TIME IN DISTRIBUTED SYSTEMS - II

DISTRIBUTED SYSTEMS

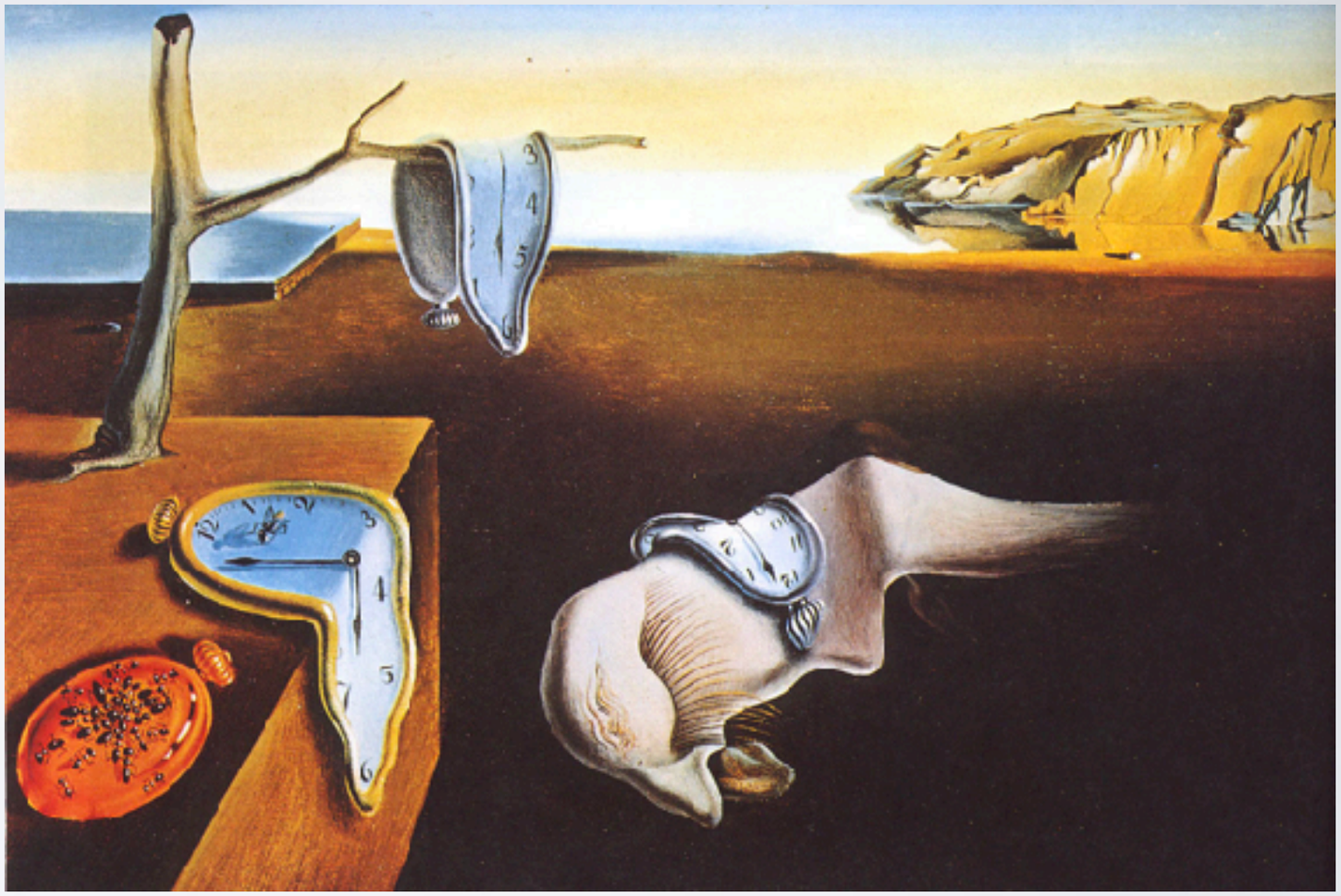
Master of Science in Cyber Security

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

OUTLINE

- The need of logical time
- Happened Before Relationship
- Scalar/logical/Lamport's clocks
- Vector Clocks
- Lamport's ME



The Persistence of Memory, Salvador Dali

LOGICAL CLOCK

Physical clock synchronization algorithms try to coordinate distributed clocks to reach a common value

- They are based on the estimation of transmission delay but in several systems it could be hard to obtain a “good” estimation.
- For several applications it is not important when things happened but in which order they happened

However in a Distributed System, each system has its own “logical clock”

- If clocks are not aligned it is not possible to order events generated by different processes

ORDERING MATTERS

Scenario: Distributed chat used in a mission critical workplace.

- Why distributed?
  - No central point of failure;
  - No server can stores encrypted copies of messages, no metadata records.
- Asynchronous model: why?
  - Mission critical, it has to work even if the system loses synchrony (e.g., adversary in some routers or broadcast storm).
- We cannot synchronise and we cannot timestamp with our software clocks.

ORDERING MATTERS

Real order of messages

Alice: If the price of Apple drops, what we do?

Mike: We have to sell Apple and buy Amazon

Alice: Who will do it?

Bob's supervisor: Bob will do

ORDERING MATTERS

Real order of messages

Alice: If the price of Apple drops, what we do?

Mike: We have to sell Apple and buy Amazon

Alice: Who will do it?

Bob's supervisor: Bob will do

Bob's local view

Mike: We have to sell Apple and buy Amazon

Bob's supervisor: Bob will do

ORDERING MATTERS

Real order of messages

Alice: If the price of Apple drops, what we do?

Mike: We have to sell Apple and buy Amazon

Alice: Who will do it?

Bob's supervisor: Bob will do

Bob's local view

Mike: We have to sell Apple and buy Amazon

Bob's supervisor: Bob will do

We have to avoid this. What we need?

ORDERING MATTERS

Real order of messages

Alice: If the price of Apple drops, what we do?

Mike: We have to sell Apple and buy Amazon

Alice: Who will do it?

Bob's supervisor: Bob will do

Bob's local view

Mike: We have to sell Apple and buy Amazon

Bob's supervisor: Bob will do

We have to avoid this. What we need?

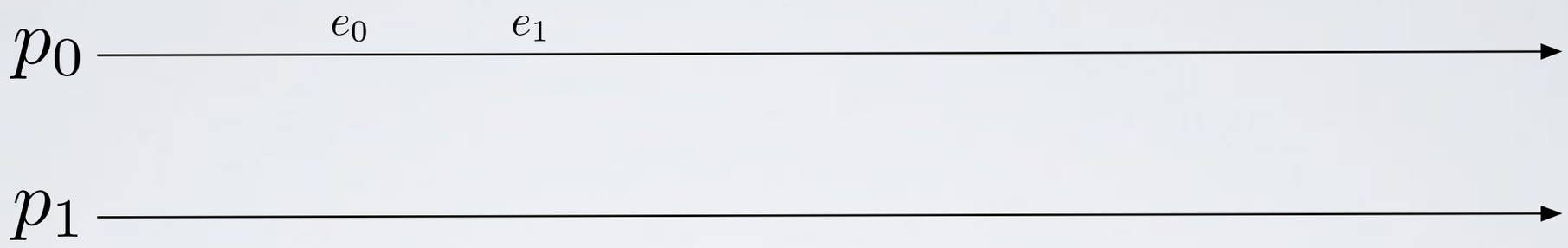
Bob has to know, when receives the message from Mike, that there is something that causally generated it

OUR GOAL

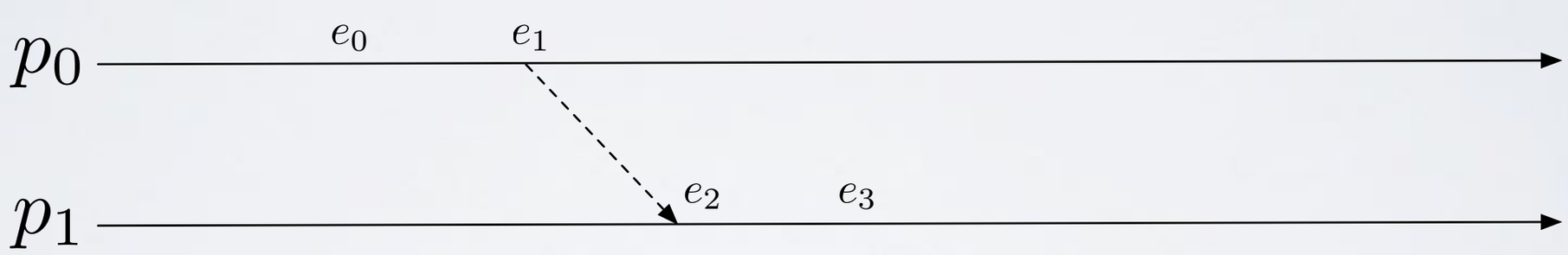
Find a way to timestamp events that follows our intuitive notion of causality.

CAUSAL RELATIONSHIPS

A. Two events occurred at some process p_i happened in the same order as p_i observes them:



B. When p_i sends a message to p_j the send event happens before the receive event:



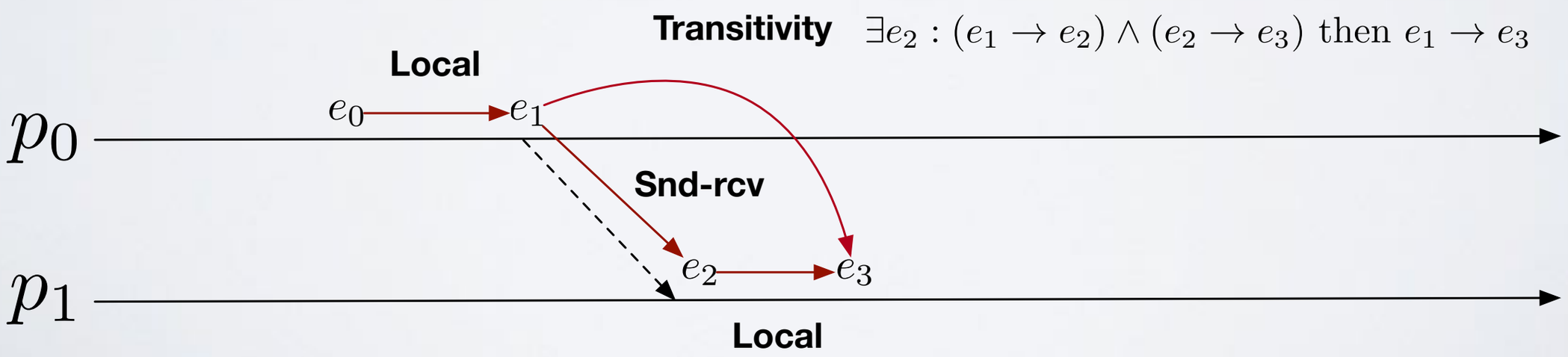
Lamport introduced the *happened-before* relation that captures the causal dependencies between events (causal order relation)

- We denote with $\rightarrow_i$ the ordering relation between events in a process p_i
- We denote with $\rightarrow$ the *happened-before* relation between any pair of events

HAPPENED-BEFORE RELATION

Two events *e* and *e'* are related by happened-before relation (*e* → *e'*) if:

- Local ordering: $\exists p_i | e \rightarrow_i e'$
- Snd-rcv ordering: $\forall m, send(m) \rightarrow receive(m)$
  - e=*send(m)* is the event of sending a message *m*
  - e'=*receive(m)* is the event of receipt of the same message *m*
- Transitivity: $\exists e'' : (e \rightarrow e'') \wedge (e'' \rightarrow e') \text{ then } e \rightarrow e'$
  - the *happened-before* relation is transitive

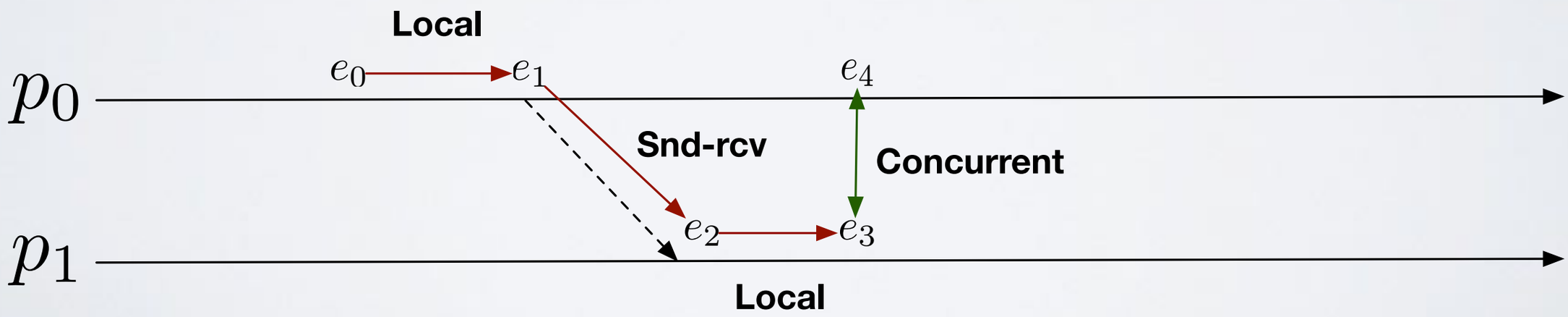


HAPPENED-BEFORE RELATION

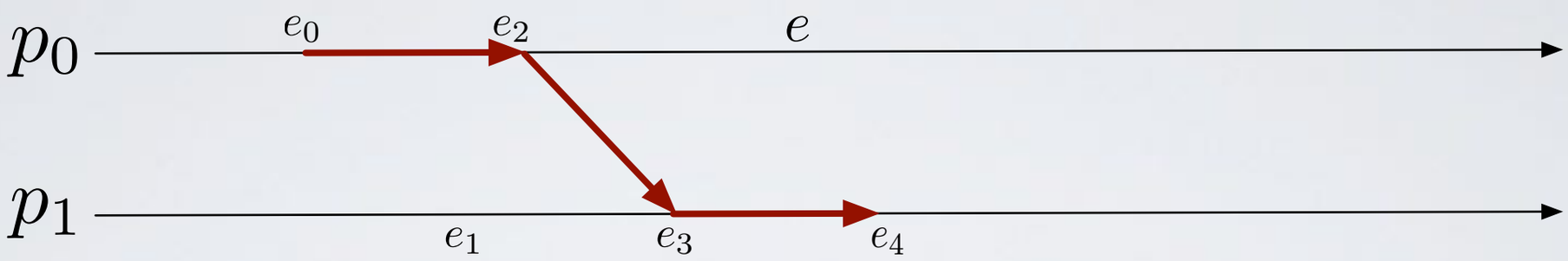
Applying these three rules is possible to define a causal ordered sequence of events $e_1, e_2, \ldots, e_n$

Notes:

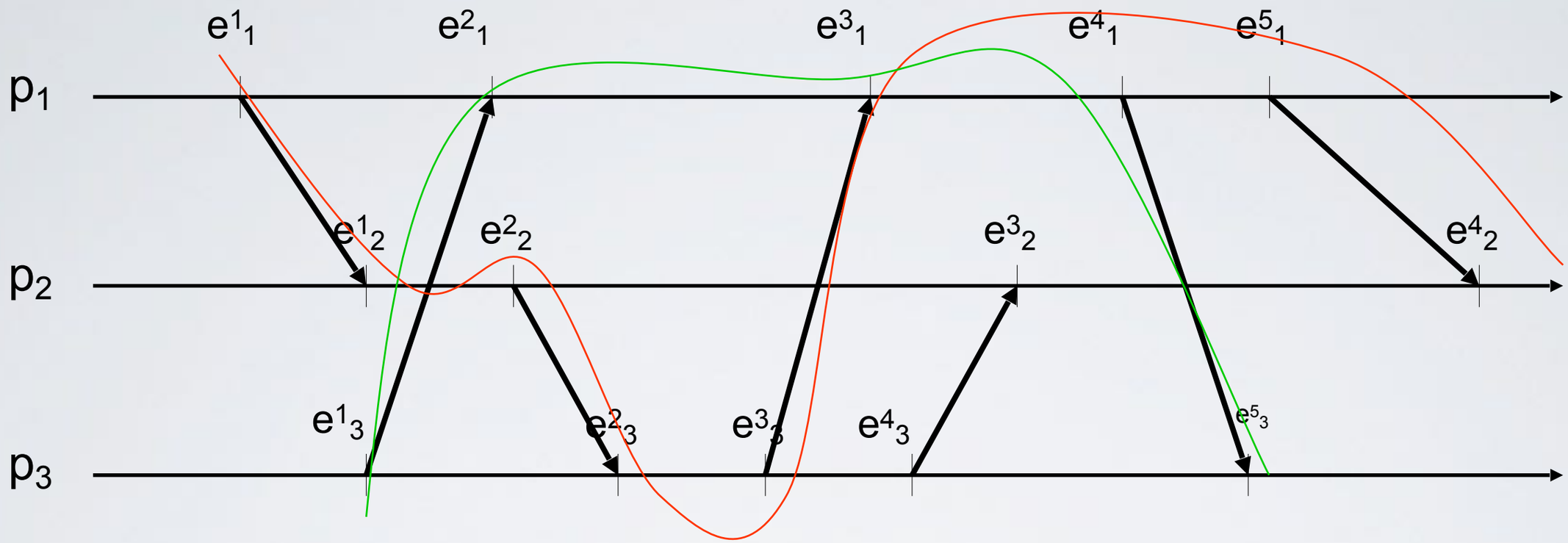
- The sequence $e_1, e_2, \ldots, e_n$ may not be unique
- It may exists a couple of events $<e_1,e_2>$ such that e_1 and e_2 are not in happened-before relation
- If e_4 and e_3 are not in happened-before relation then they are **concurrent** ($e_4||e_3$)
- For any two events e_x and e_y in the execution history of a distributed system, either $e_x \rightarrow e_y$, $e_y \rightarrow e_x$ or $e_y||e_x$



HAPPENED-BEFORE RELATION IS A POSET



HAPPENED-BEFORE: EXAMPLE



$S_1 = < e^1_1, e^1_2, e^2_2, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 >$

$S_2 = < e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 >$

Note:

e^1_3 and e^1_2 are concurrent

LOGICAL CLOCK

The Logical Clock, introduced by Lamport, is a monotonically increasing software counting register

- Logical clock is not related to physical clock

Each process p_i employs its logical clock L_i to apply a timestamp to events

$L_i(e)$ is the "logical" timestamp assigned, using the logical clock, by a process p_i to events e.

Property: If $e \rightarrow e'$ then $L(e) < L(e')$

Observation:

- The ordering relation obtained through logical timestamps is only a partial order. Consequently timestamps could not be sufficient to relate two events

SCALAR LOGICAL CLOCK

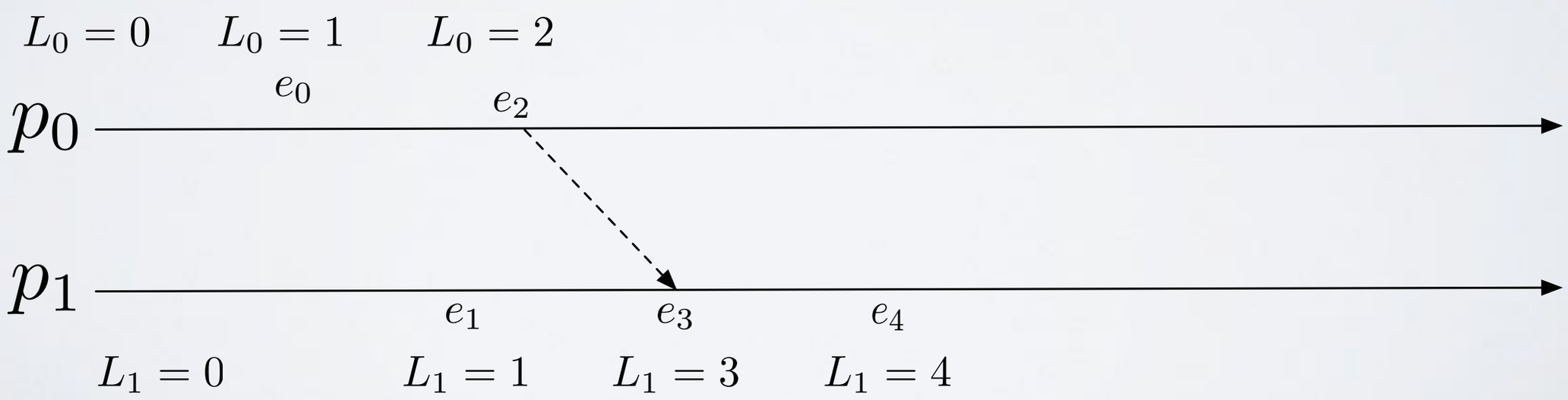
Each process p_i initializes its logical clock $L_i=0$.

When p_i sends a message m

- creates an event *send(m)*
- increases L_i
- timestamps m with $t=L_i$

When p_i receives a message m with timestamp t

- Updates its logical clock $L_i = max(t, L_i)$
- Produces an event *receive(m)*
- Increases L_i



SCALAR LOGICAL CLOCK

Rule 1: Each process p_i initializes its logical clock $L_i=0$.

When p_i sends a message m

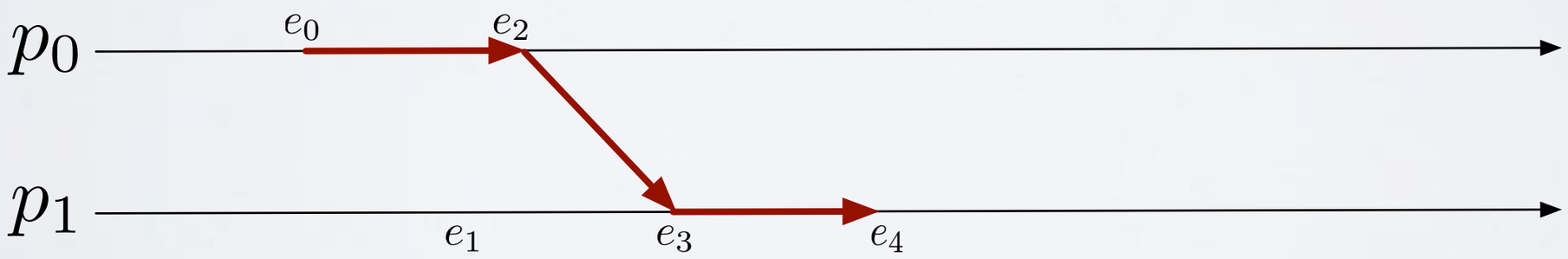
- creates an event *send(m)*
- increases L_i
- timestamps m with $t=L_i$

Rule 2: When p_i receives a message m with timestamp t

- Updates its logical clock $L_i = max(t, L_i)$
- Produces an event *receive(m)*
- Increases L_i

Why?

Property: If $e \rightarrow e'$, then $L(e) < L(e')$



SCALAR LOGICAL CLOCK

Rule 1: Each process p_i initializes its logical clock $L_i=0$.

When p_i sends a message m

- creates an event *send(m)*
- increases L_i
- timestamps m with $t=L_i$

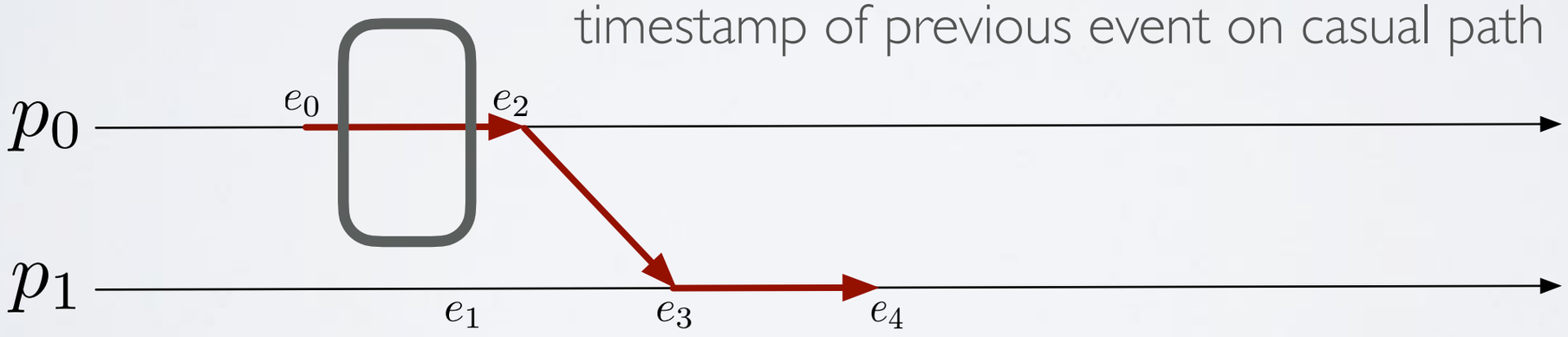
Rule 2: When p_i receives a message m with timestamp t

- Updates its logical clock $L_i = max(t, L_i)$
- Produces an event *receive(m)*
- Increases L_i

Why?

Property: If $e \rightarrow e'$, then $L(e) < L(e')$

Rule 1 = increase by one, with respect to timestamp of previous event on casual path



SCALAR LOGICAL CLOCK

Rule 1: Each process p_i initializes its logical clock $L_i=0$.

When p_i sends a message m

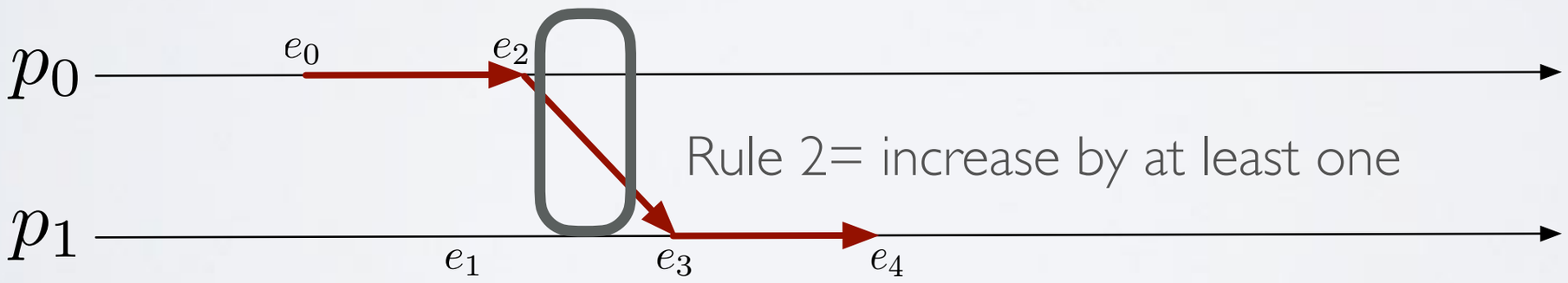
- creates an event *send(m)*
- increases L_i
- timestamps m with $t=L_i$

Rule 2: When p_i receives a message m with timestamp t

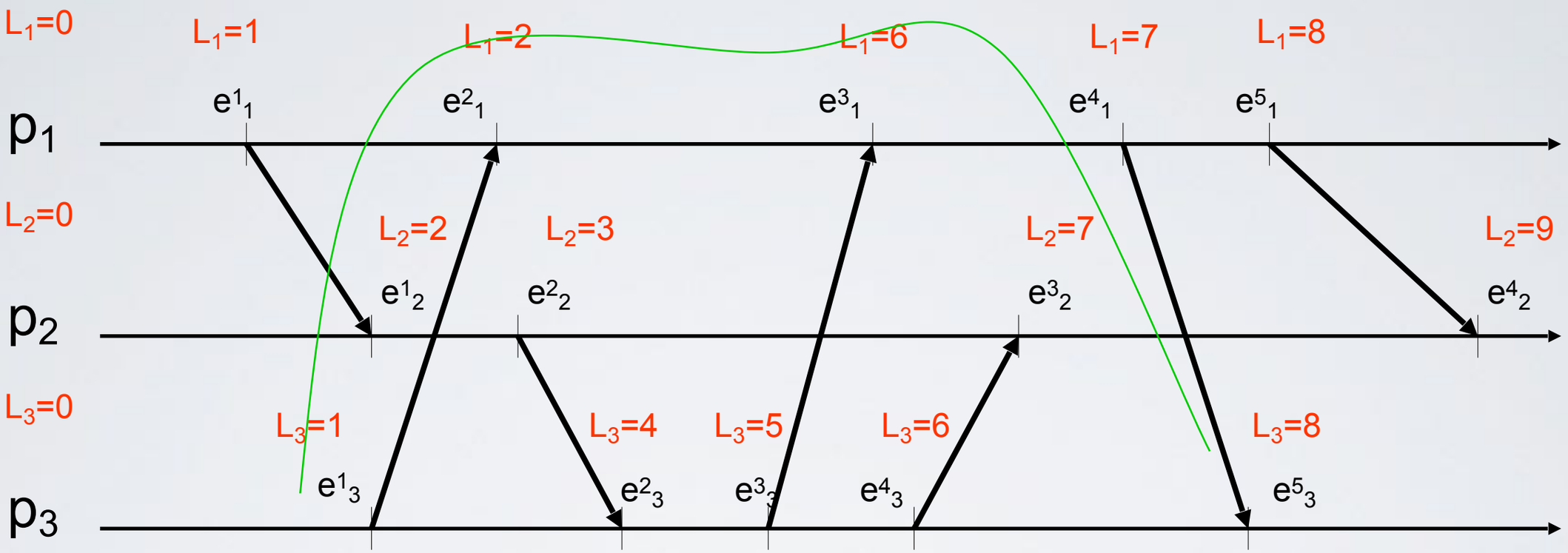
- Updates its logical clock $L_i = max(t, L_i)$
- Produces an event *receive(m)*
- Increases L_i

Why?

Property: If $e \rightarrow e'$, then $L(e) < L(e')$



SCALAR LOGICAL CLOCK: EXAMPLE



Note:

- $e^1_1 \rightarrow e^2_1$ and timestamps reflect this property
- $e^1_1 \parallel e^1_3$ and respective timestamps have the same value
- $e^1_2 \parallel e^1_3$ but respective timestamps have different values

LIMITS OF SCALAR LOGICAL CLOCK

Scalar logical clock can guarantee the following property

- IF $e \rightarrow e'$ then $L(e) < L(e')$

But it is not possible to guarantee

- IF $L(e) < L(e')$ **~~then~~** $e \rightarrow e'$

Consequently:

- It is not possible to determine, analyzing only scalar clocks, if two events are concurrent or correlated by the happened-before relation.

Mattern [1989] and Fridge [1991] proposed an improved version of logical clock where events are timestamped with local logical clock and node identifier

- Vector Clock- We want the property: IF $L(e) < L(e')$ then $e \rightarrow e'$

CAPTURING CAUSALITY- VECTOR CLOCK

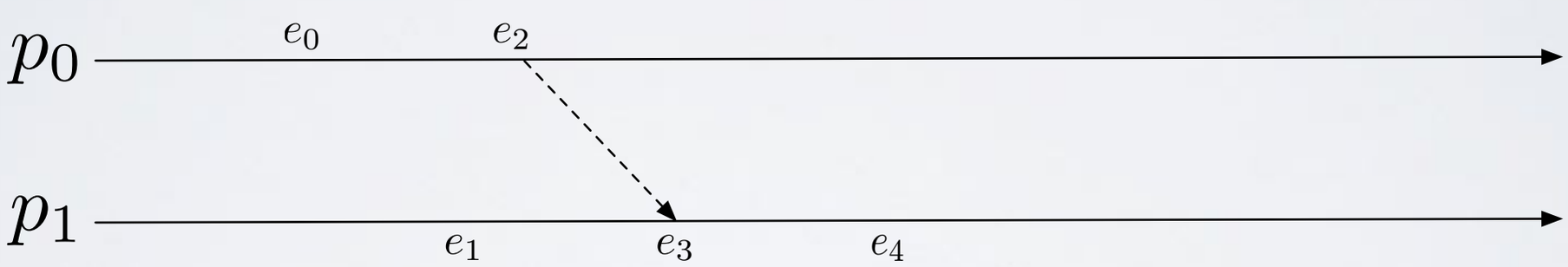
Our GOAL: IF *L(e) < L(e')* then *e → e'*

First intuition:

- L(e) has not to be a single number. What if L(e) is a history of events that happened before e (including e)?

$$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$$

$$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$$



$$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$$

$$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$$

$$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$$

CAPTURING CAUSALITY

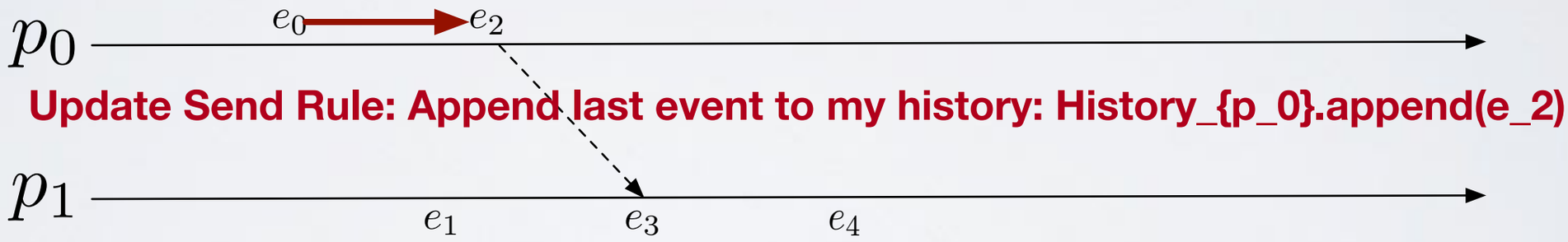
Our GOAL: IF *L(e) < L(e')* then *e → e'*

First intuition:

- L(e) has not to be a single number. What if L(e) is a history of events that happened before e (including e)?

$$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$$

$$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$$



$$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$$

$$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$$

$$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$$

CAPTURING CAUSALITY

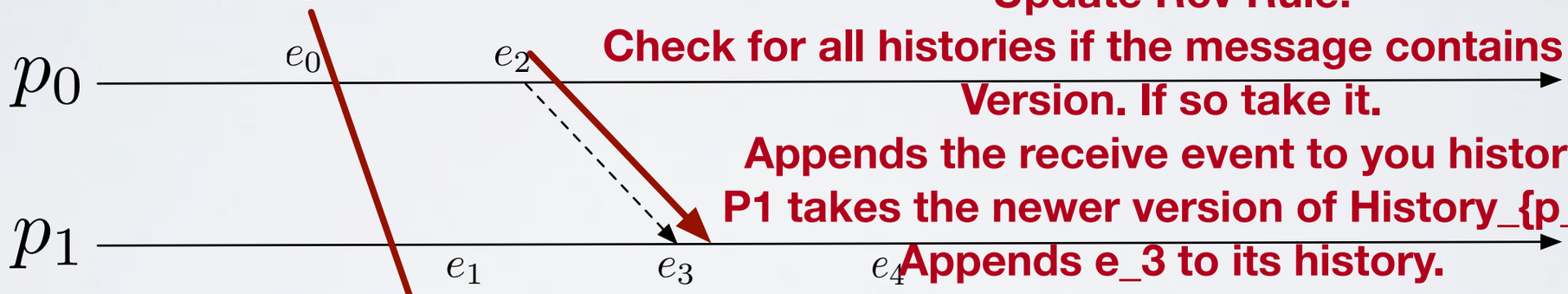
Our GOAL: IF *L(e) < L(e')* then *e → e'*

First intuition:

- L(e) has not to be a single number. What if L(e) is a history of events that happened before e (including e)?

$$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$$

$$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$$



Update Rcv Rule:
Check for all histories if the message contains a newer Version. If so take it.
Appends the receive event to you history.
P1 takes the newer version of History_{p_0}
Appends e_3 to its history.

Take History_p_0 from L(e_2)

$$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$$

Appends e_3

$$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$$

$$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$$

CAPTURING CAUSALITY

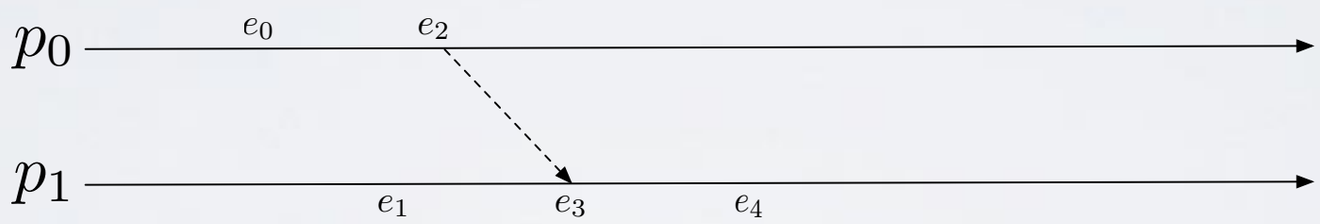
Our GOAL: IF *L(e) < L(e')* then *e → e'*

First intuition:

- L(e) has not to be a single number. What if L(e) is a history of events that happened before e (including e)?

$$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$$

$$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$$



$$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$$

$$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$$

$$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$$

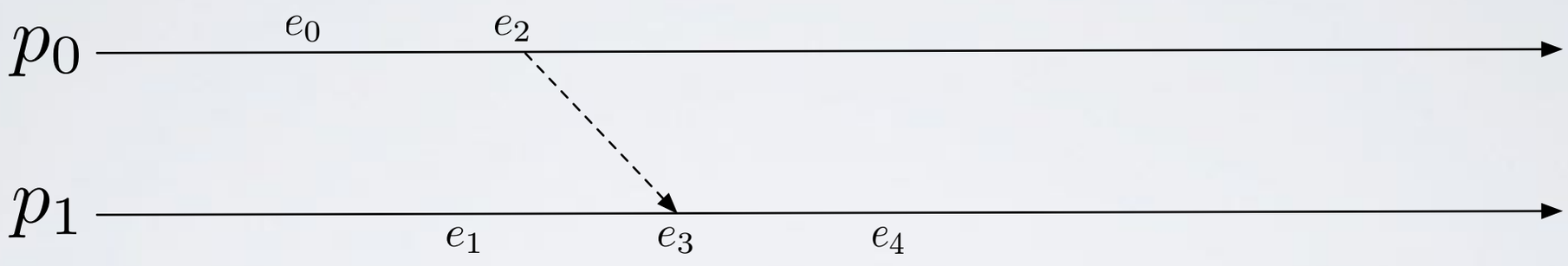
$$L(e_i) > L(e_j) \text{ if and only if } \forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$$

CAPTURING CAUSALITY

$L(e_i) > L(e_j)$ if and only if $\forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$

$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$

$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$



$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$

$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$

$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$

$$\textbf{if} e \rightarrow e' \textbf{then } L(e) < L(e')$$

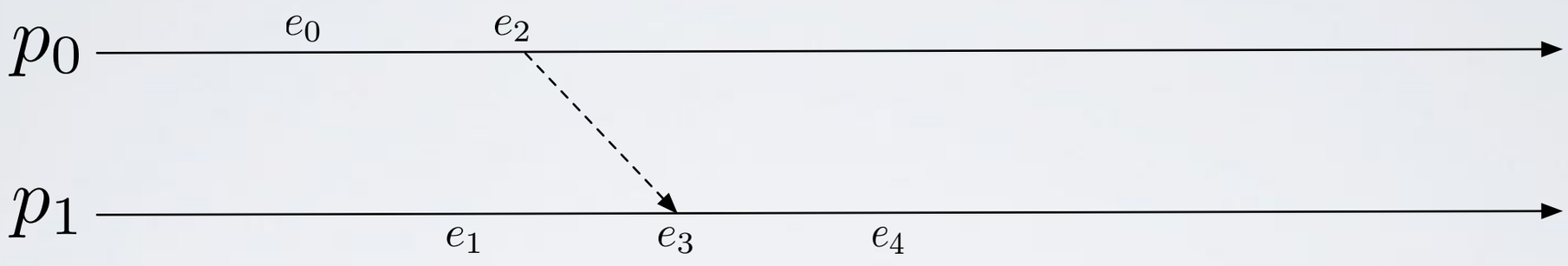
Intuition: the timestamp L(e') ``contains" L(e)

CAPTURING CAUSALITY

$L(e_i) > L(e_j)$ if and only if $\forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$

$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$

$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$



$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$

$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$

$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$

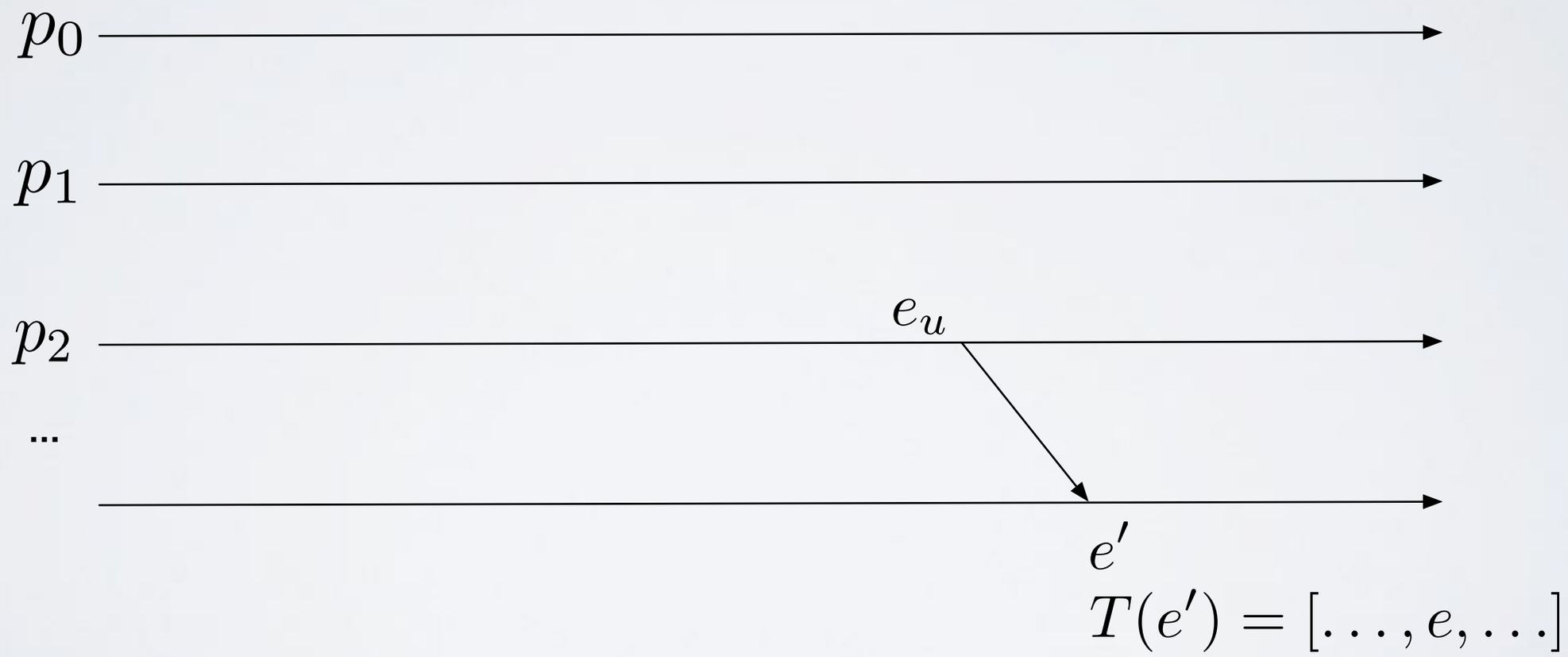
$$\textbf{if } L(e) < L(e') \textbf{ then } e \rightarrow e'$$

Intuition: L(e') contains e, thus exists a causal path from e to e'

CAPTURING CAUSALITY

$$\textbf{if}\, L(e) < L(e')\textbf{then}\ e \rightarrow e'$$

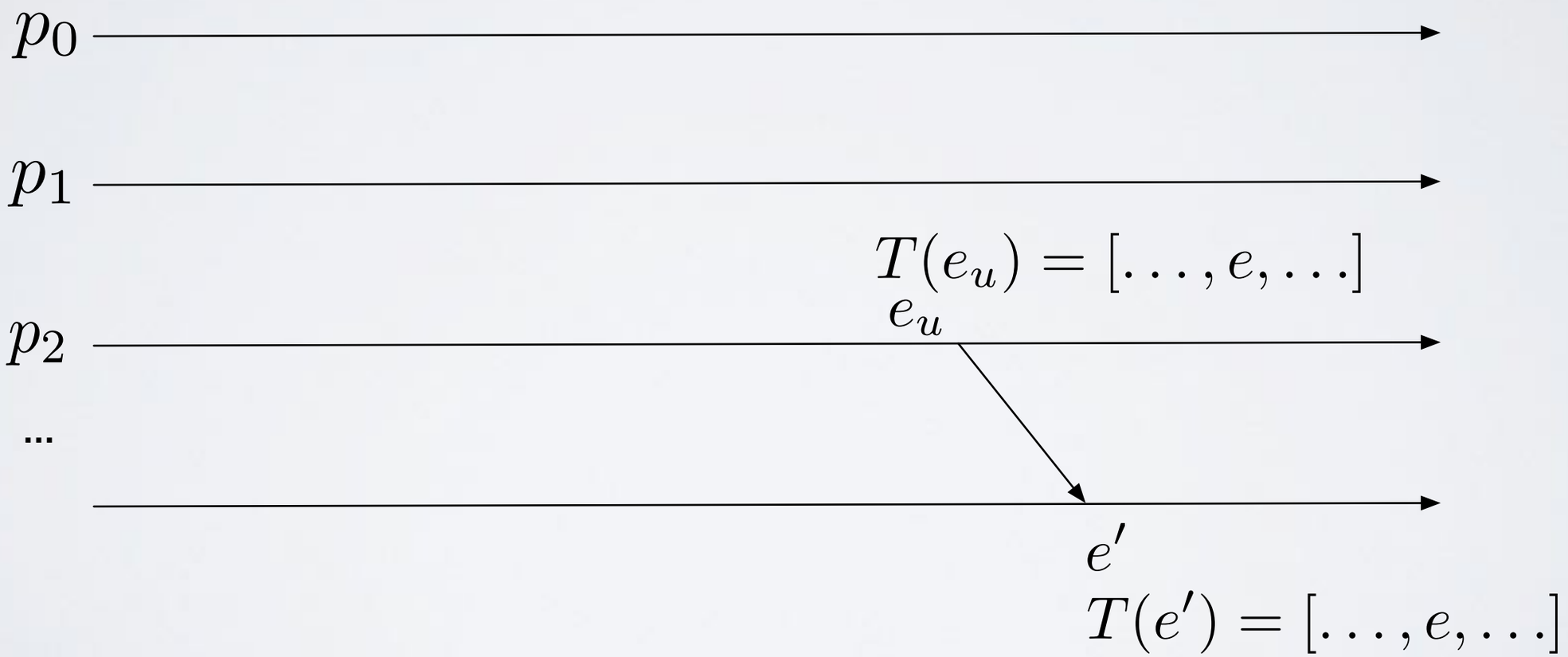
Intuition: L(e') contains e, thus it exists a causal path from e to e'.



CAPTURING CAUSALITY

$$\textbf{if}\, L(e) < L(e')\textbf{then}\ e \rightarrow e'$$

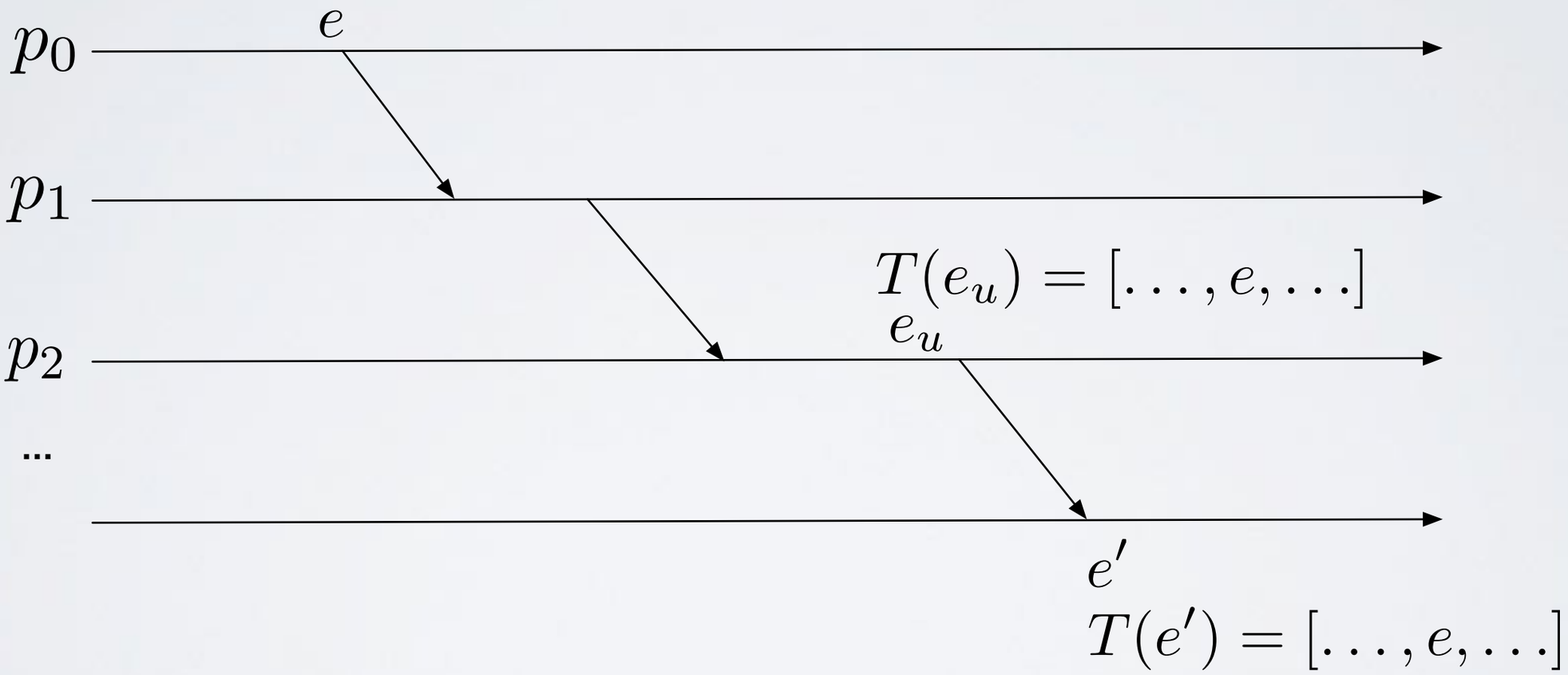
Intuition: L(e') contains e, thus it exists a causal path from e to e'.



CAPTURING CAUSALITY

$$\textbf{if}\, L(e) < L(e')\textbf{then}\ e \rightarrow e'$$

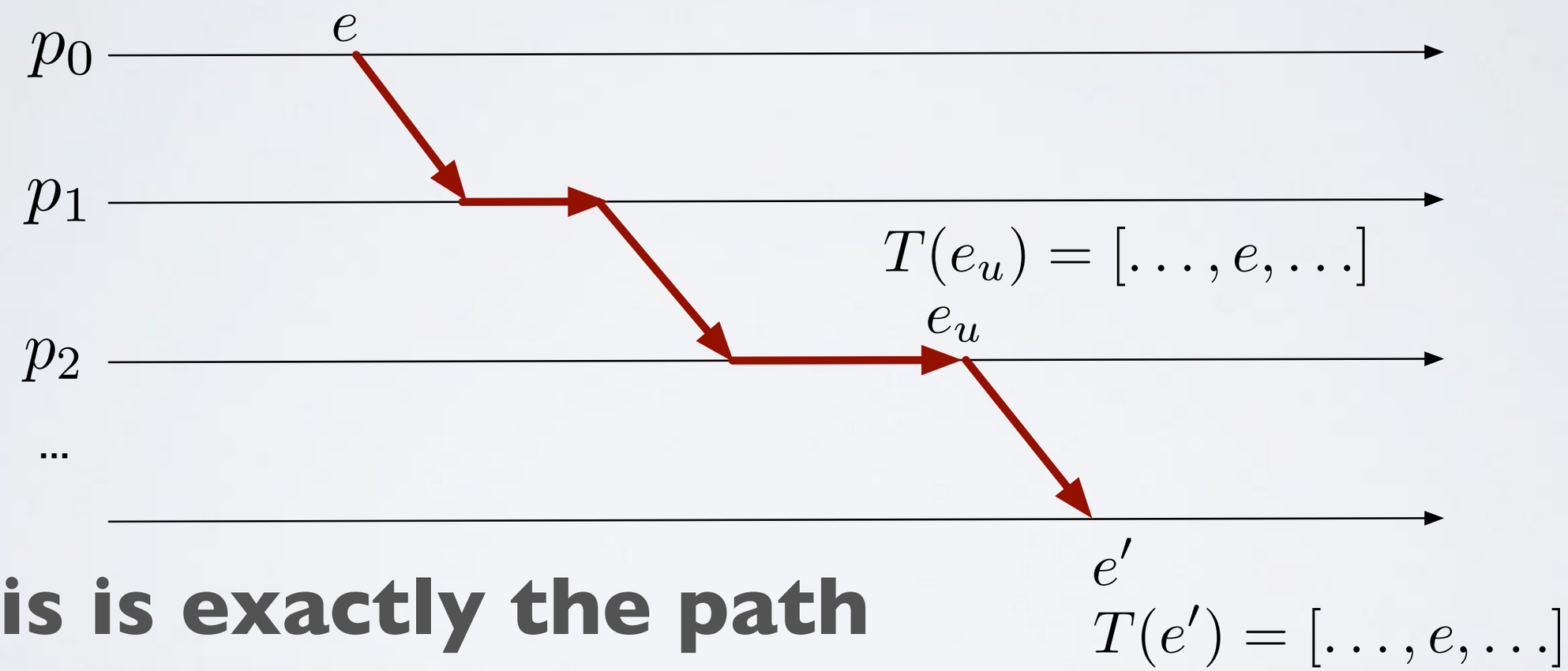
Intuition: L(e') contains e, thus it exists a causal path from e to e'.



CAPTURING CAUSALITY

$$\textbf{if}\, L(e) < L(e')\textbf{then}\ e \rightarrow e'$$

Intuition: L(e') contains e, thus it exists a causal path from e to e'.



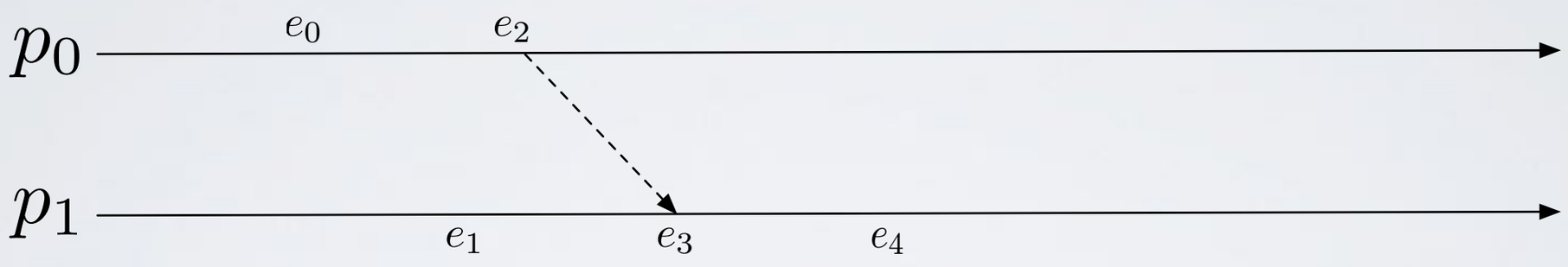
This is exactly the path Creating the causal relation

ENGINEERING OUR SOLUTION

$L(e_i) > L(e_j)$ if and only if $\forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$

$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$

$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$



$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$

$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$

$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$

$$\exists e : L(e) = [\ldots, History_{p_x} = [e_0, e_1, e_2], \ldots]$$

$$\exists e' : L(e') = [\ldots, History_{p_x} = [e_0, e_1, e_3], \ldots]$$

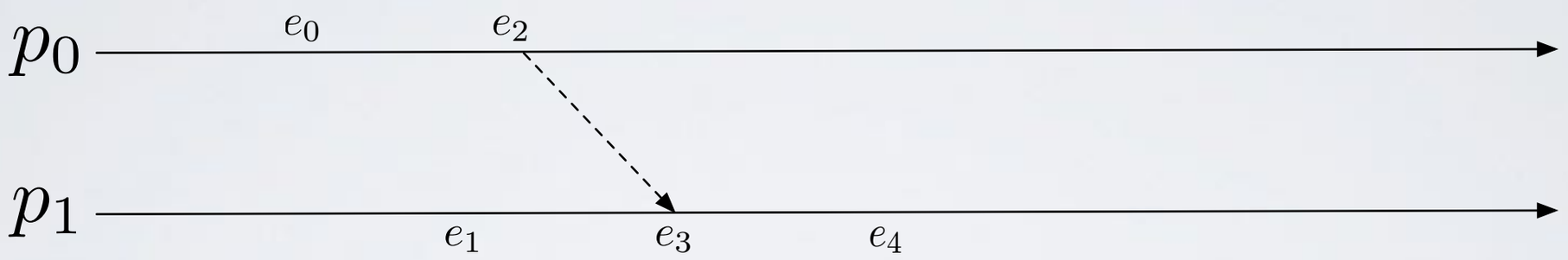
Possible?

ENGINEERING OUR SOLUTION

$L(e_i) > L(e_j)$ if and only if $\forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$

$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$

$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$



$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$

$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$

$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$

$\exists e : L(e) = [\ldots, History_{p_x} = [e_0, e_1, e_2], \ldots]$

$\exists e' : L(e') = [\ldots, History_{p_x} = [e_0, e_1, e_3, e_4], \ldots]$

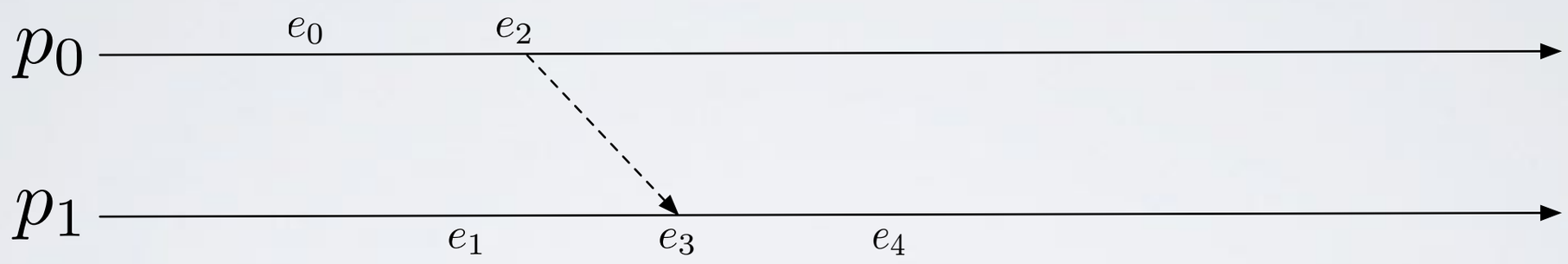
Possible?

ENGINEERING OUR SOLUTION

$L(e_i) > L(e_j)$ if and only if $\forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$

$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$

$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$



$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$

$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$

$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$

Then we have

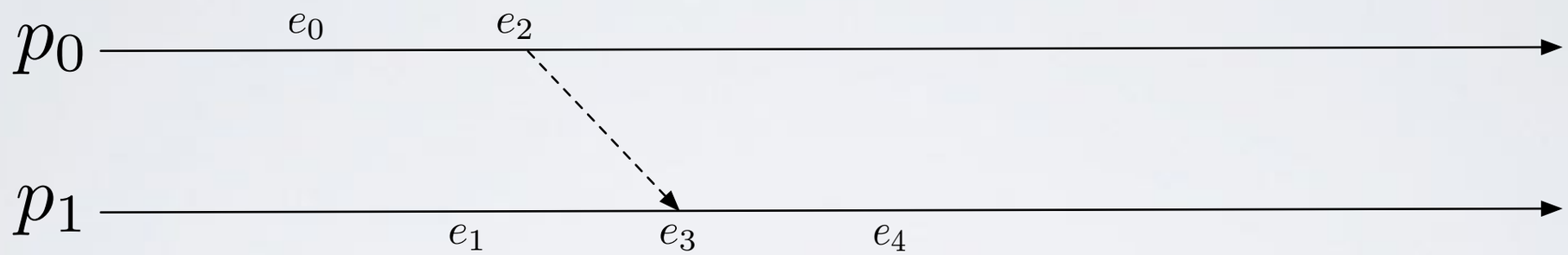
$History_x \subset History'_x \rightarrow History_x$ is a proper prefix of $History'_x$

ENGINEERING OUR SOLUTION

$L(e_i) > L(e_j)$ if and only if $\forall k : L(e_j)_{History_k} \subseteq L(e_i)_{History_k} \wedge \exists x : L(e_j)_{History_x} \subset L(e_i)_{History_x}$

$L(e_0) = [\text{History}_{p_0} : [e_0], \text{History}_{p_1} : []]$

$L(e_2) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : []]$



$L(e_1) = [\text{History}_{p_0} : [], \text{History}_{p_1} : [e_1]]$

$L(e_3) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3]]$

$L(e_4) = [\text{History}_{p_0} : [e_0, e_2], \text{History}_{p_1} : [e_1, e_3, e_4]]$

Then we have

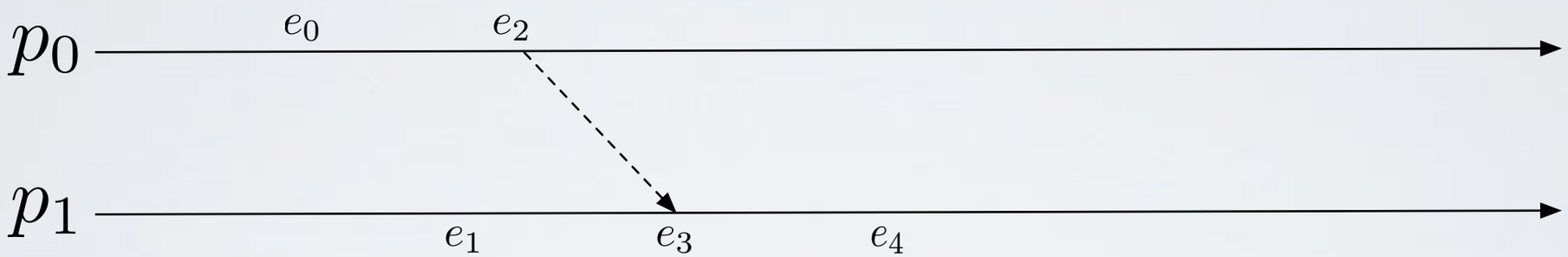
$$History_x \subset History'_x \rightarrow len(History_x) < len(History'_x)$$

ENGINEERING OUR SOLUTION

$$L(e_i) > L(e_j) \text{ if and only if } \forall k \in [0, n-1] : L(e_j)_k \leq L(e_i)_k \exists x : L(e_j)_x < L(e_i)_x$$

$$L(e_0) = [\text{History}_{p_0} : len([e_0]), \text{History}_{p_1} : len([])] = [1, 0]$$

$$L(e_2) = [\text{History}_{p_0} : len([e_0, e_2]), \text{History}_{p_1} : len([])] = [2, 0]$$



$$L(e_1) = [\text{History}_{p_0} : len([]), \text{History}_{p_1} : len([e_1])] = [0, 1]$$

$$L(e_3) = [\text{History}_{p_0} : len([e_0, e_2]), \text{History}_{p_1} : len([e_1, e_3])] = [2, 2]$$

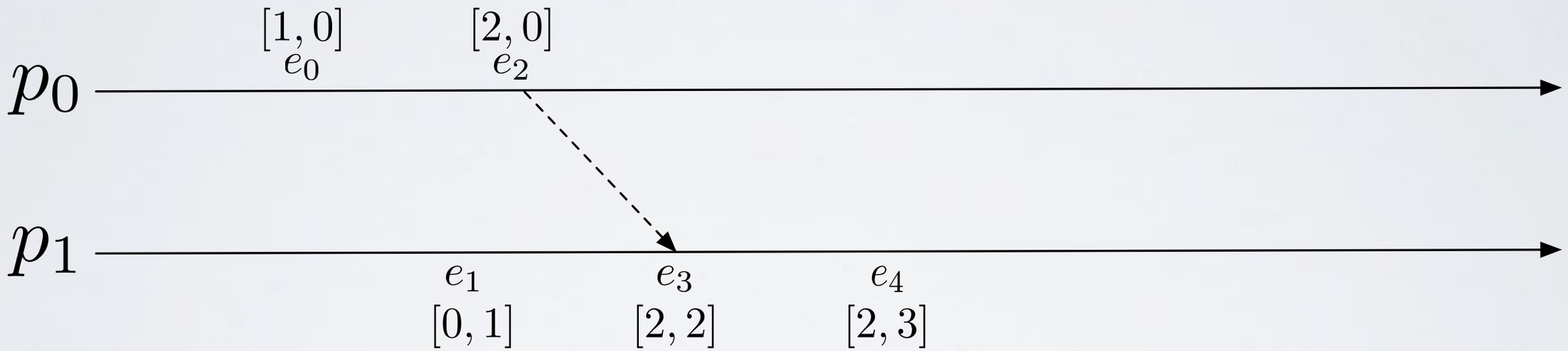
$$L(e_4) = [\text{History}_{p_0} : len([e_0, e_2]), \text{History}_{p_1} : len([e_1, e_3, e_4])] = [2, 3]$$

Then we have

$$History_x \subset History'_x \rightarrow len(History_x) < len(History'_x)$$

VECTOR CLOCKS

$L(e_i) > L(e_j)$ if and only if $\forall k \in [0, n-1] : L(e_j)_k \leq L(e_i)_k \exists x : L(e_j)_x < L(e_i)_x$



VECTOR CLOCK : DEFINITION

A vector clock for a set of *N* processes is an array of *N* integer counters

- Each process p_i maintains a vector clock V_i and timestamps events by mean of it
- Similarly to scalar clock, a vector clock is attached to message *m* (in this case we attach an array of integer)

Vector clocks allow processes to order events in *happened-before* order on the basis of timestamps

- *Scalar clocks*: $e \rightarrow e' \Rightarrow L(e) < L(e')$
- *Vector clocks*: $e \rightarrow e' \Leftrightarrow V(e) < V(e')$

VECTOR CLOCK : AN IMPLEMENTATION

Each process p_i initializes its clock V_i

- $V_i[j] = 0 \quad \forall j = \{1, \ldots, N\}$

p_i increases $V_i[i]$ by 1 when it generates a new event

- $V_i[i] = V_i[i] + 1$

When p_i sends a message m

- Creates an event *send(m)*
- Increases V_i
- timestamps m with $t=V_i$

VECTOR CLOCK : AN IMPLEMENTATION

Each process p_i initializes its clock V_i

- $V_i[j] = 0 \quad \forall j \in \{1,\ldots,N\}$

p_i increases $V_i[i]$ by 1 when it generates a new event

- $V_i[i] = V_i[i] + 1$

When p_i receives a message m containing timestamp V_t

- Updates its logical clock $V_i[j] = max(V_t[j], V_i[j]) \quad \forall j \in \{1,\ldots,N\}$
- Generates an event *receive(m)*
- Increases V_i

VECTOR CLOCK: PROPERTIES

A vector clock V_i

- $V_i[i]$ represents the number of events produced by p_i
- $V_i[j]$ with $i \neq j$ represents the number of events generated by pj that p_i knowns

$$V = V' \Leftrightarrow V[j] = V'[j] \quad \forall j \in \{1,\ldots,N\}$$

$$V \leq V' \Leftrightarrow V[j] \leq V'[j] \quad \forall j \in \{1,\ldots,N\}$$

$$V < V' \Leftrightarrow V \leq V' \ \wedge \ \exists j \in \{1,\ldots,N\} \mid V[j] < V'[j]$$

VECTOR CLOCK: AN EXAMPLE

1
2
0

V(e)

1
2
2

V(e')

$\Rightarrow \quad e \mapsto e'$

1
2
0

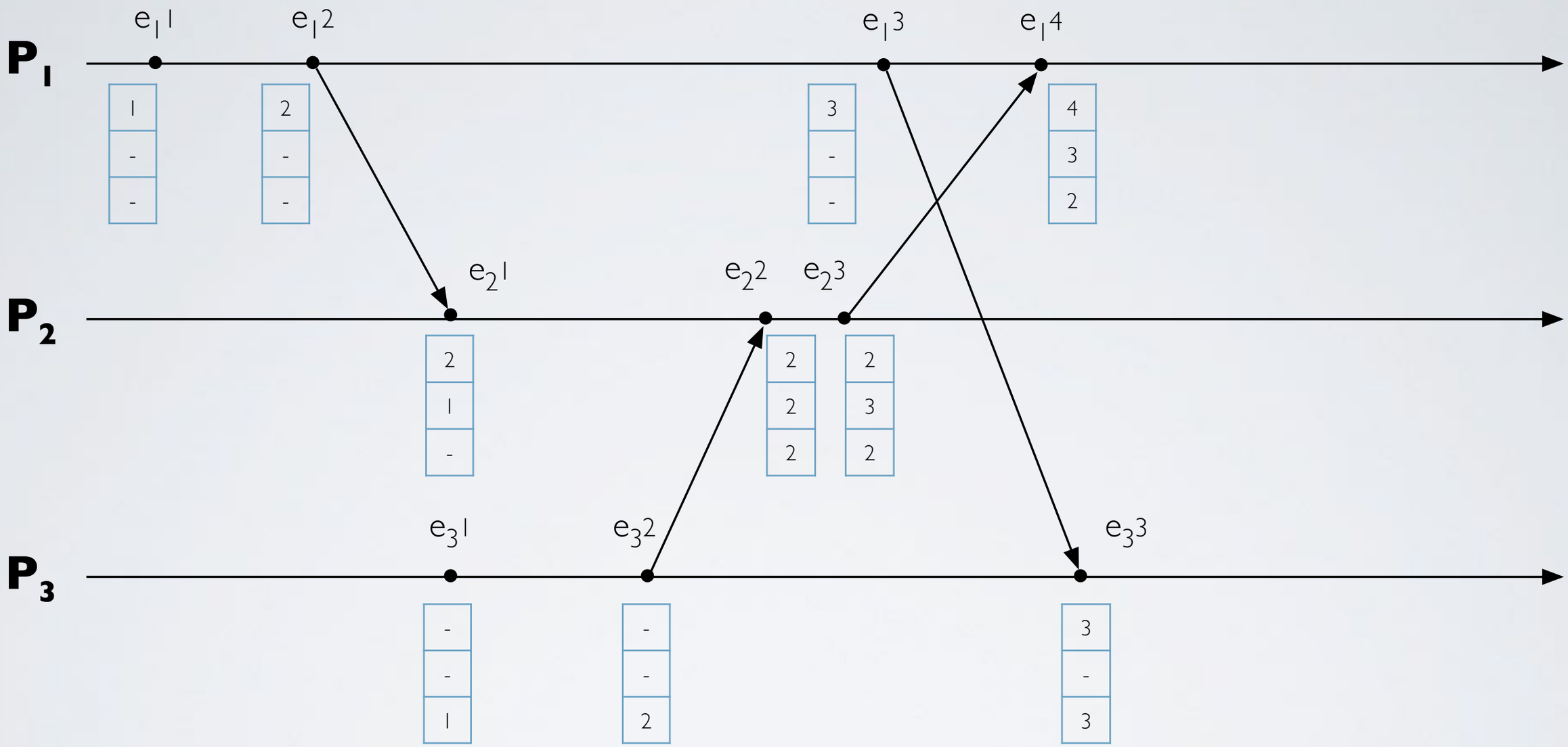
V(e)

1
0
2

V(e')

$\Rightarrow \quad e \parallel e'$

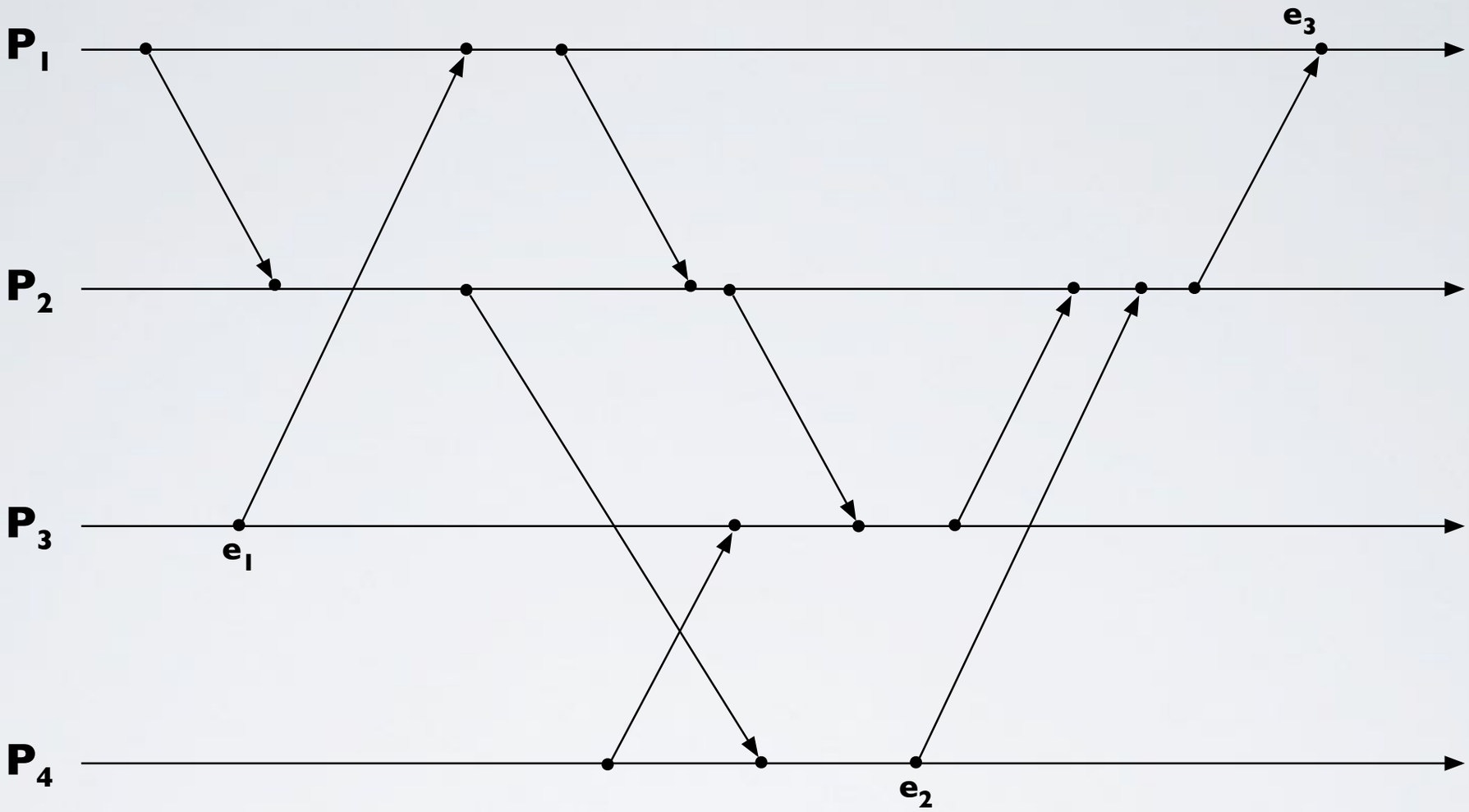
VECTOR CLOCK: AN EXAMPLE



$[-,-,1] < [4,3,2] \Rightarrow e_3^1 \mapsto e_1^4$

$[4,3,2] \; ? \; [3,-,3] \Rightarrow e_1^4 \; || \; e_3^3$

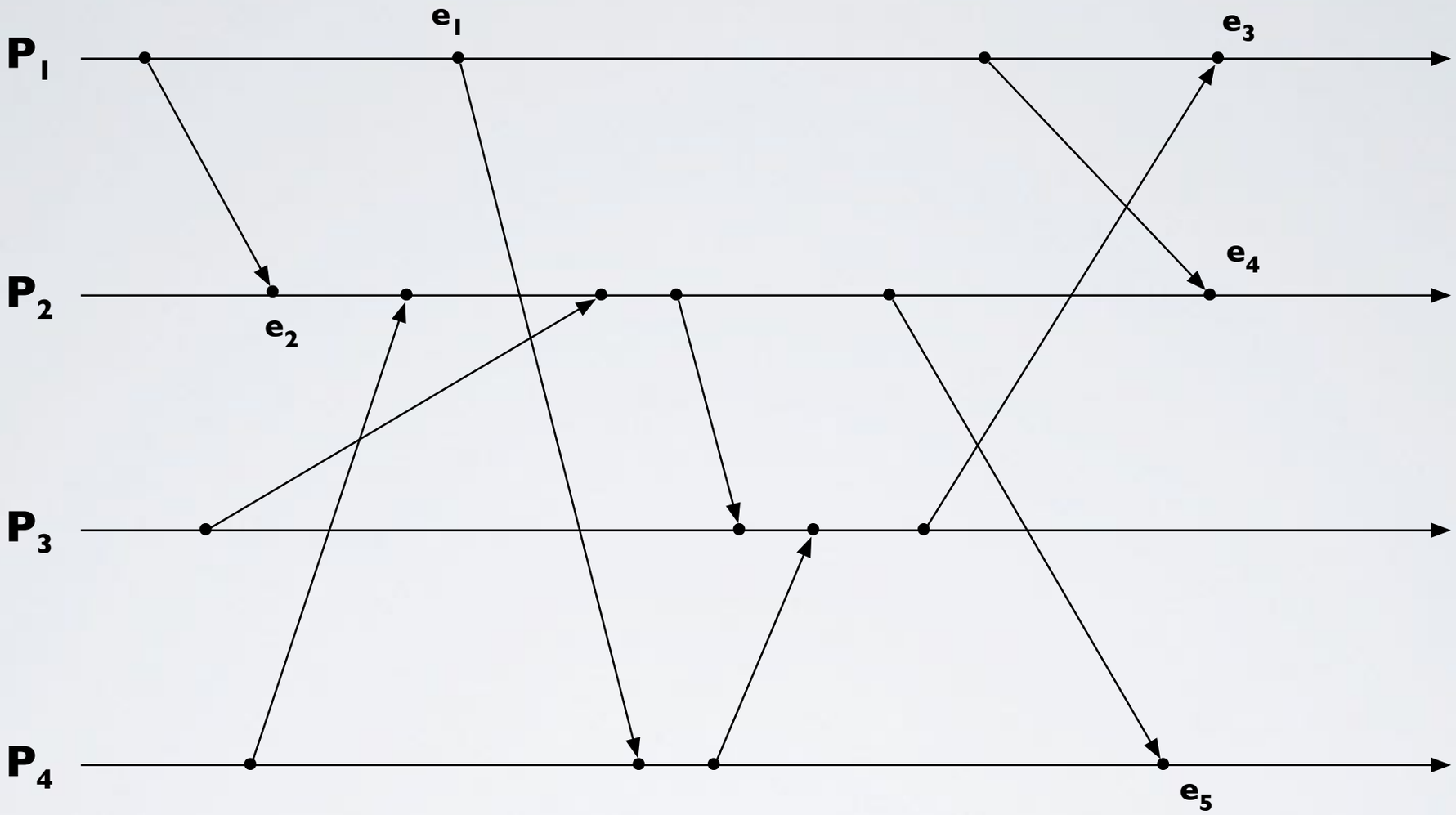
VECTOR CLOCK: EXERCISE



Apply a vector clock to each event and then identify false statements among the following:

- e1 → e2
- *e1* → e3
- e2 → e3

VECTOR CLOCK: EXERCISE



Apply a vector clock to each event and then identify false statements among the following:

- e1 → e2
- *e1* → e3
- e2 → e3

LOGICAL CLOCK IN DISTRIBUTED ALGORITHMS

We have seen two mechanisms to represent logical time

- Logical Clocks (also called Scalar Clocks and Lamport's Clock).
- Vector Clocks

Each mechanism can be used to solve different problems, depending on the problem specification

- Scalar Timestamp → Lamport's Mutual Exclusion
- Vector Timestamp → Causal Broadcast

BREAK TIME

- Consider an asynchronous message passing system that uses vectors clock to implement some causal consistency check. The message passing system is composed by 4 processes with IDs 1 to 4, and, as usual, the ID is used as displacement in the vector clock (i.e., the locations are (p1, p2, p3, p4)). Vector clocks are updated increasing before send. Processes communicate by point2point links. You start debugging process p1 (the process with id 1) in the middle of the algorithm execution. You see the following stream of messages exiting and entering the ethernet card of process p1:·
- Time 00:00 – EXITING: Send Message [MSG CONTENT] Vector Clock: (1, 0, 0, 0)
- Time 00:05 – ENTERING: Rcvd Message [MSG CONTENT] Vector Clock: (1, 2, 3, 1)
- Q3.1: Draw an execution that justifies the vectors clocks you are seeing. Is such an execution unique?
- Q3.2: There exists an execution that justifies the vector clocks and where there exists at least a process that does not send any message? Justify your answer.

BREAK TIME

- Time 00:00 – EXITING: Send Message [MSG CONTENT] Vector Clock: (1, 0, 0, 0)
- Time 00:05 – ENTERING: Rcvd Message [MSG CONTENT] Vector Clock: (1, 2, 3, 1)
- Q3.1: Draw an execution that justifies the vectors clocks you are seeing. Is such an execution unique?

BREAK TIME

- Time 00:00 – EXITING: Send Message [MSG CONTENT] Vector Clock: (1, 0, 0, 0)
- Time 00:05 – ENTERING: Rcvd Message [MSG CONTENT] Vector Clock: (1, 2, 3, 1)
- Q3.2: There exists an execution that justifies the vector clocks and where there exists at least a process that does not send any message? Justify your answer.

LAMPORT'S MUTUAL EXCLUSION - NO CRASH TOLERANT

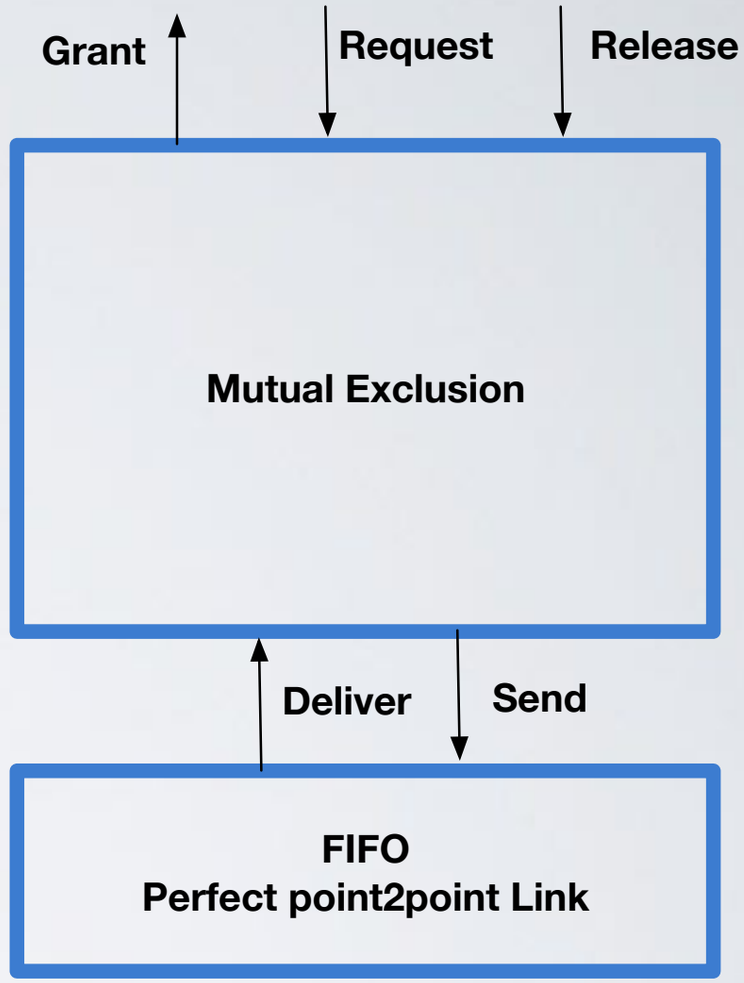
MUTUAL EXCLUSION ABSTRACTION

Events:

- Request: From upper layer - Requests access to Critical Section (CS)
- Grant: To upper layer - Grant the access to CS.
- Release: From upper layer - Release the CS.

Properties:

- (Mutual Exclusions) At any time t, only one process is inside the CS.
- (Liveness) If a process p requests access, then it eventually enters the CS.
- (Fairness) If the request of process p happens before the request of process q, then q cannot access the CS before p.



LAMPORT'S ALGORITHM

THE ALGORITHM ASSUMES NO CRASHES F=0!

When a process wants to enter the Critical Section (CS) it sends a request message to all the other

An history of the operations is maintained using Scalar Clocks

FIFO LINKS ARE ASSUMED!

Each transmission and reception event is relevant to the computation:

- The clock is incremented for each send and receive event

Algorithm 1 Lamport's ME Algorithm on process p_i - MSGS are REQ, ACK, RLS

1: **upon event** INIT
2: $Requests = Acks = \emptyset$
3: $scalar\_clock = 0$
4: $my\_req = \perp$
5: $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$ ▷ Set of all processes

6: ▷ Request access to CS from upper layer
7: **upon event** REQUEST
8: $scalar\_clock = scalar\_clock + 1$
9: $my\_req = (REQ, ts =< i, scalar\_clock >)$
10: **for all** $p_j \in \Pi$ **do**
11: SEND FIFOPERFECTLINK(p_j, req_msg) ▷ Send a REQ containing my ID (i) and ts (scalar_clock) to all $p \in \Pi$

12: ▷ Release CS from upper layer
13: **upon event** RELEASE
14: $my\_req = \perp$
15: $scalar\_clock = scalar\_clock + 1$
16: **for all** $p_j \in \Pi$ **do**
17: SEND FIFOPERFECTLINK$(p_j, (RLS, ts =< i, scalar_clock >))$

18: ▷ $ts(x) < ts(y)$ when $scalar\_clock$ of x is less than the one of y, or they are equal and the id that sent x is less than the id that sent y
19: **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in \Pi : \exists m \in Acks | ts(m) > ts(my\_req) \wedge sender(m) = p$
20: **trigger event** GRANTED

21: **upon event** DELIVER MESSAGE(m)
22: $scalar\_clock = max(clock(m), scalar\_clock) + 1$
23: **if** m is a REQ **then**
24: $Request\_set = Request\_set \cup \{m\}$
25: $scalar\_clock = scalar\_clock + 1$
26: SEND FIFOPERFECTLINK$(sender(m), (ACK, ts =< i, scalar_clock >))$
27: **else if** m is a ACK **then**
28: $Acks = Acks \cup \{m\}$
29: **else if** m is a RLS $\wedge \exists req \in Request\_set$: sender(req)=sender(m) **then**
30: $Requests = Requests \setminus \{req\}$

LAMPORT'S ALGORITHM

Local data structures to each process p_i

- *ck:* Is the counter for process p_i
- *Requests*: a set maintained by p_i where CS access requests are stored

Algorithm rules for a process p_i:

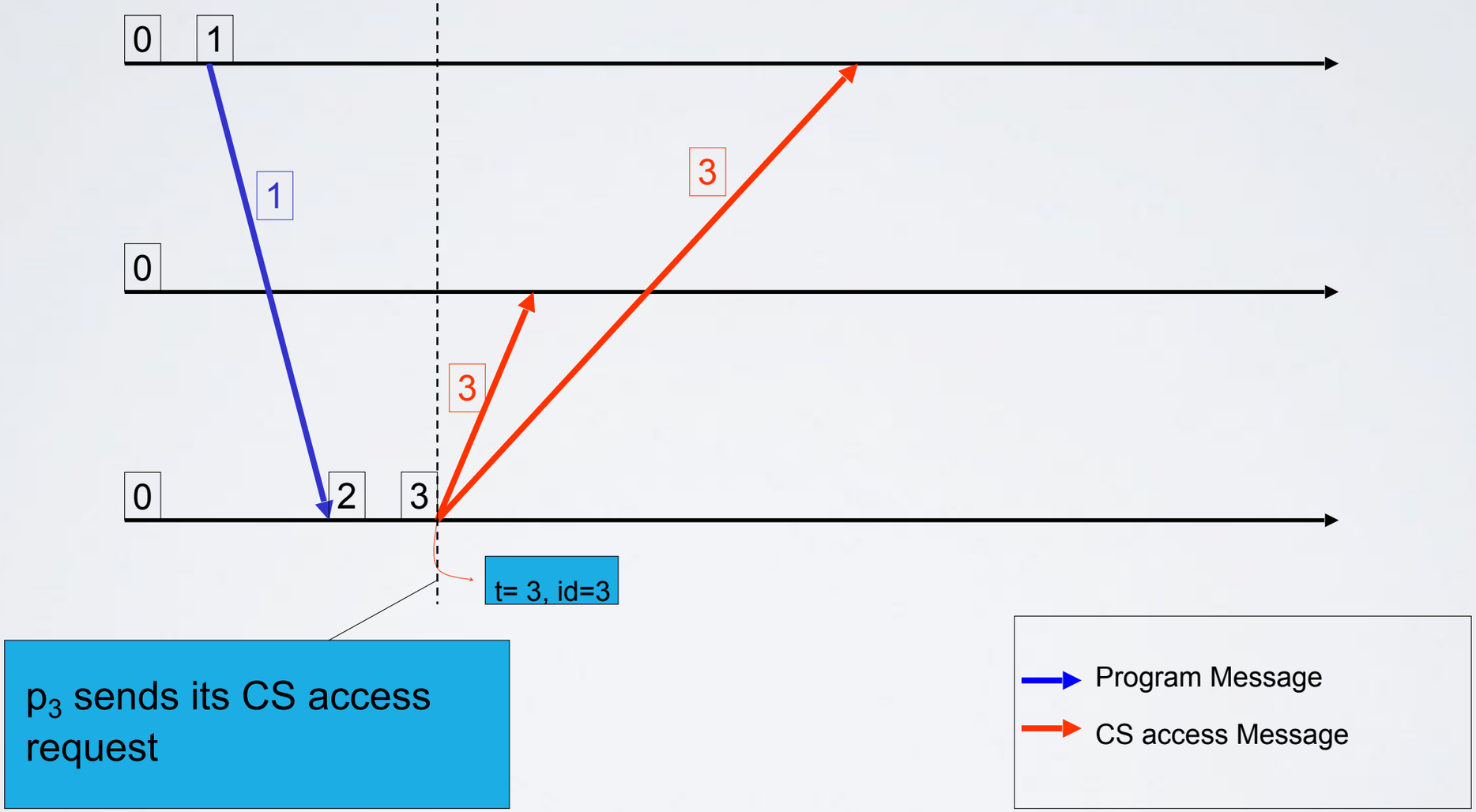
- Access the CS
  - p_i sends a request message, attaching *ck*, to all the other processes
  - p_i adds its request to *Requests*
- Request reception from a process p_j
  - p_i puts p_j request (including the timestamp) in its *Requests*
  - p_i sends back an *ACK* message to p_j including its local timestamp ck

LAMPORT'S ALGORITHM

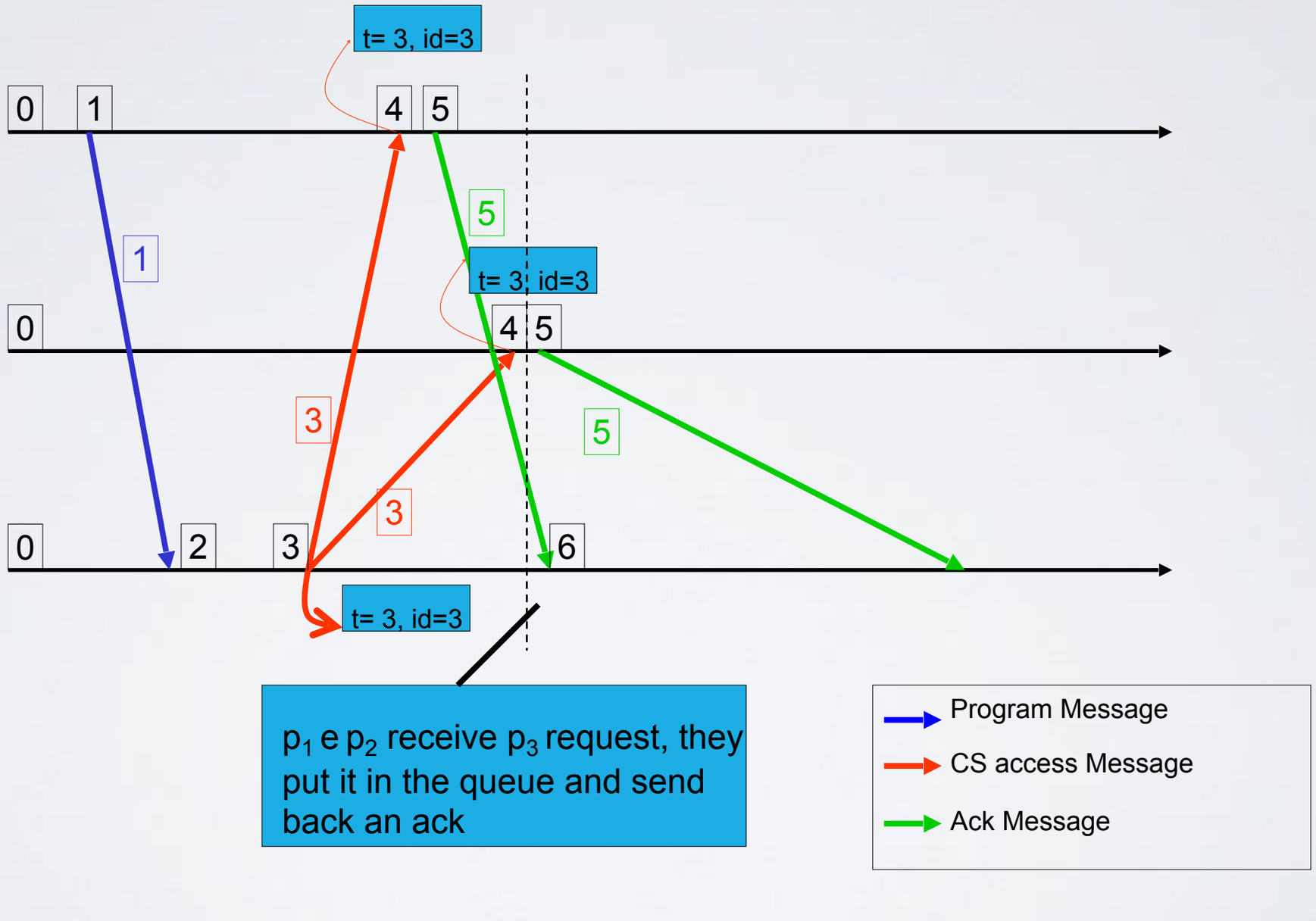
Algorithm rules for a process p_i

- p_i enters the *CS* iff
  - The request of p_i is the one with smallest ts in its Requests
  - p_i has already received an *ACK* with timestamp *t'* from any other process and *t'>t*
- Release of the CS
  - p_i sends a *RELEASE* message to all the other processes
  - p_i deletes its request from Requests
- Reception of a release message from a process p_j
  - p_i deletes p_j's request from Requests

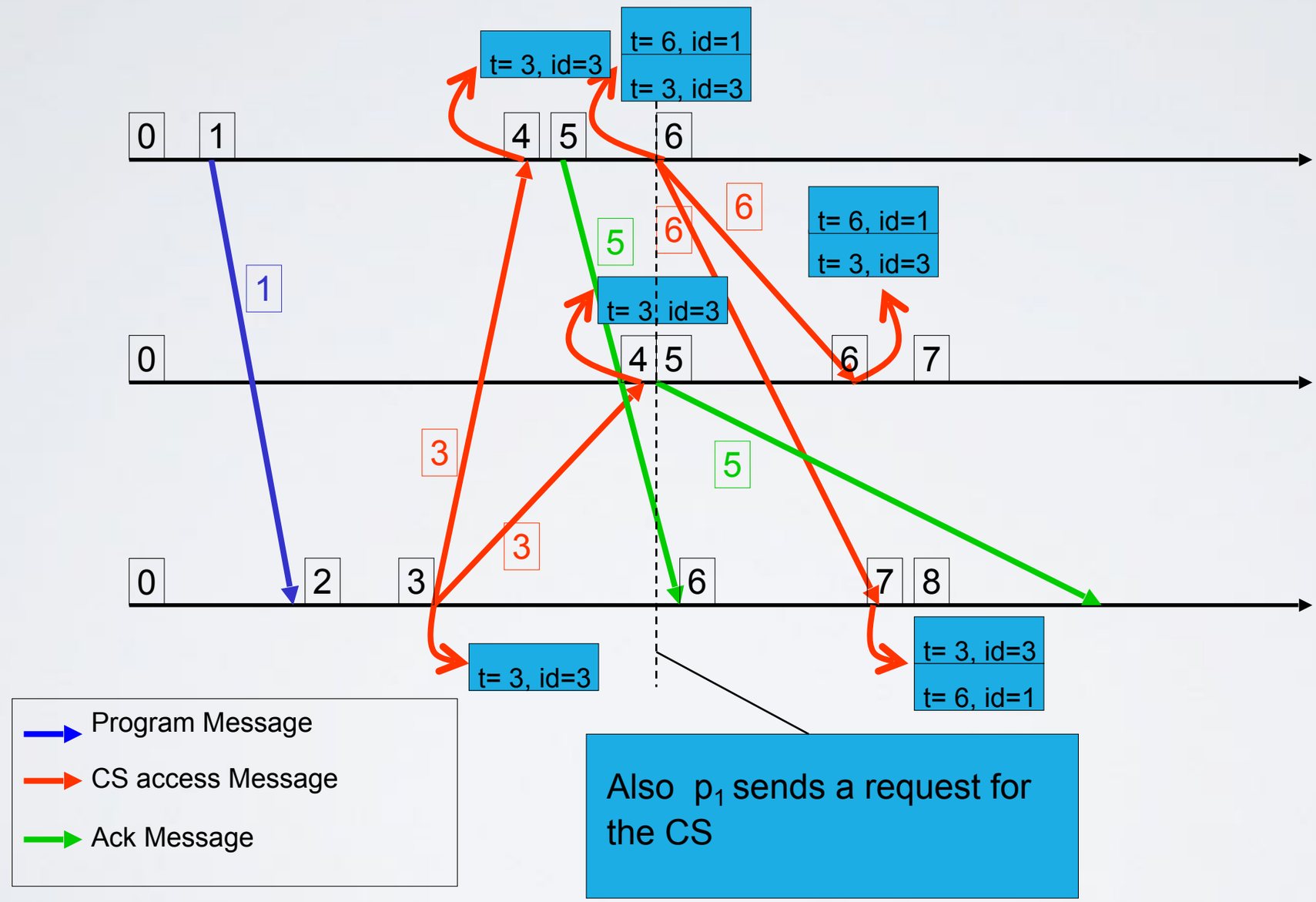
LAMPORT'S ALGORITHM: EXAMPLE



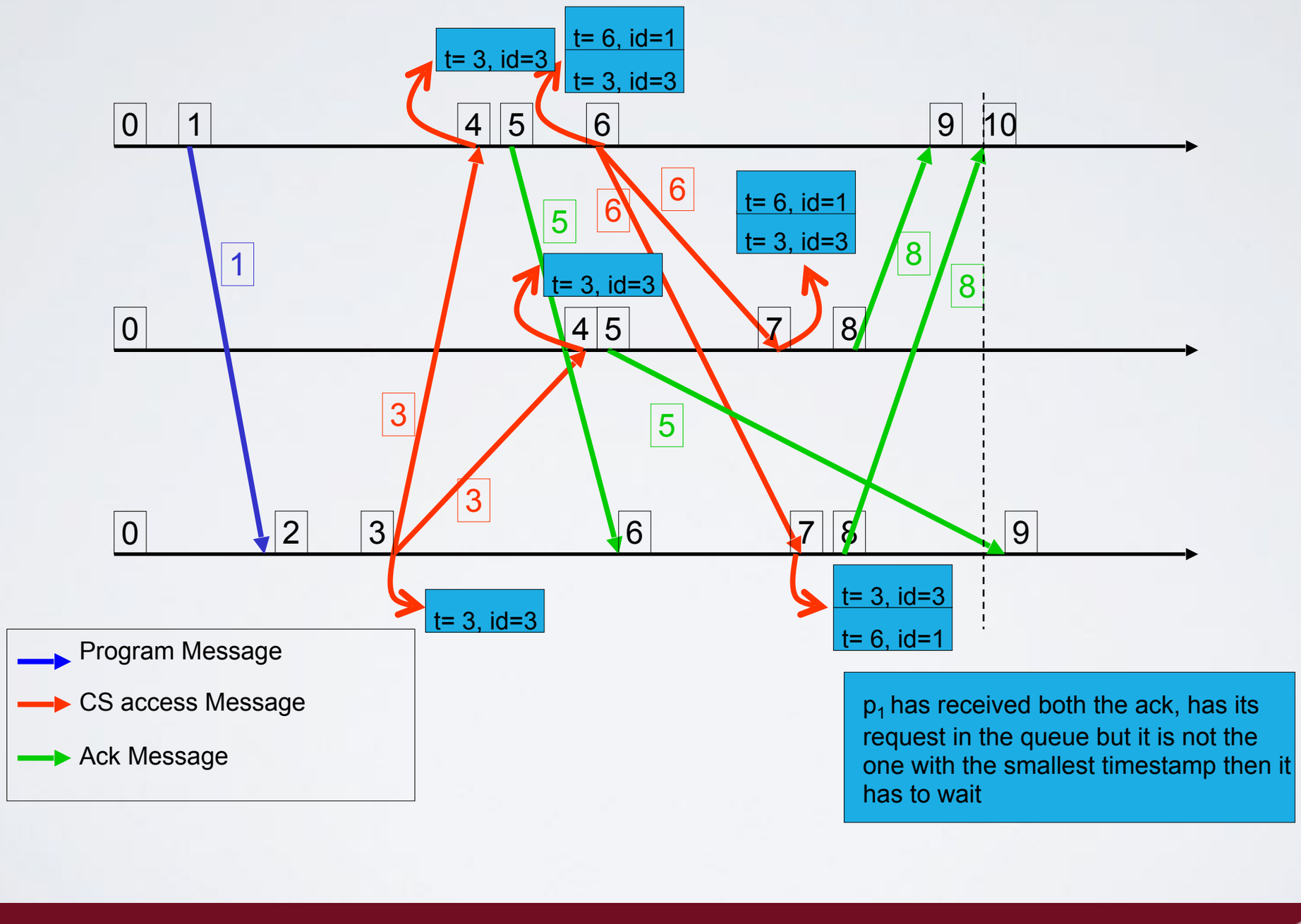
LAMPORT'S ALGORITHM: EXAMPLE



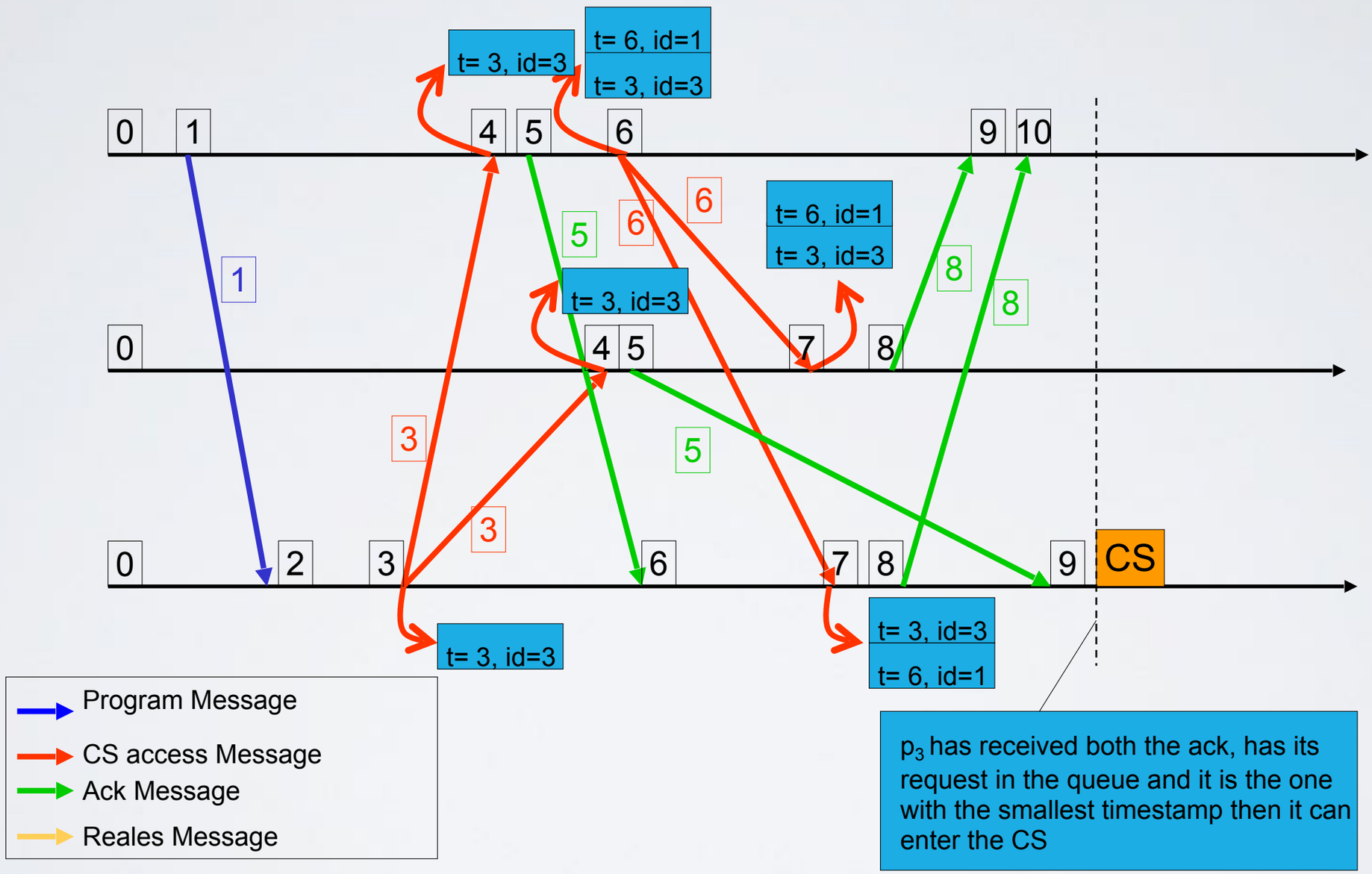
LAMPORT'S ALGORITHM: EXAMPLE



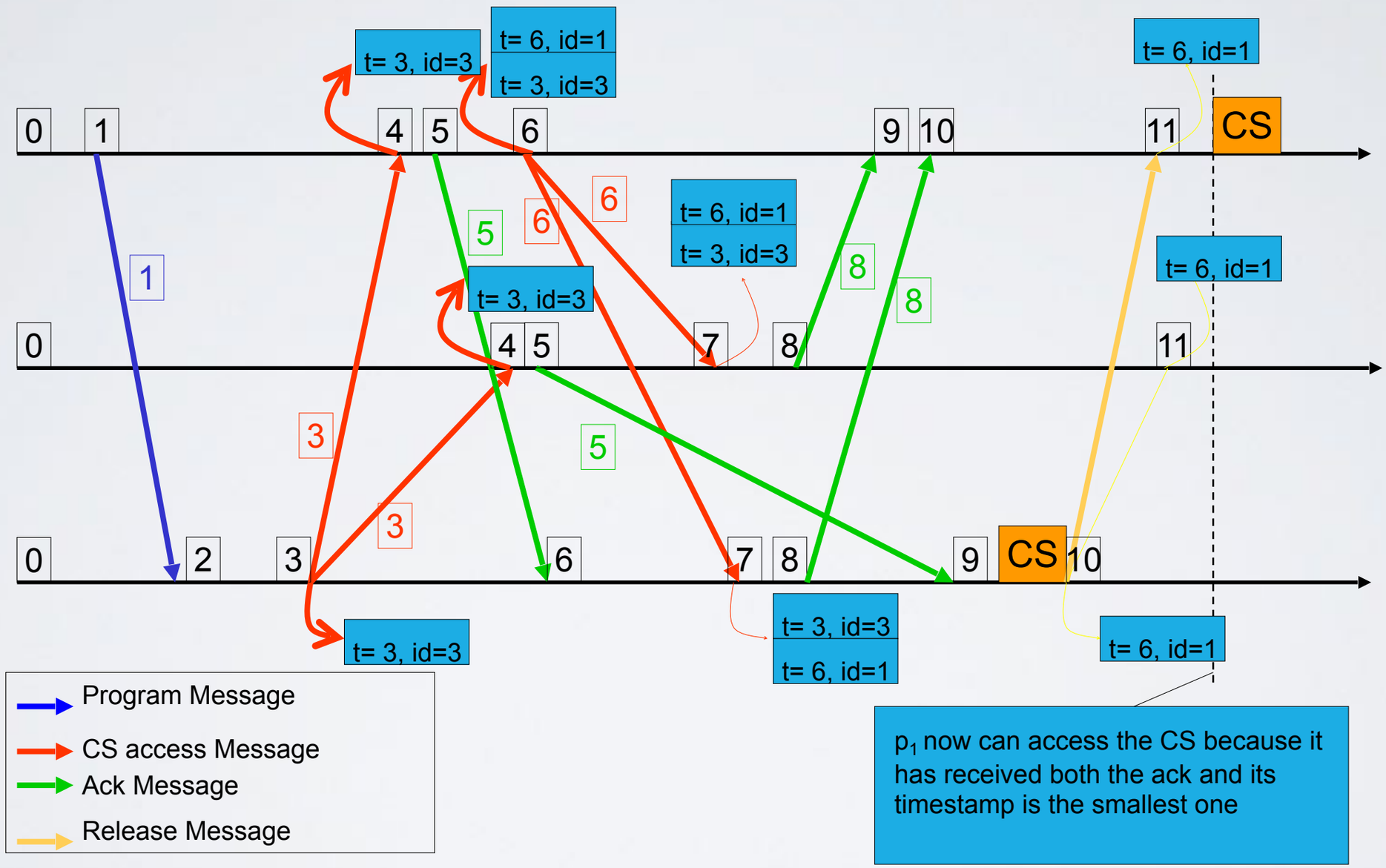
LAMPORT'S ALGORITHM: EXAMPLE



LAMPORT'S ALGORITHM: EXAMPLE



LAMPORT'S ALGORITHM: EXAMPLE



LAMPORT'S ALGORITHM: SAFETY PROOF

(Mutual Exclusion) Assume by contradiction that both p_i and p_j enter the *CS (i < j).*

- You cannot enter CS if you have not received acks from everyone, and such acks happened after your request (when you create a new request its ts is greater then the tss of all old acks).
- ⇒ both the processes have received an ACK from any other process and each my_req is the smallest in the respective queue.
  - p_i received the ack from p_j. When p_j sends the ack to p_i it inserts in its set the req of p_i.
  - p_j received the ack from p_i. When p_i sends the ack to p_j it inserts in its set the req of p_j.
  - If p_j REQUESTS after acking p_i, then ts(req,p_j) > ts(req, p_i). Contradiction. (The same for p_i)
  - So p_j REQUESTS before acking p_i, then by FIFO the (req,p_j) reaches p_i before the ack of p_j. Thus p_i has (req,p_j) in its set and p_j has (req,p_i) in its set. **Since request are total ordered then we have a contradiction.**

LAMPORT'S ALGORITHM: PROPERTIES

(Fairness) different requests are satisfied in the same order as they are generated

- Such order comes from the happened-before relation:

Proof: Suppose p_i enters before p_j, even if (req,p_i) happened after (req,p_j). Since p_i enters only after the ack of p_j, **by FIFO it** sees (req,p_j) before receiving the ack that allows him to enter. Since (req,p_j) happens before (req,p_i) we have ts((req,p_j))< ts((req,p_i)), thus (req,p_i) is not the request with minimal timestamp in the set of p_i.

LAMPORT'S ALGORITHM: PERFORMANCES

Lamport's algorithm needs 3(N-1) messages for the CS execution

- N-1 requests
- N-1 acks
- N-1 releases

In the best case (no one is in the CS and only one process asks for the CS) **there is a delay (from the request to the access) of 2 messages**

HOMEWORKS

In the vector clock we are assuming that each process knows the number, and the IDs of all the other. Assuming that you do not know how many processes are in the system and initially you do not know the id of all the others. Assuming that everyone has an unique ID (no two processes have the same ID). can you implement something that is similar to a vector clock? Could you write the code? What is the peculiarity of these special vector clocks associated with two events that are concurrent?

**What happens if just two processes have the same ID?

HOMEWORKS

Suppose we run Lamport's Mutual Exclusion (ME) algorithm in a system where links are not FIFO. For each property of Lamport's ME algorithm, discuss whether it is violated. If a property is violated, provide an execution example, including a detailed space-time diagram and comments on the internal state of each process, that demonstrates the violation. If a property is not violated, explain why it remains intact.