), then every recipient of both messages M_1 and M_2 must receive M_1 before M_2
- Useful in developing distributed algorithms
- E.g. In the replicated database systems, it is important that every process in charge of updating the replica receives updates in the same order to maintain the consistency of the database

Violation of Causal Ordering of Messages



Use of Vector Clocks for The Causal Ordering of Messages

- Birman-Schiper-Stephenson Protocol **(BSS)**
  - Processes communicate using broadcast messages.
  - Message delivery must be reliable
- Schiper-Eggli-Sandoz Protocol **(SES)**
  - Does not require processes to communicate only through broadcast messages.
  - Message delivery must be reliable

Basic Idea

- Deliver a message to a process only if the message immediately preceding it has been delivered to the process.
- Otherwise the message is not delivered immediately but buffered until the message immediately preceding it is delivered
- A vector accompanying each message contains the necessary information for a process to decide whether there exists a message preceding it

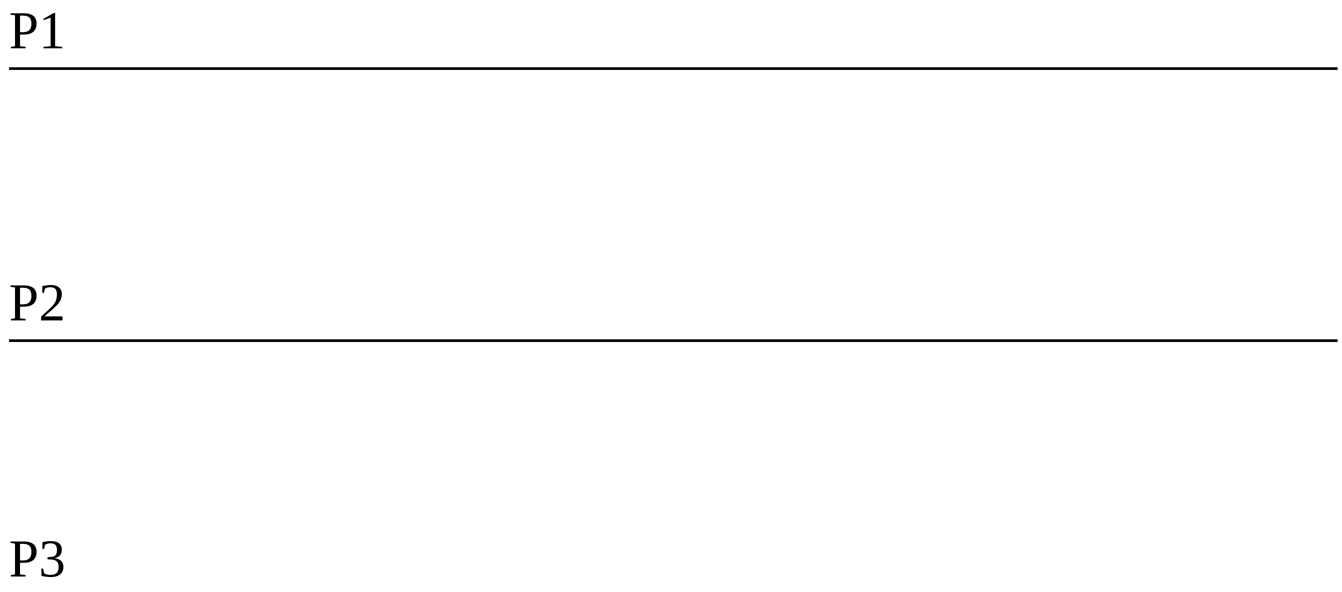
Birman-Schiper-Stephenson Protocol

1. Before broadcasting a message **m**, a process P_i increments the vector time $VTp_i[i]$ and timestamps **m**
2. A process $P_j \neq P_i$, upon receiving message **m** time stamped VT_m from P_i, delays its delivery until both the following conditions are met:
   a. $VTp_j[i] = VT_m[i] - 1$: Ensures that Pj has received all messages from Pi that preced m
   b. $VTp_j[k] \geq VT_m[k]$, for all k in $\{1, 2, \ldots, n\} - \{ i \}$(n is the number of processes) : Ensures that Pj has received all those messages received by Pi before sending m
   c. Delayed messages are queued at each process in a queue that is sorted by vector time of the messages
   d. Concurrent messages are ordered by the time of their receipt

Birman-Schiper-Stephenson Protocol

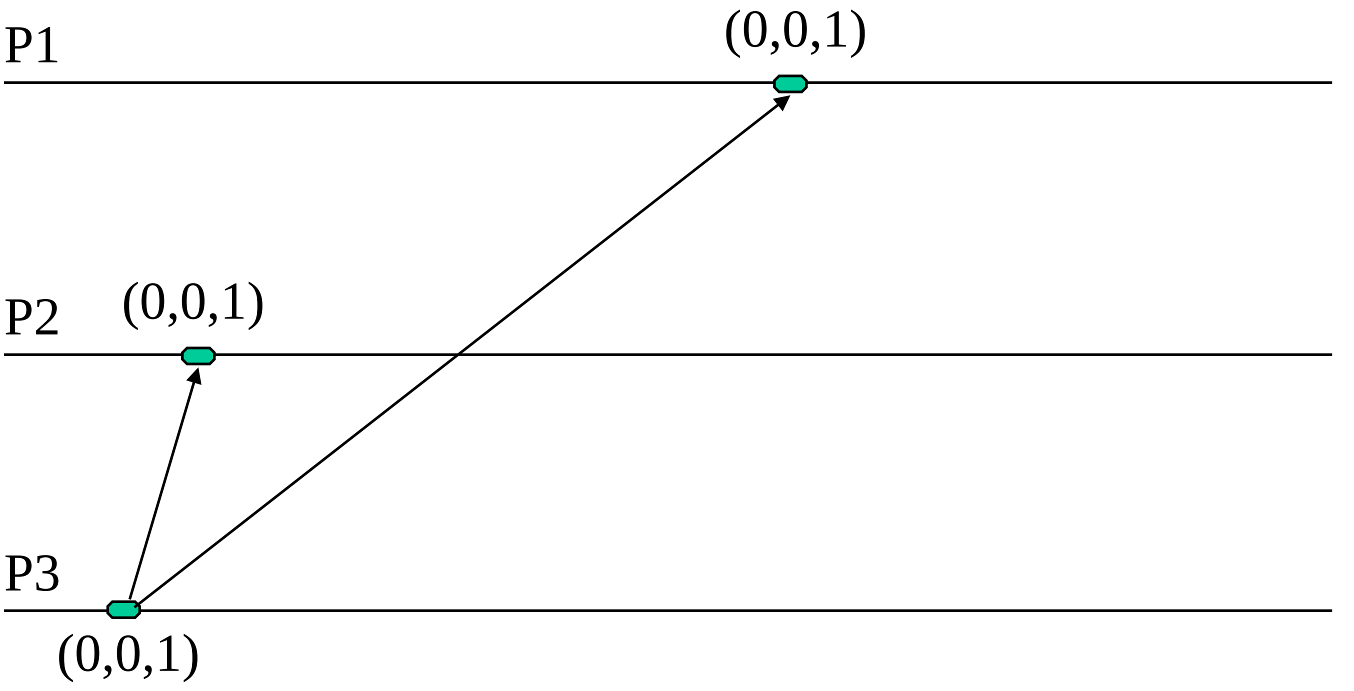
3. When a message is delivered at a process P_j, VTp_j is updated as per vector clock rule IR2
4. Check buffered messages to see if any can be delivered.

Birman-Schiper-Stephenson Protocol

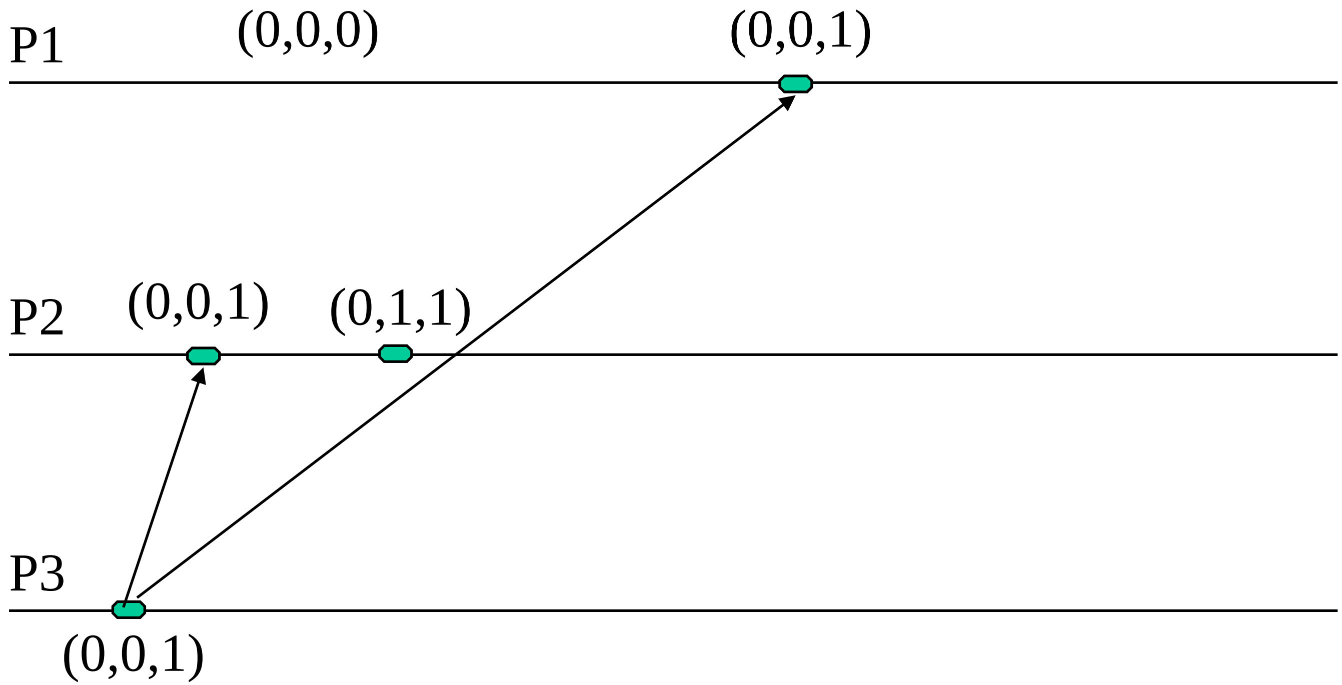


Birman-Schiper-Stephenson Protocol

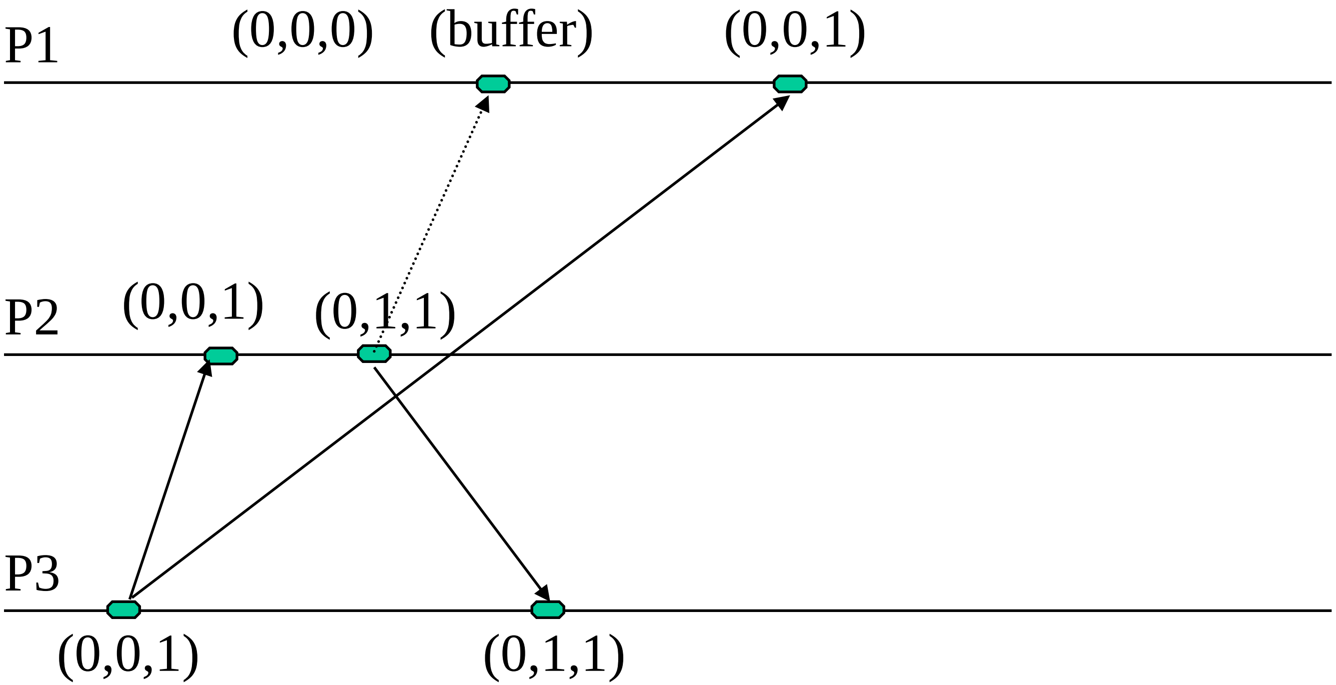
Birman-Schiper-Stephenson Protocol



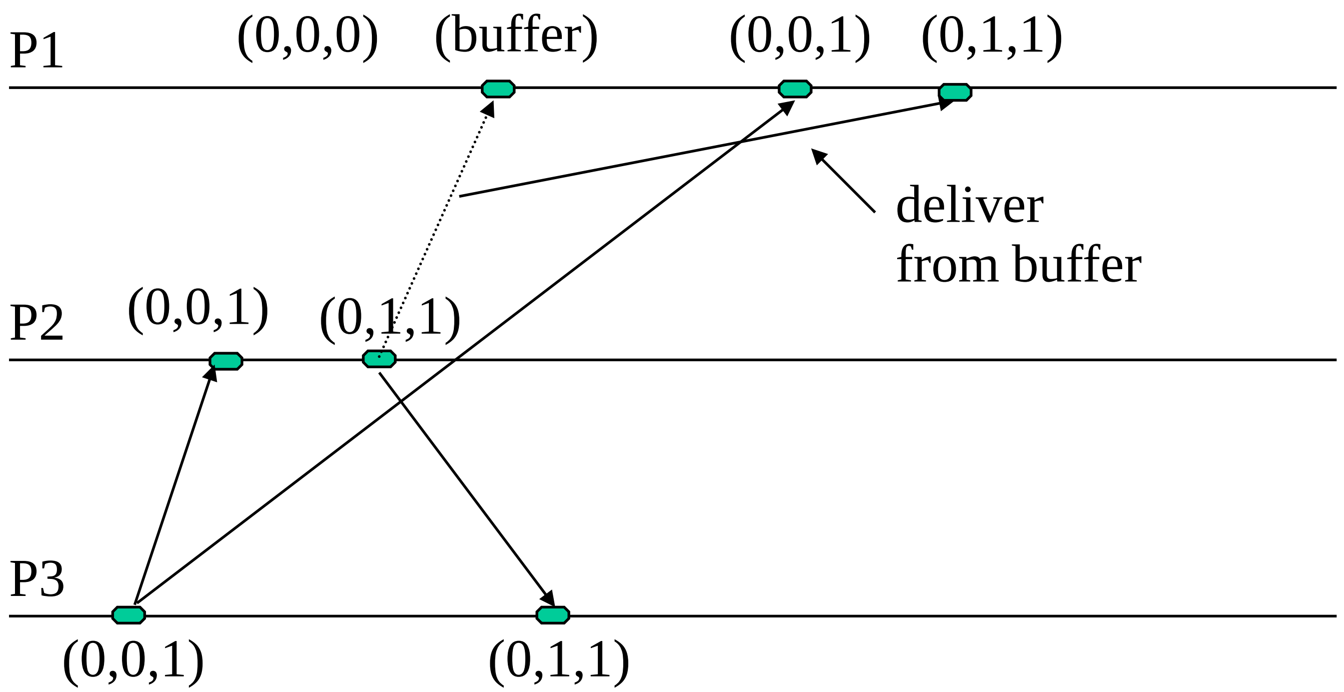
Birman-Schiper-Stephenson Protocol



Birman-Schiper-Stephenson Protocol



Birman-Schiper-Stephenson Protocol



Schiper-Eggli-Sandoz Protocol

Data Structures and Notations:

- Each process P maintains a vector denoted by V_P of size (N-1), where N is the number of processes in the system.
- An element of V_P is an ordered pair (P’,t) where P’ is the ID of the destination process of a message and t is a vector timestamp.
- The communication channels can be non-FIFO
- t_M = logical time at the sending of message M
- tp_i = present/current logical time at process P_i

Schiper-Eggli-Sandoz Protocol

<u>Sending of message M from process P_1 to process P_2:</u>

- Send message M (timestamped t_M) along with V_P_1 to process P_2
- Insert pair (P_2, t_M) into V_P_1.
- If V_P_1 contains a pair (P_2, t), it simply gets overwritten by the new pair (P_2, t_M)
- Note that the pair (P_2, t_M) was not sent to P_2
- Any future message carrying the pair (P_2, t_M) cannot be delivered to P_2 until $t_M < t_{P2}$

Schiper-Eggli-Sandoz Protocol

Arrival of a message M at process P_2:

If V_M (the vector with message M) does not contain any pair (P_2,t) **Then**

Message can be delivered

Else /* A pair (P_2,t) exists in V_M */

If (not (t < tp_2)) **Then**

the message cannot be delivered

/* It is buffered for later delivery */

Else

the message can be delivered

Endif

Endif

Schiper-Eggli-Sandoz Protocol

If message M can be delivered at process P_2 then:

Step-1: Merge V_M accompanying M with V_P_2 in the following manner:

a) If

$$(\exists(P,t) \in \mathrm{V\_M}, \text{such that } \mathrm{P} \neq \mathrm{P_2}) \text{ and } (\forall(\mathrm{P'},\mathrm{t}) \in \mathrm{V\_P_2}, \mathrm{P'} \neq \mathrm{P})$$

then

insert (P, t) into V_P_2

This rule performs the following: if there is no entry for process P in V_P_2, and V_M contains an entry for process P, insert that entry into V_P_2

Schiper-Eggli-Sandoz Protocol

b) If

$$\forall P, P \neq P_2, if\,((P,t) \in \mathrm{V\_M}) \wedge ((P,t') \in \mathrm{V\_P_2})$$

Then

(P, t’) Є V_P$_2$ can be substituted by the pair (P, t_{sup}) where t_{sup} is such that

$$\forall i, t_{\sup}[i] = \max(t[i], t'[i])$$

This rule is simply the IR2 of vector clocks

Above two actions satisfies following conditions:

i. No message can be delivered to P as long as t’ > t_P

ii. No message can be delivered to P as long as t > t_P

Schiper-Eggli-Sandoz Protocol

Step-2: Update site P_2's logical clock

Step-3: Check for the buffered messages that con now be delivered since local clock has been updated

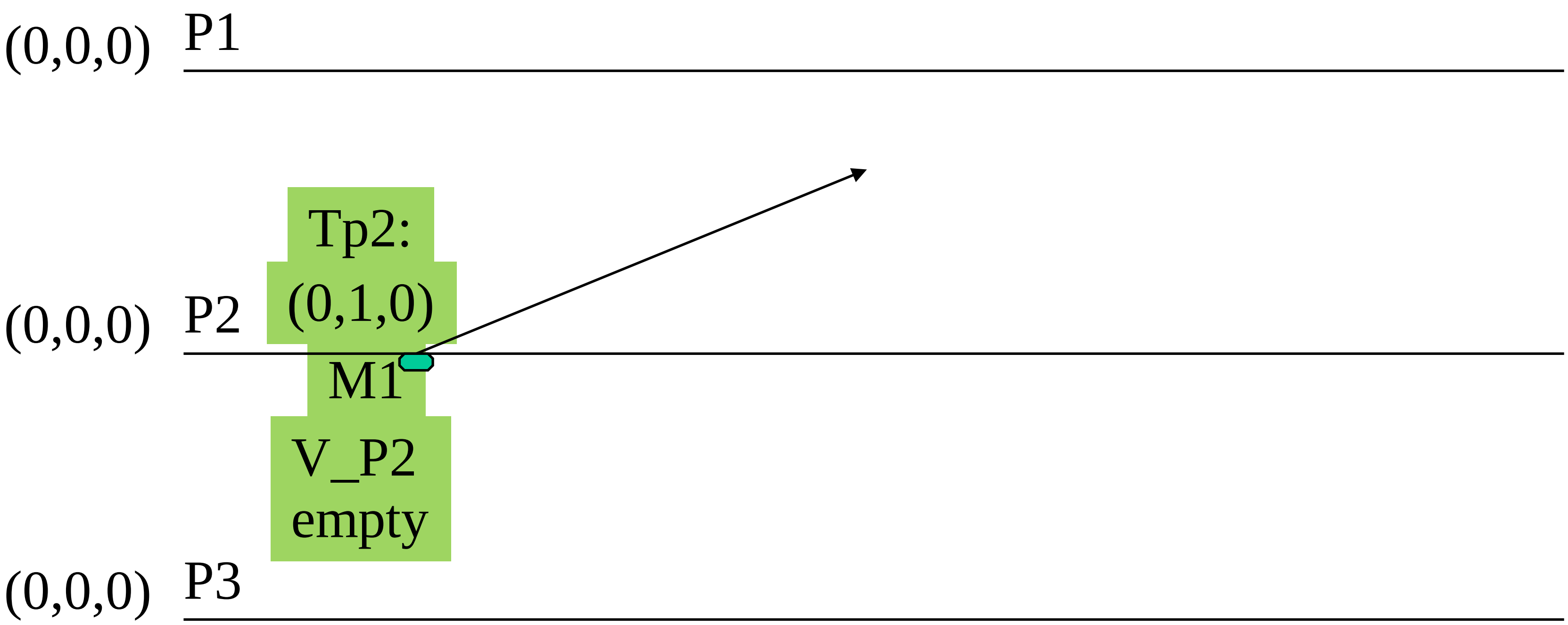
SES Buffering Example

(0,0,0) P1

(0,0,0) P2

(0,0,0) P3

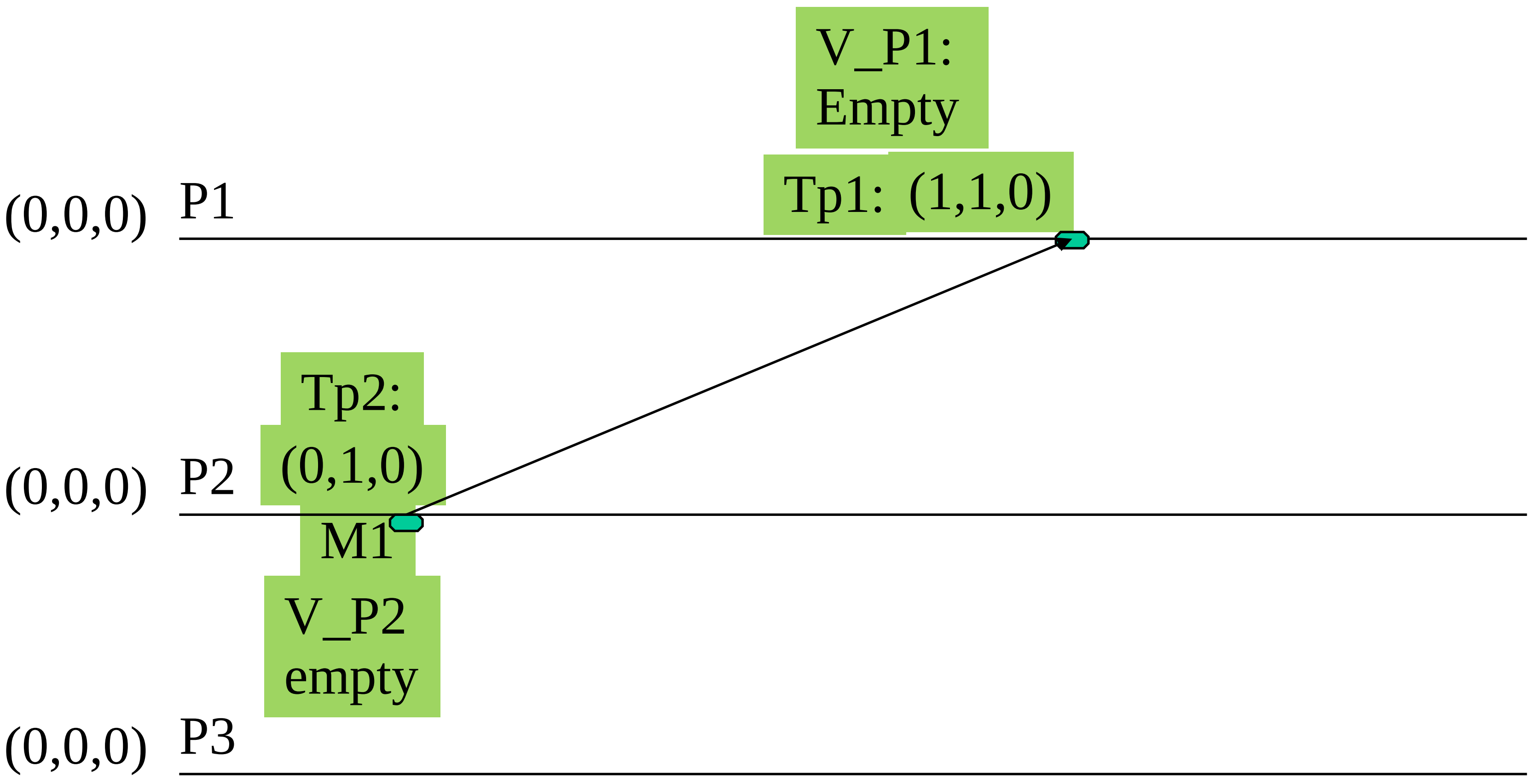
SES Buffering Example



M1 from P2 to P1

Send M1 + Tm (=<0,1,0>) + Empty V_P2

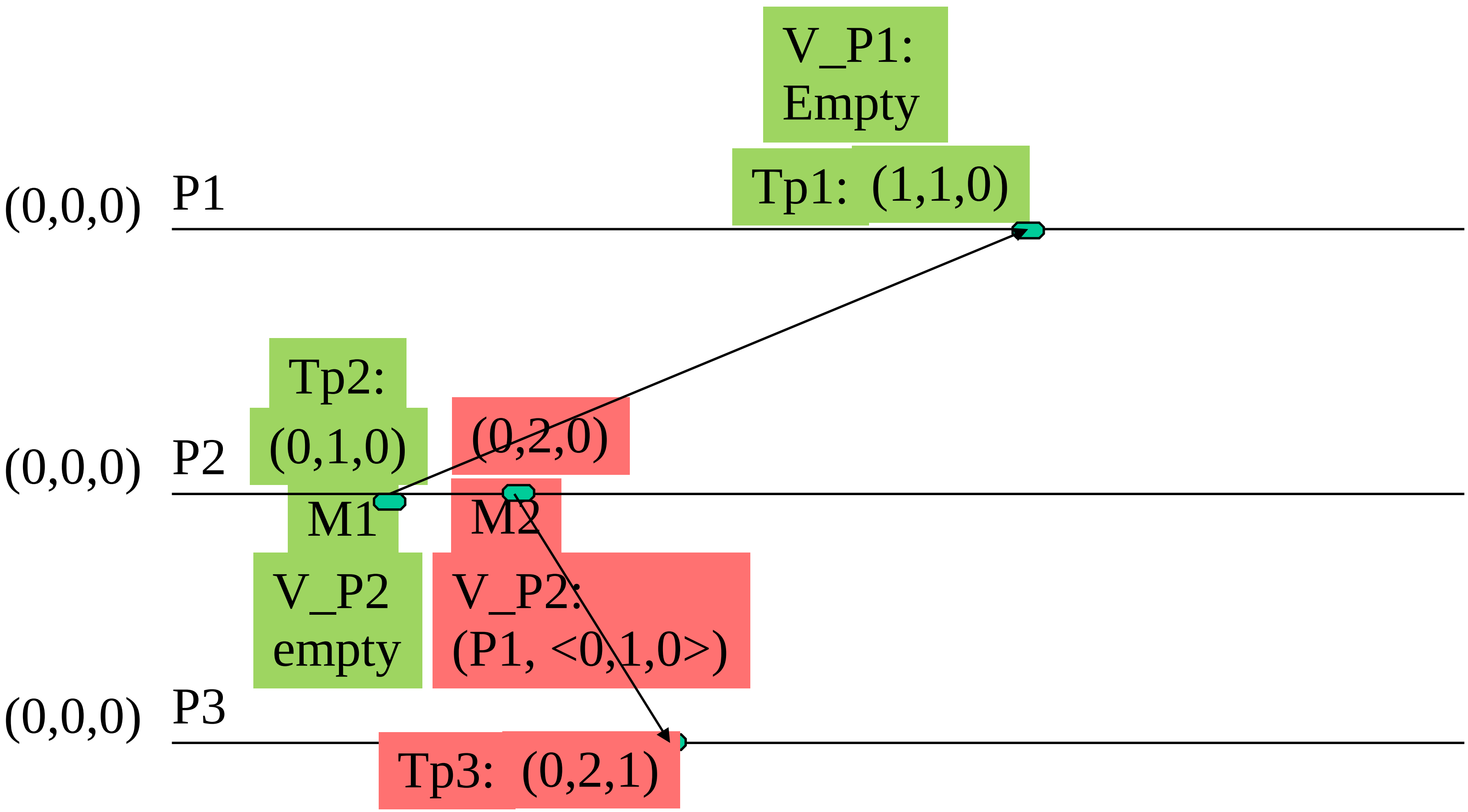
SES Buffering Example



<u>When M1 is received by P1</u>

Tp1 becomes <1,1,0>, by rules 1 and 2 of vector clock.

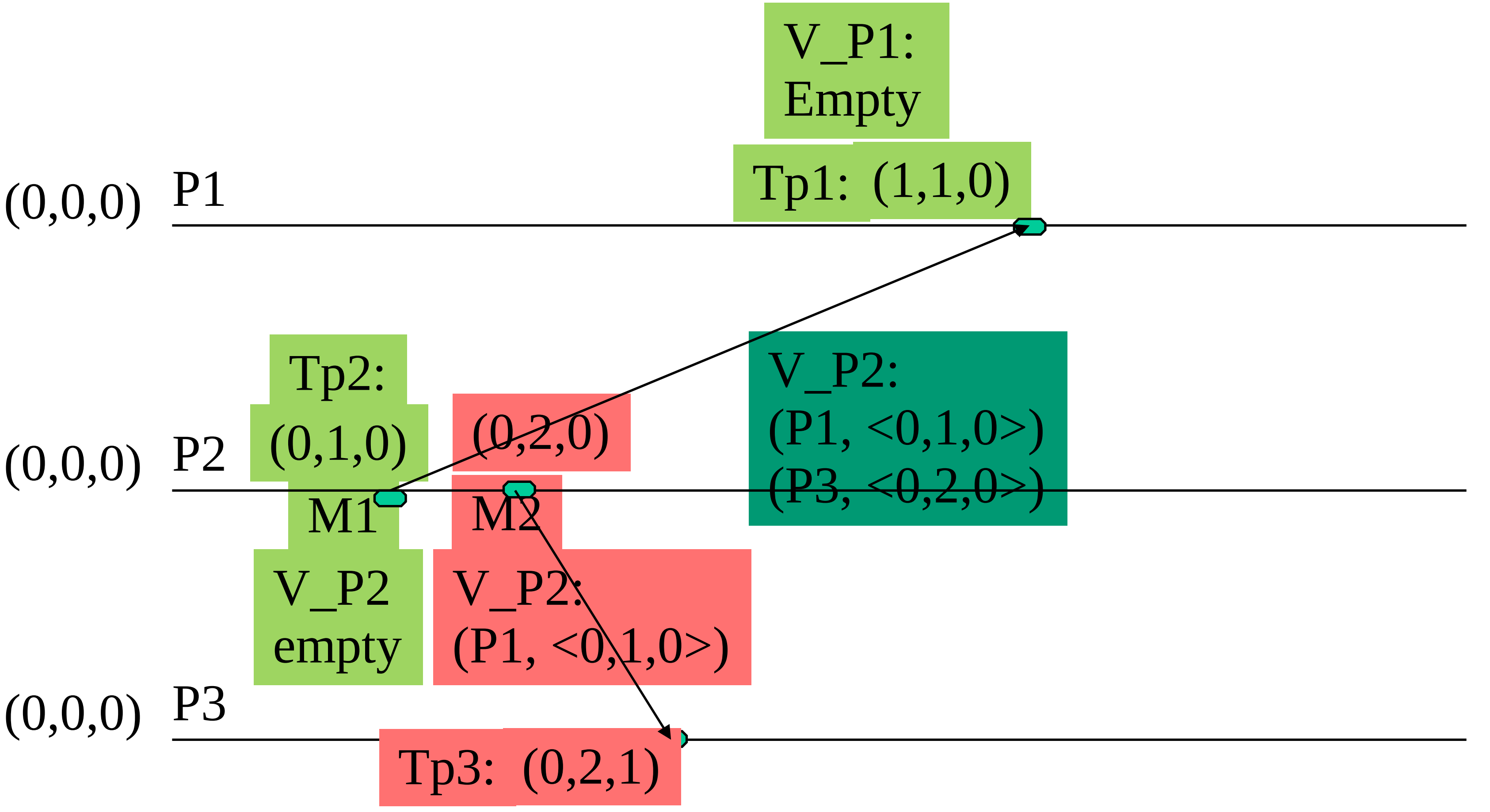
SES Buffering Example



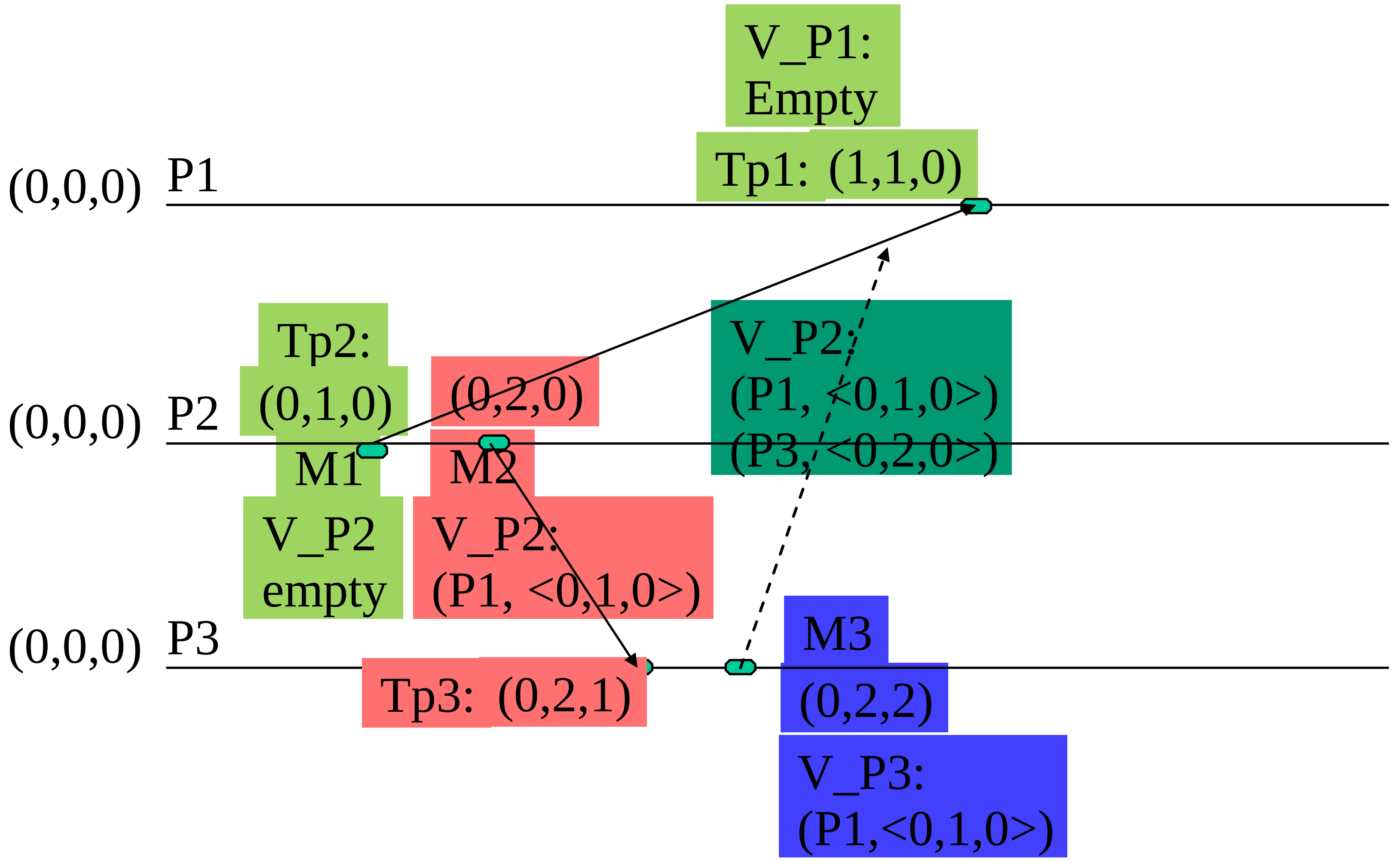
<u>M2 from P2 to P3</u>

M2 + Tm (<0, 2, 0>) + (P1, <0,1,0>)

SES Buffering Example



SES Buffering Example



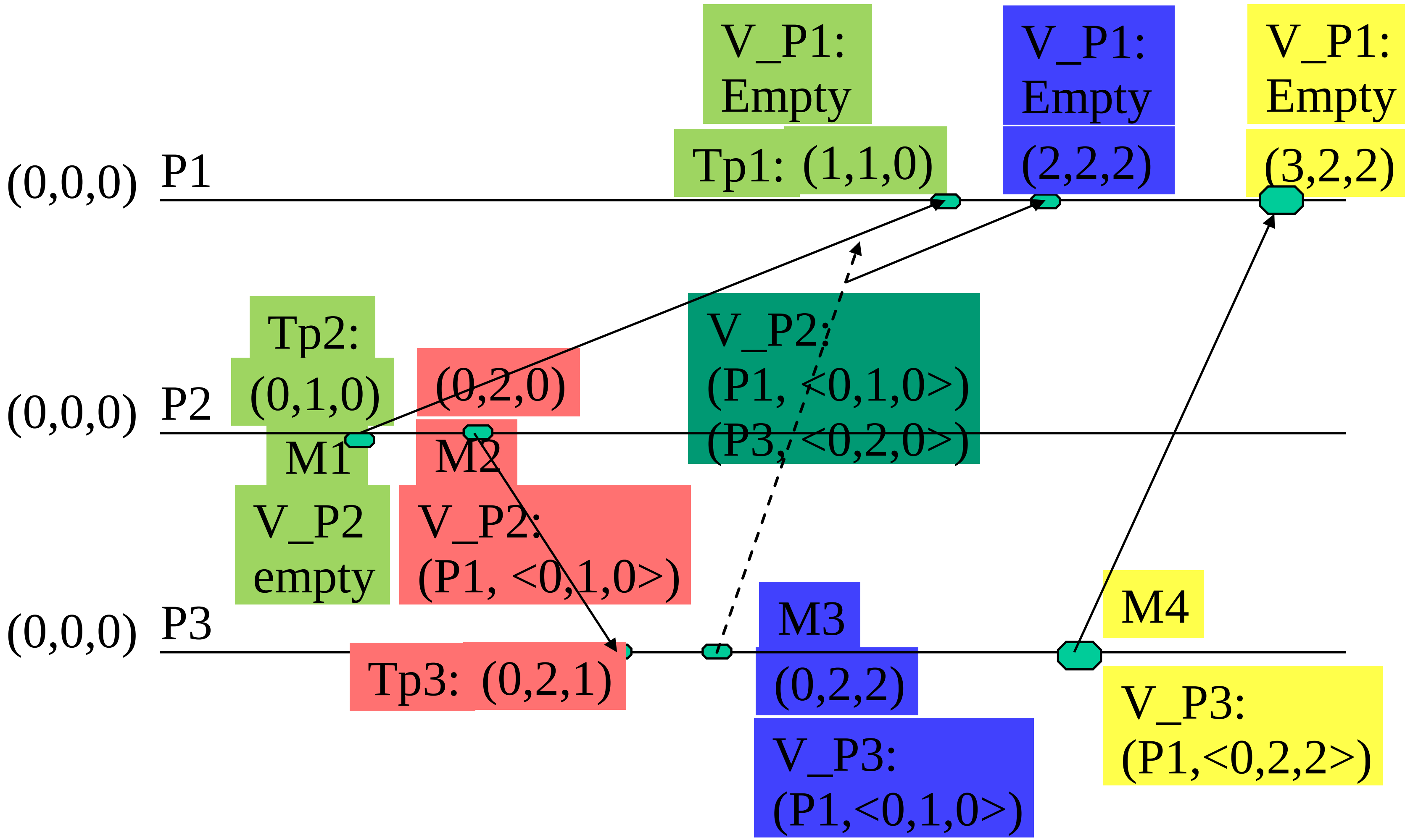
<u>M3 from P3 to P1</u>

M3 + <0,2,2> + (P1, <0,1,0>)

<u>M3 gets buffered because</u>

Tp1 is <0,0,0>, t in (P1, t) is <0,1,0> & so Tp1 < t

SES Buffering Example

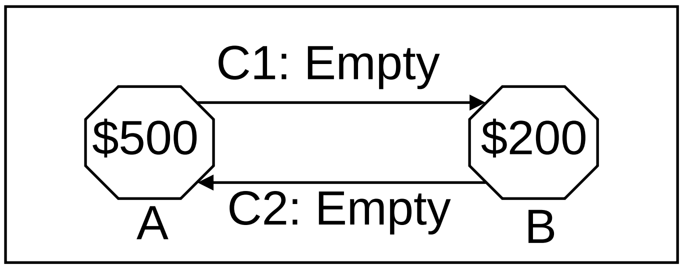


<u>After updating Tp1, P1 checks buffered M3</u>

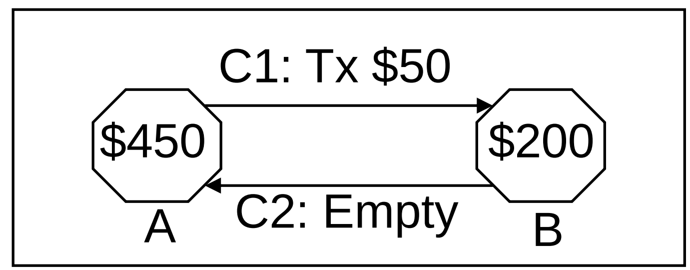
Now, Tp1 > t [in (P1, <0,1,0>] So M3 is delivered.

Global State

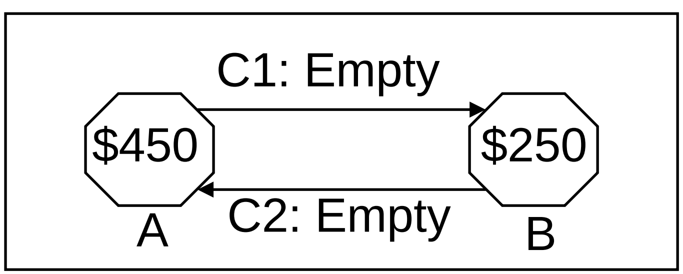
Global State 1



Global State 2



Global State 3



Recording Global State…

- E.g. Global state of A is recorded in (1) and not in (2)
  - State of B, C1, and C2 are recorded in (2)
  - Extra amount of $50 will appear in global state
  - Reason: A's state recorded *before* sending message and C1's state *after* sending message.
- Inconsistent global state if n < n', where
  - n is number of messages sent by A along the channel before A's state was recorded
  - n' is number of messages sent by A along the channel before channel's state was recorded.
- Consistent global state: n = n'

Recording Global State…

- Similarly, for consistency m = m’
  - m’: no. of messages received along channel before B’s state recording
  - m: no. of messages received along channel by B before channel’s state was recorded.
- Also, n’ >= m, as in no system no. of messages sent along the channel be less than that received
- Hence, **n >= m**
- Consistent global state should satisfy the above equation

Definitions

Local State: For a site S_i, its local state at a given time is the local context of the distributed application

- LS_i : local state at S_i
- $send(m_{ij})$: message M sent from S_i to S_j
- $rec(m_{ij})$: message M received by S_j, from S_i
- $time(x)$: Time of event x
- $transit(LS_i, LS_j)$: set of messages sent/recorded at LS_i and not received/recorded at LS_j

Definitions

- For a message m_{ij}, sent by S_i to S_j, we say that
  - $send(m_{ij}) \in LS_i$ iff $time(send(m_{ij})) < time(LS_i)$
  - $rec(m_{ij}) \in LS_j$ iff $time(rec(m_{ij})) < time(LS_j)$
- For the local states LS_i and LS_j of any two sites S_i and S_j, we define two sets of messages:
- **Transit:**

  $transit(Ls_{i,} LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \land rec(m_{ij}) \notin LS_j\}$
- **Inconsistent:**

  $inconsistent(Ls_{i,} LS_j) = \{m_{ij} \mid send(m_{ij}) \notin LS_i \land rec(m_{ij}) \in LS_j\}$

Definitions

Global State:

- A global state GS, of a system is a collection of the local states of its sites;
- That is GS = { LS_1, LS_2, …, LS_n }

Consistent Global State:

- A global state GS = { LS_1, LS_2, …, LS_n } is consistent iff,

$$\forall i, \forall j : 1 \leq i, j \leq n :: inconsistent(LS_i, LS_j) = \Phi$$

- Thus, for every received message a corresponding send event is recorded in the global state.
- For inconsistent global state, there is at least one message whose received event is recorded but its send event is not recorded in the global state

Definitions

Transitless Global State:

- A global state is transitless If and only if,

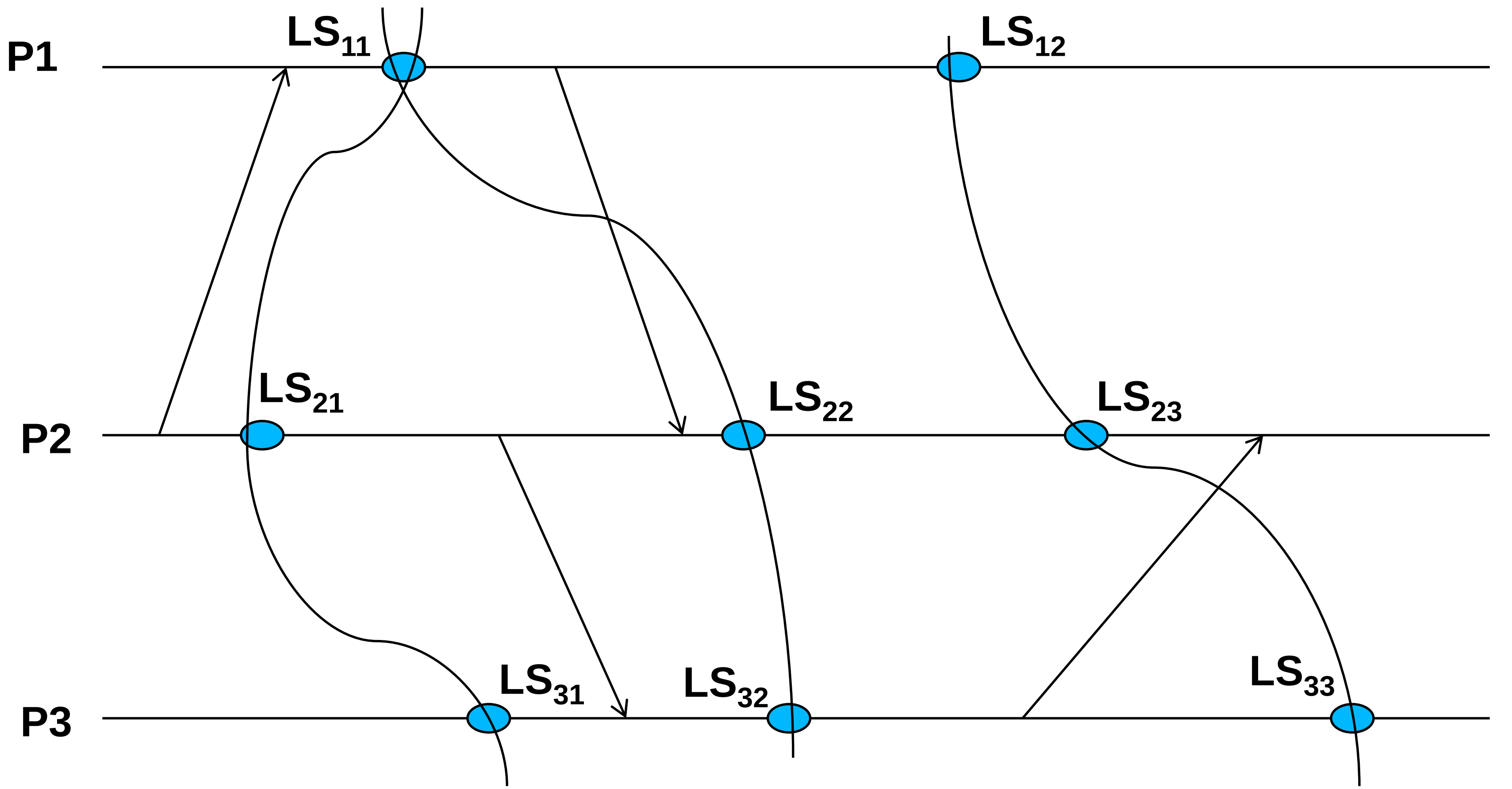
$$\forall i, \forall j : 1 \leq i, j \leq n :: transit(LS_i, LS_j) = \Phi$$

- Thus, all channels are empty in a transitless global state

Strongly Consistent Global State:

- A global state is said to be strongly consistent if it is consistent and transitless

Consistent and Inconsistent Global States



Consistent Global State : $\{LS_{12}, LS_{23}, LS_{33}\}$

Inconsistent Global State : $\{LS_{11}, LS_{22}, LS_{32}\}$

Strongly consistent Global State : $\{LS_{11}, LS_{21}, LS_{31}\}$

Chandy-Lamport's Global State Recording Algorithm

- The idea behind this algorithm is that we can record a consistent state of the global system if we know that all messages that have been sent by one process have been received by another.

- This is accomplished by the use of a Marker (sort of dummy message, with no effect on the functions of processes.) which traverses the distributed system across all channels.

- This Marker, in turn, causes each process to record a snapshot of itself and, eventually, of the entire system

Chandy-Lamport’s Global State Recording Algorithm

Assumptions

- There are a finite number of processes and communications channels.
- Communication channels have infinite buffers that are error free.
- Messages on a channel are received in the same order as they are sent.
- Processes in the distributed system do not share memory or clocks.

Chandy-Lamport's Global State Recording Algorithm

Marker Sending Rule for a process P

1. P records its state
2. For each outgoing channel C from P on which a marker has not been already sent, P sends a marker along C before P sends further messages along C

Chandy-Lamport's Global State Recording Algorithm

Marker Receiving Rule for a process Q On receiving a marker along a channel C:

If (Q has not recorded its state) **Then**

- Record the state of Q
- Record the state of C as an empty sequence
- Follow the marker sending rule

Else

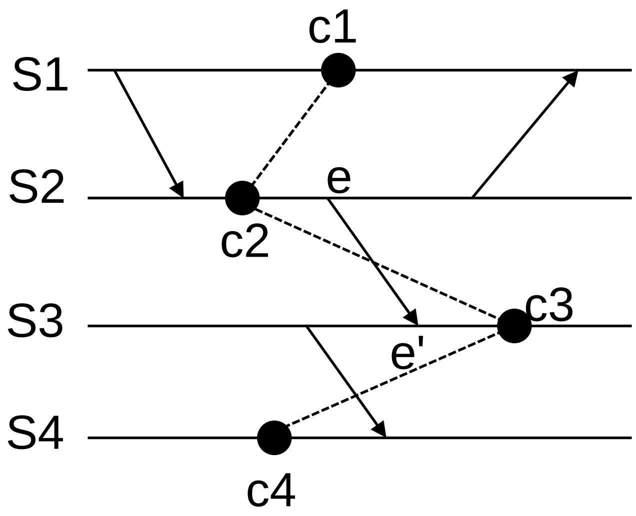
- Record the state of C as the sequence of messages received along C after Q's state was recorded and before Q received the marker along C

Chandy-Lamport's Global State Recording Algorithm

- Marker delineates messages into those that need to be included in the recorded state and those that are not to be recorded in the state
- Algorithm can be initiated by any process by executing the marker sending rule
- Algorithm can be initiated by several processes concurrently with each process getting its own version of a consistent global state.
- Each initiation needs its own unique marker (<pid, seq_no>) and different initiations of same process can be distinguished by a local sequence number
- Each process has to send the information recorded to the initiator of the recording process
- The identification of the initiator process can be easily carried by the marker

Cuts

- Cut :
  - A graphical representation of a global state.
  - Cut is a set C = {c_1, c_2, .., c_n} where c_i is cut event at site S_i.
  - If a cut event c_i at site S_i is S_i ‘s local state at that instant, then clearly a cut denotes a global state of the system



Consistent Cut

- Let e_k denote an event at site S_k, A cut C = $\{c_1, c_2, .., c_n\}$ is a consistent cut iff

$$\forall S_i, \forall S_j, \nexists e_i, \nexists e_j \; such\,that \left(e_i \rightarrow e_j\right) \wedge \left(e_j \rightarrow c_j\right) \wedge \left(e_i \nrightarrow c_i\right)$$

$$Where\; c_i \in C\; and\; c_j \in C$$

- Theorem : A Cut C = $\{c_1, c_2, .. , c_n\}$, is a consistent cut if and only if no two cut events are causally related , that is,

$$\forall c_i \forall c_j \;\; :: \sim \left(c_i \rightarrow c_j\right) \wedge \sim \left(c_j \rightarrow c_i\right)$$

Time of a Cut

- If $C = \{c_1, c_2, .., c_n\}$ is a cut with vector time stamp VTc_i, then the vector time of the cut,

  $VTc = sup(VTc_1, VTc_2, .., VTc_n)$

- sup is a component-wise maximum, i.e.,

  $VTc_i = \max(VTc_1[i], VTc_2[i], .., VTc_n[i])$

- Theorem :
  – A cut is consistent iff $VTc = (VTc_1 [1], VTc_2 [2], .., VTc_n [n])$
  – Proof is trivial

Termination Detection

- Termination
  - completion of the sequence of algorithm. E.g., leader election, deadlock detection, deadlock resolution.
- System Model
  - A process may either be active or idle
  - Only active processes can send messages
  - An active process may become idle at any time
  - An idle process can become active on receiving a computation message
  - A computation is said to have terminated if and only if all the processes are idle and there are no messages in transit.
  - The messages sent by the termination detection algorithm are referred to as control messages

Termination Detection

- Basic Idea
  - One of the cooperating processes monitors the computation and is called the controlling agent
  - Initially, all processes are idle. Weight of controlling agent is 1 (0 for others).
  - **Start of computation:** when message is sent from controller to a process. Weight splits between two into half
  - **Repeat this:** any time a process sends a computation message to another process, split the weights between the two processes
  - **End of computation:** process sends its weight to the controller. Add this weight to that of controller's. (Sending process's weight becomes 0).
  - ***Rule:*** Sum of Weights always 1.
  - ***Termination:*** When weight of controller becomes 1 again.

Huang's Termination Detection Algorithm

- Notations
  - n processes
  - P_i process; without loss of generality, let P_0 be the controlling agent
  - W_i weight of process P_i; initially, $W_0 = 1$ and for all other i, $W_i = 0$.
  - B(DW) computation message with assigned weight DW
  - C(DW) control message sent from process to controlling agent with assigned weight DW

Huang's Termination Detection Algorithm

- **Algorithm**
  - **Rule 1 : P_i sends a computation message to P_j**
    1. Derive W_1 and W_2 such that $W_1 + W_2 = W_i$, $W_1 > 0$, $W_2 > 0$
    2. $W_i = W_1$
    3. Send $B(W_2)$ to P_j
  - **Rule 2 : P_j receives a computation message B(DW) from P_i**
    1. $W_j = W_j + DW$
    2. If P_j is idle, P_j becomes active
  - **Rule 3 : P_i becomes idle**
    1. Send $C(W_i)$ to P_0
    2. $W_i = 0$
  - **Rule 4 : P_0 receives a control message C(DW)**
    1. $W_0 = W_0 + DW$
    2. If $W_0 = 1$, the computation has completed.

Huang’s Algorithm – An Example