



Distributed Systems

Clocks & Time II

Olaf Landsiedel

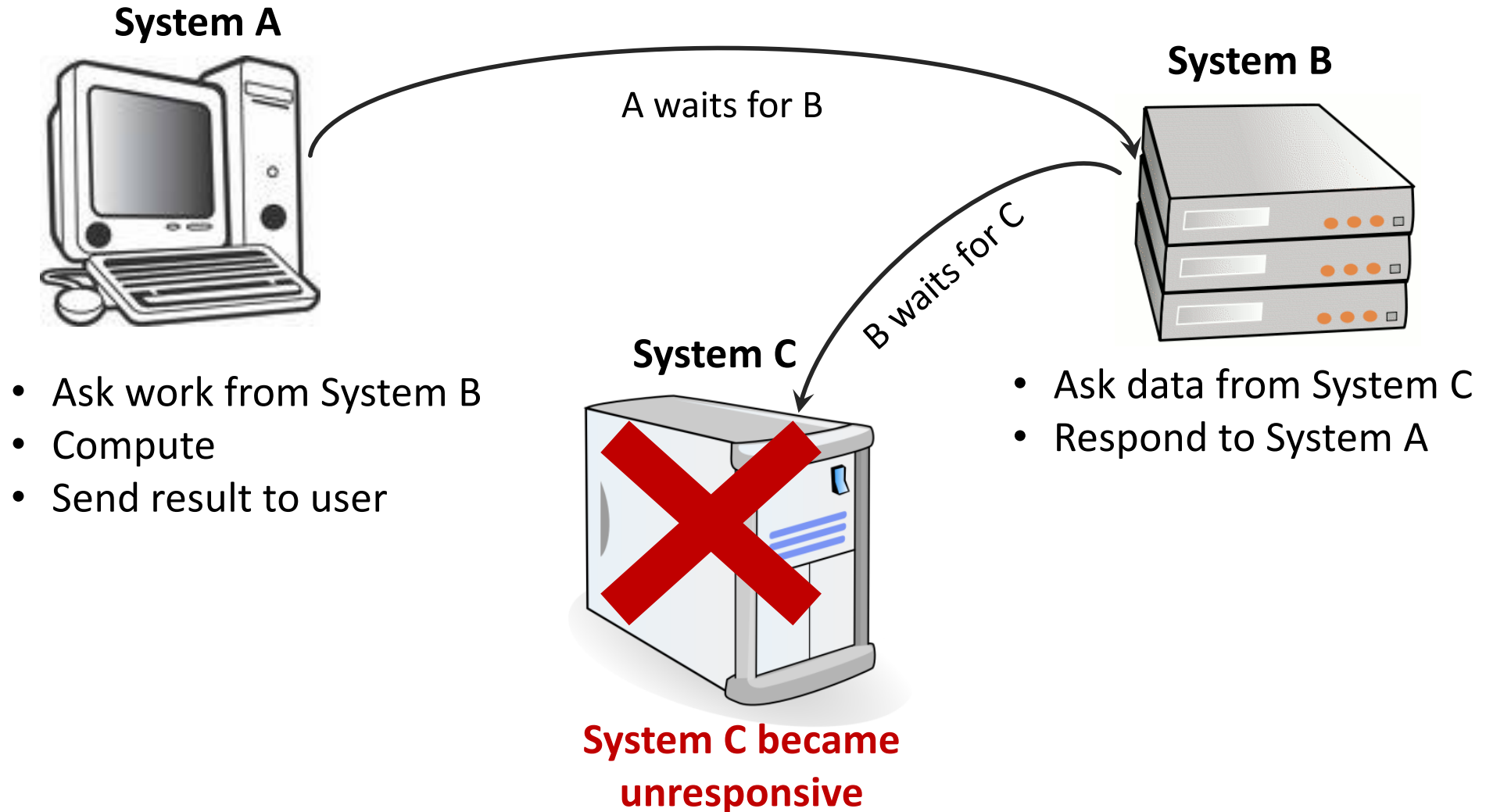
Last Time

- Topic: Clocks and Time I
 - Physical Clocks
 - Main lessons learned from last time?

Asynchronous Distributed System?

- Define:
 - Asynchronous Distributed System
 - > no notion of a global clock

A Distributed Computation With Data Dependencies



Detecting and Fixing the Problem

- User gets no response from System A
- How do we detect that System C is causing the problem?
- Easy to detect if we get a snapshot of a global state
- Constructing global state in an asynchronous distributed system is a difficult problem

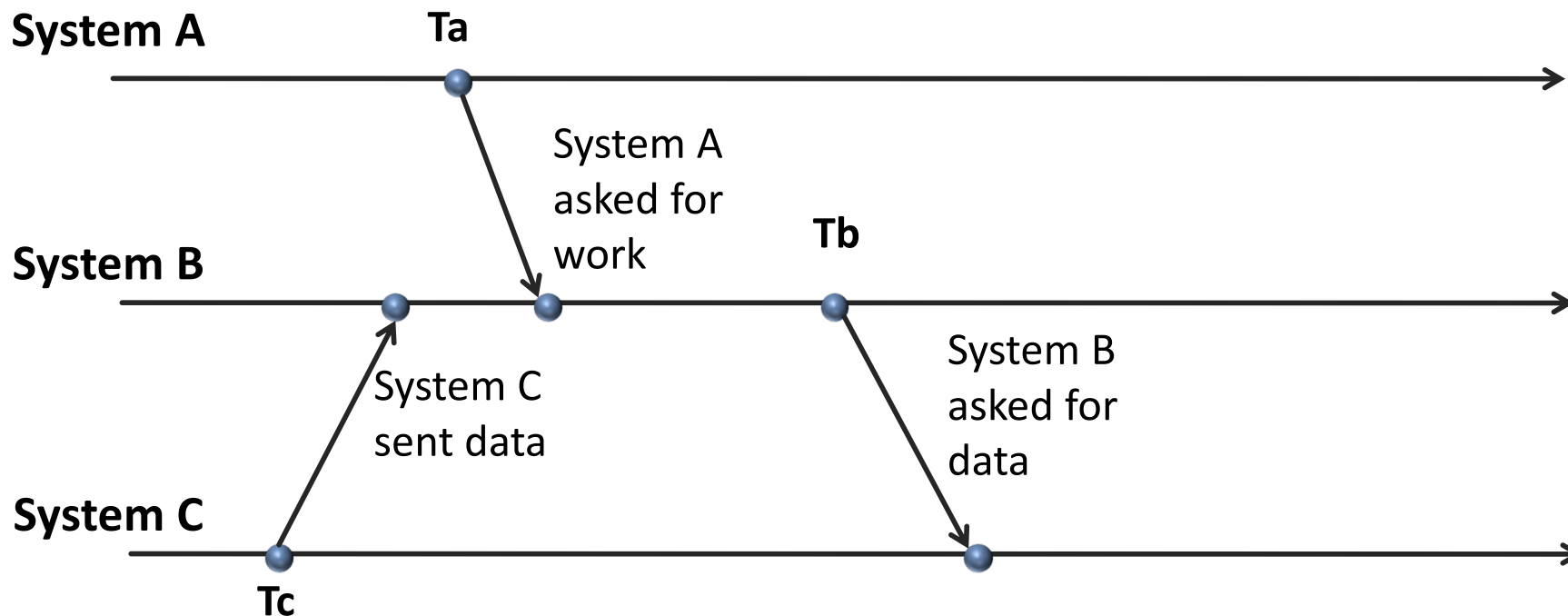
A Naïve Approach

- Each system records each event it performed and its timestamp
- Suppose events in the this system happened in this real order:
 - **Time Tc0:** System C sent data to System B (before C stopped responding)
 - **Time Ta0:** System A asked for work from System B
 - **Time Tb0:** System B asked for data from System C



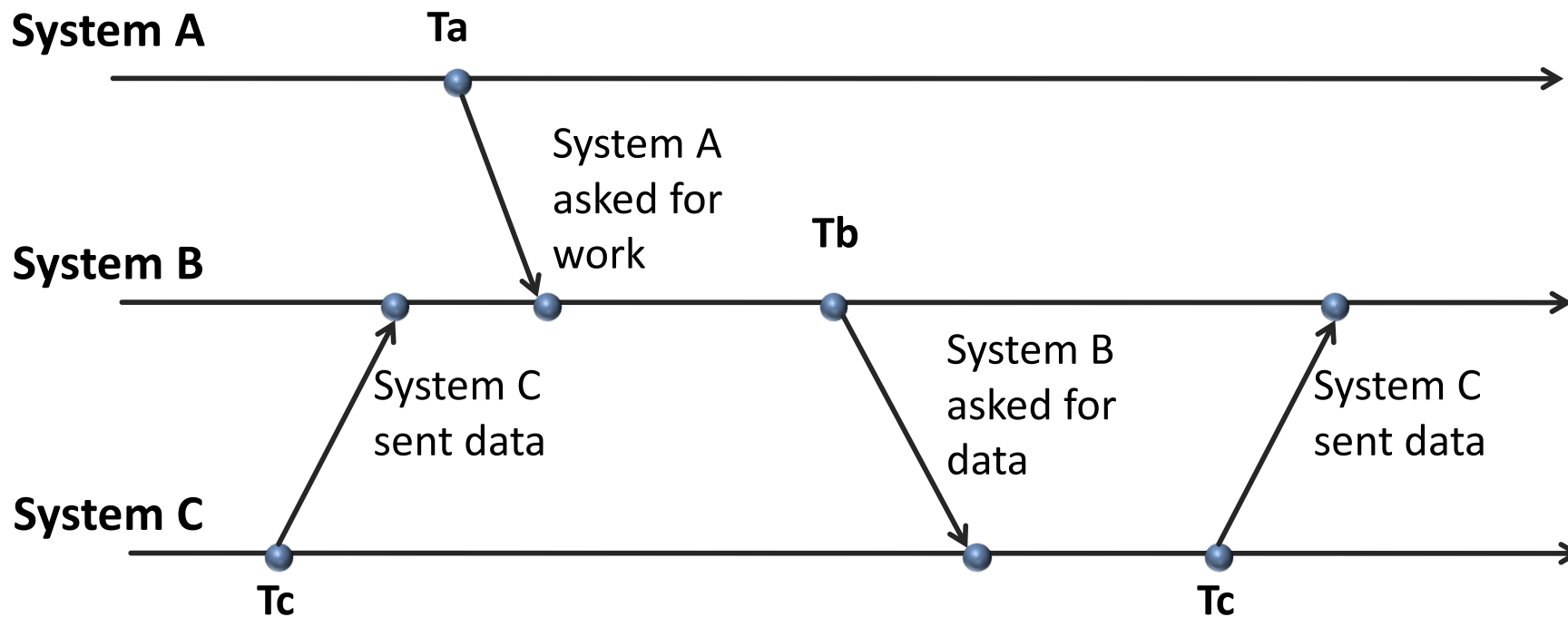
A Naïve Approach (cont)

- Ideally, we will construct real order of events from local timestamps and detect this dependency chain:



A Naïve Approach (cont)

- But in reality, we do not know if T_c occurred before T_a and T_b , because in an asynchronous distributed system clocks are not synchronized!



This Lecture's Topic

- We will learn how to solve this problem in an asynchronous distributed system
- Our goal is to construct a consistent global state in presence of unsynchronized local clocks
- We will begin by creating a formal model for
 - Distributed computation
 - Event precedence in a distributed computation
- We will use a event precedence model to define consistent global state
- Then we will learn to construct consistent global states

Model of a Distributed Computation

- Collection of processes p_0, \dots, p_n
- Processes communicate by sending messages
- Event *send*(m) enqueues message m on the sending end
- Event *receive*(m) dequeues message m on the receiving end
- Events $e_i^0, e_i^1, \dots, e_i^n$ are steps of a distributed computation at process p_i
- $h_i = e_i^0, e_i^1, \dots, e_i^n$ is a local history of p_i

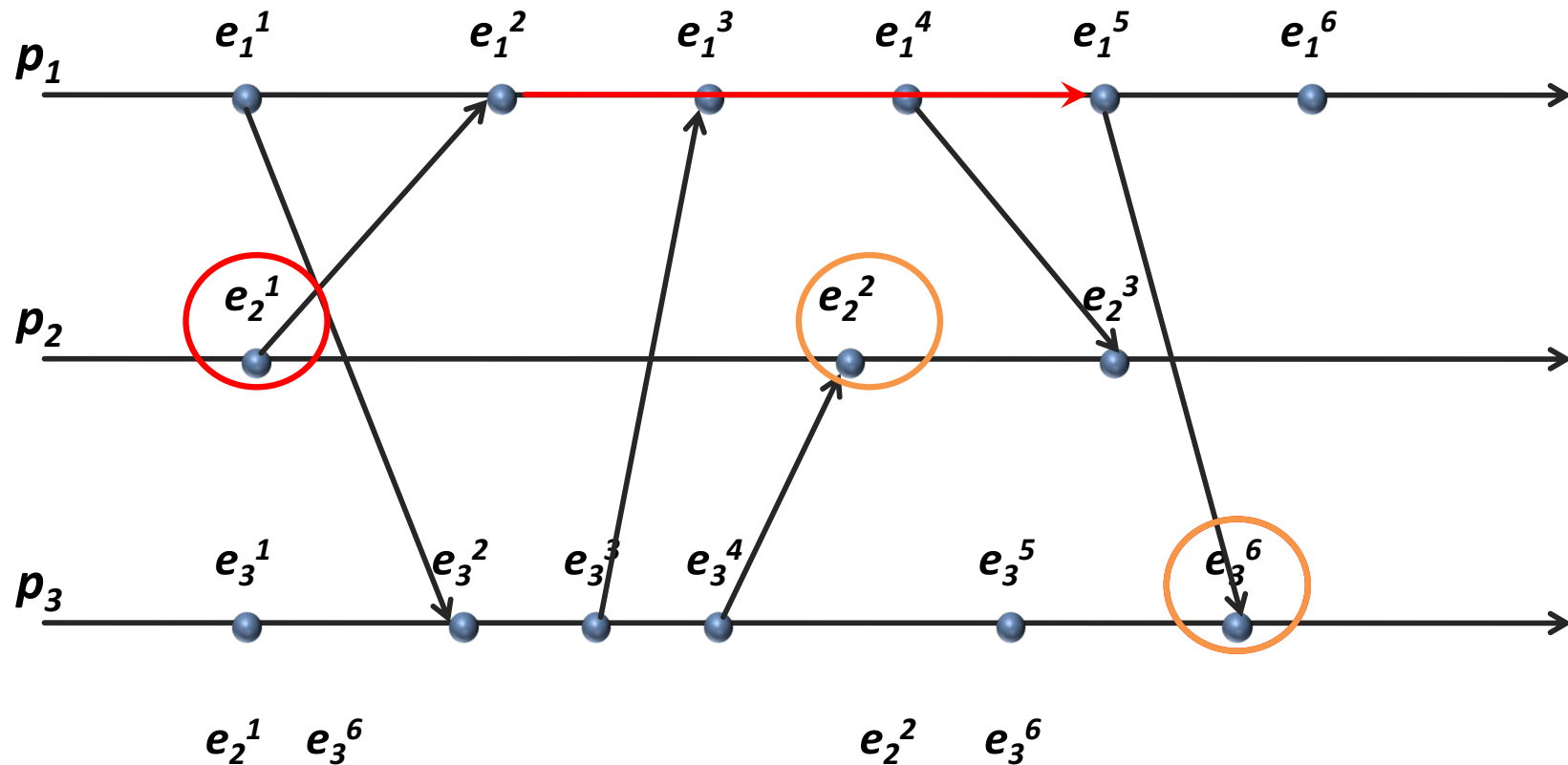
Events and Causal Precedence

- In an asynchronous distributed system there is no notion of global clock
- Events cannot be ordered according to some real order
- We can only talk about ***causal event ordering***, that is if one event affects the outcome of another event
- For example events *send(m)* and *receive(m)* are in causal order, because *receive(m)* could not have occurred before *send(m)*

Formal Rules for Ordering of Events

- If local events precede one another, then they also precede one another in the global ordering:
 - On node i : If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$
- Sending a message always precedes receipt of that message:
 - If $e_i = \text{send}(m)$ and $e_j = \text{receive}(m)$, then $e_i \rightarrow e_j$
- Event ordering is transitive:
 - If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

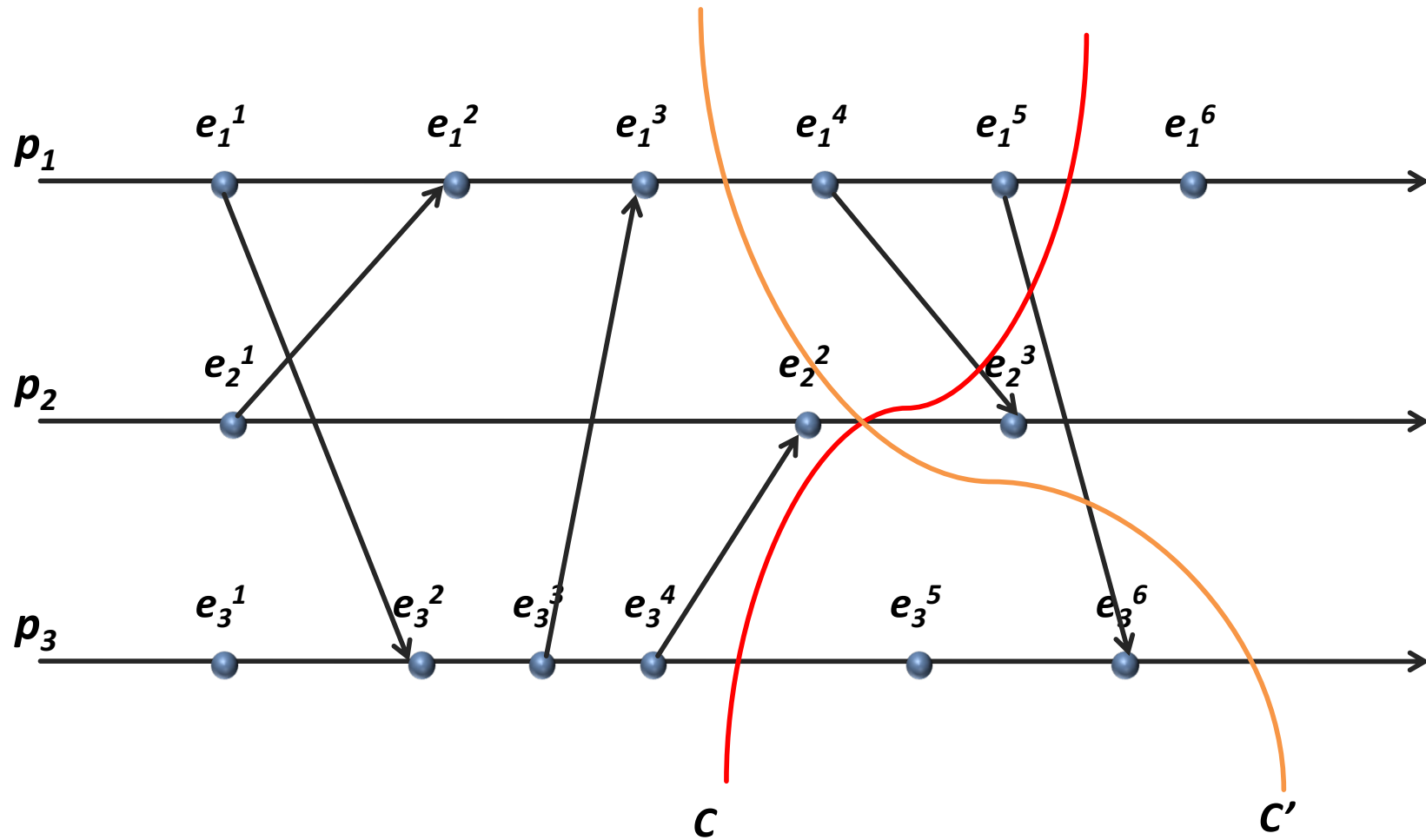
Space-time Diagram of a Distributed Computation



Cuts of a Distributed Computation

- Suppose there is an ***external monitor*** process
- External monitor constructs a global state:
 - Asks processes to send it local history
- Global state constructed from these local histories is a ***cut of a distributed computation***

Example Cuts



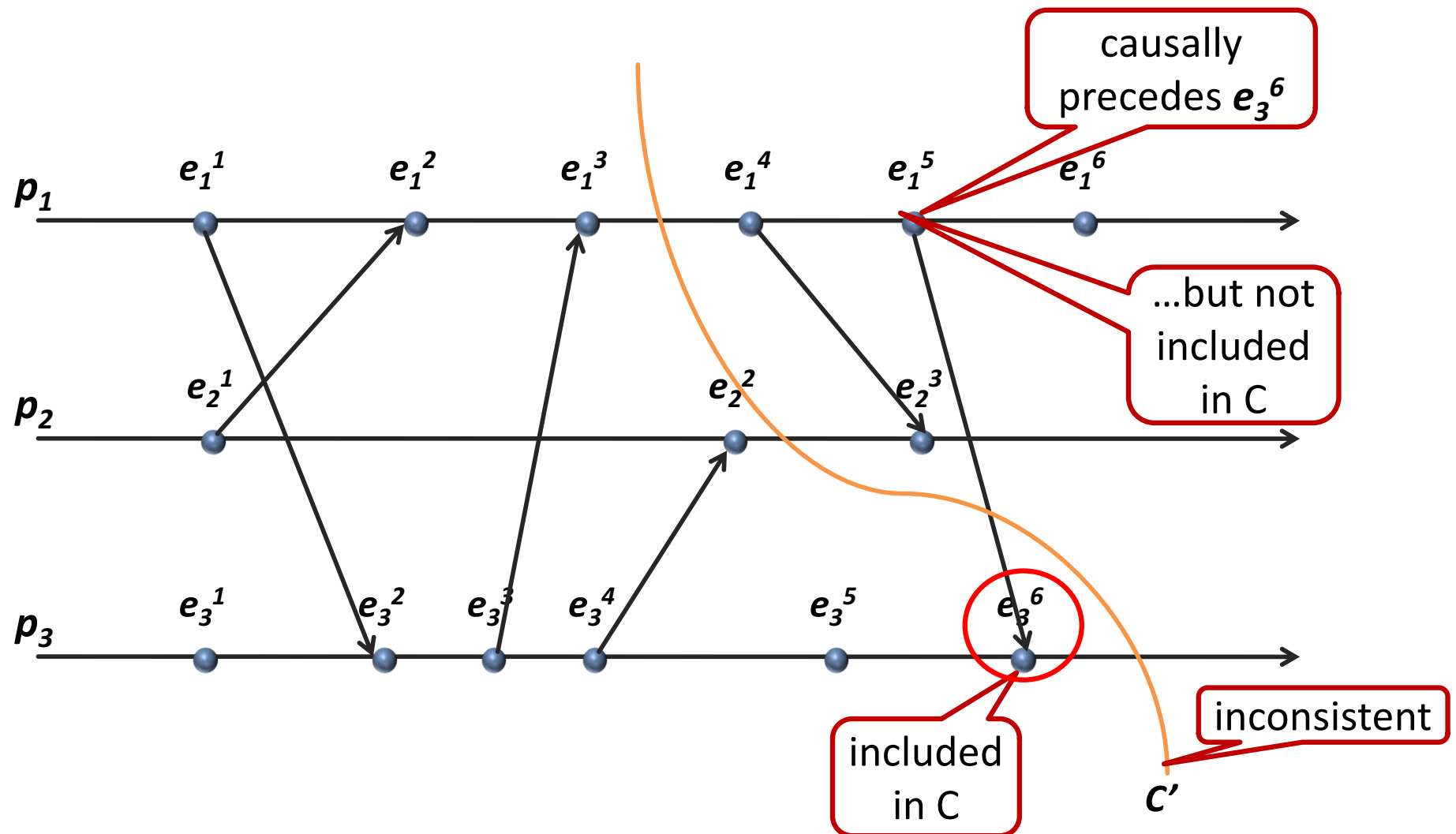
Consistent vs. Inconsistent Cuts

- A cut is consistent if for any event e included in the cut, any event e' that causally precedes e is also included in that cut

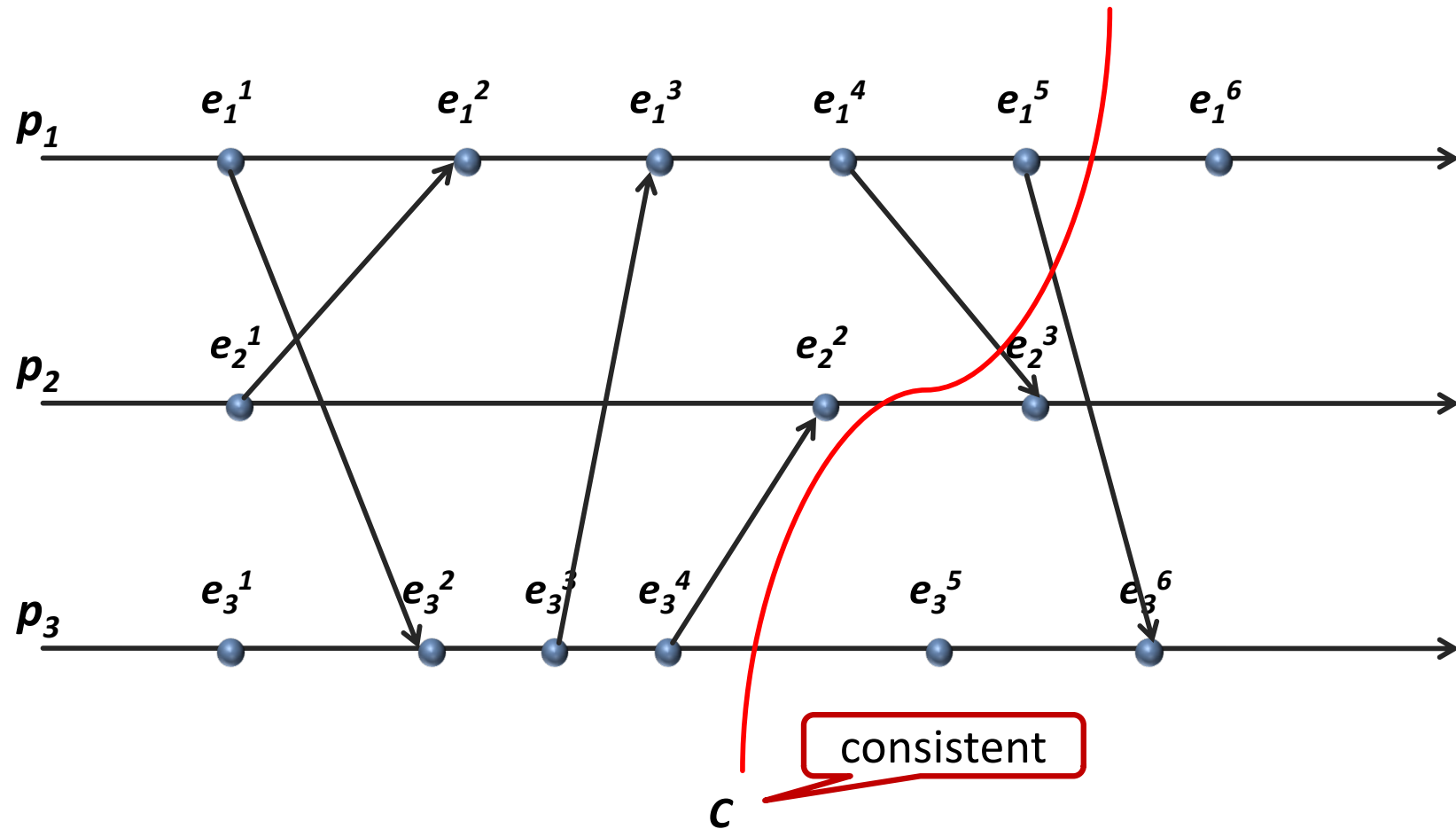
- For cut C :

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

Are These Cuts Consistent?



Are These Cuts Consistent?



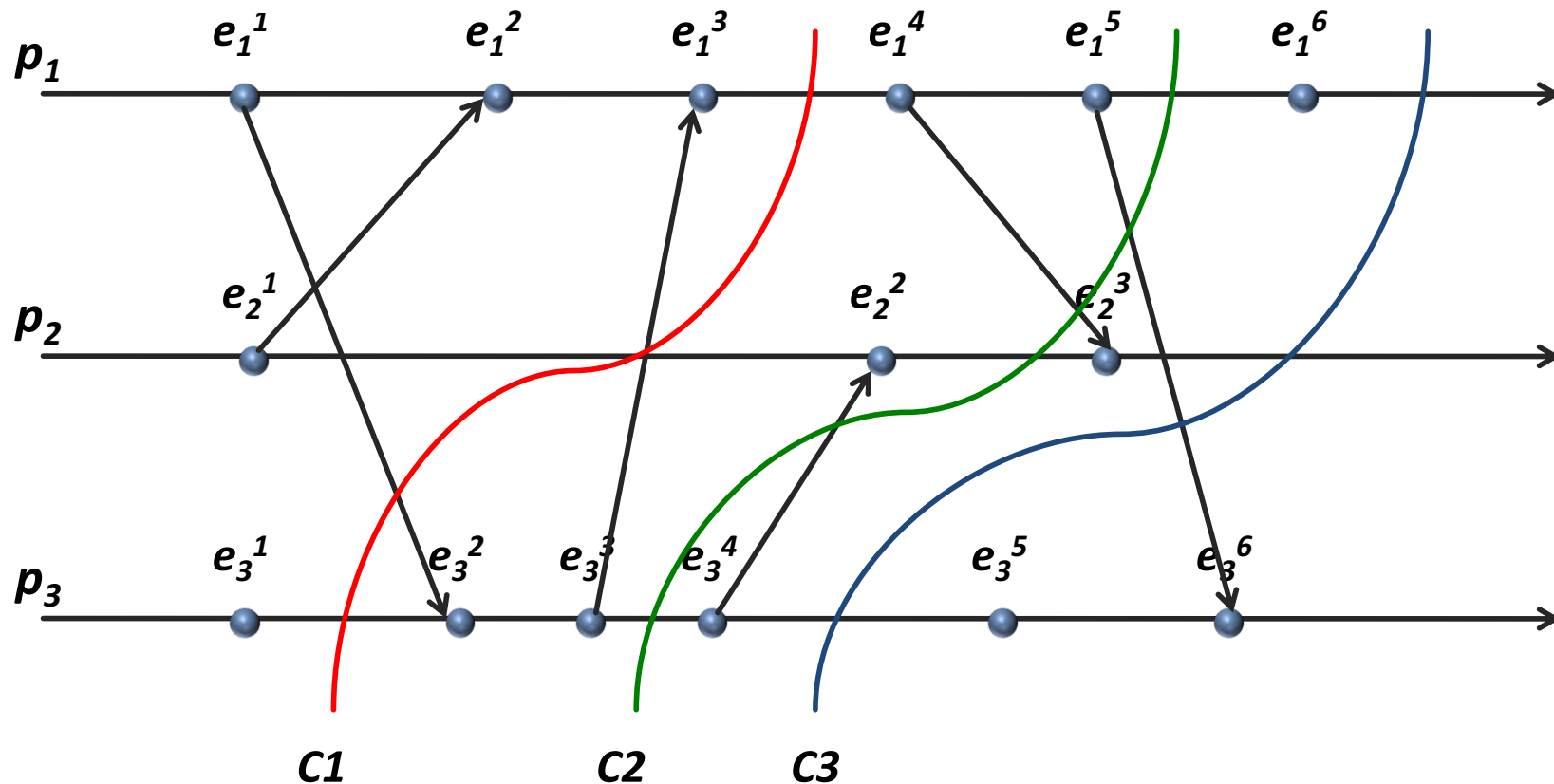
Questions

- Is event ordering transitive?
 - $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$
- When is a cut consistent?
 - $(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$



Are These Cuts Consistent?

olafland.polldaddy.com/s/cuts





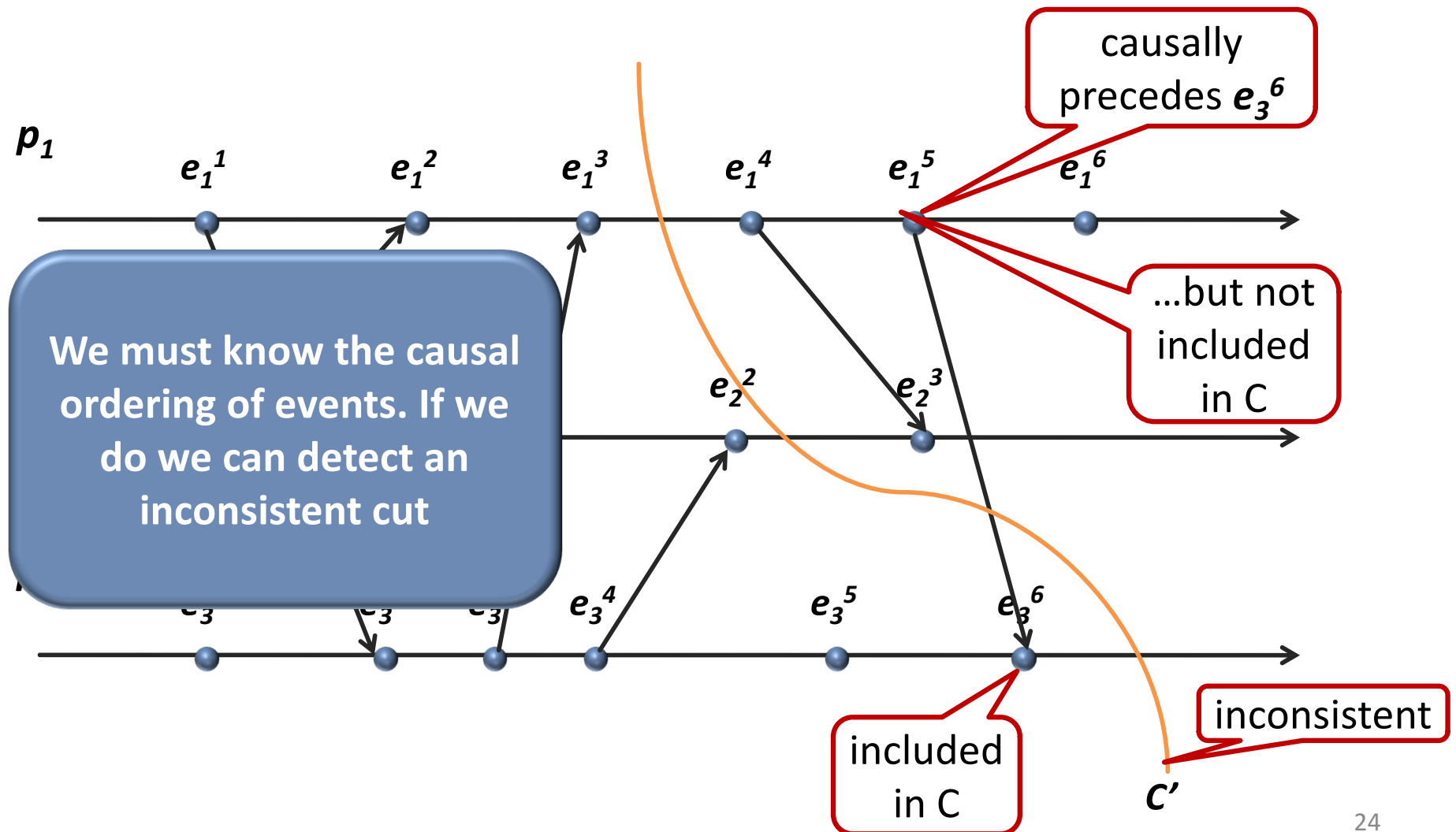
Cuts and Global State

- Recall that our goal is to construct a consistent global state
- We'd like to get a snapshot of a distributed system as would have been observed by a perfect observer
- ***A consistent cut corresponds to a consistent global state***
- What's next: we will learn how to construct consistent cuts in the absence of synchronized clocks
- Consistent global states can be constructed via:
 - Active monitoring
 - Passive monitoring

Passive vs. Active Monitoring

- **Common settings**
 - There is a process p_0 external to the distributed computation
 - There are processes p_i ($1 \leq i \leq n$), part of the computation
- **Passive monitoring:**
 - Each process p_i sends to p_0 a timestamp of each event it executes
- **Active monitoring:**
 - When p_0 desires to find out the state of the system it asks all processes p_i to send its state

What Do We Need to Know to Construct a Consistent Cut?

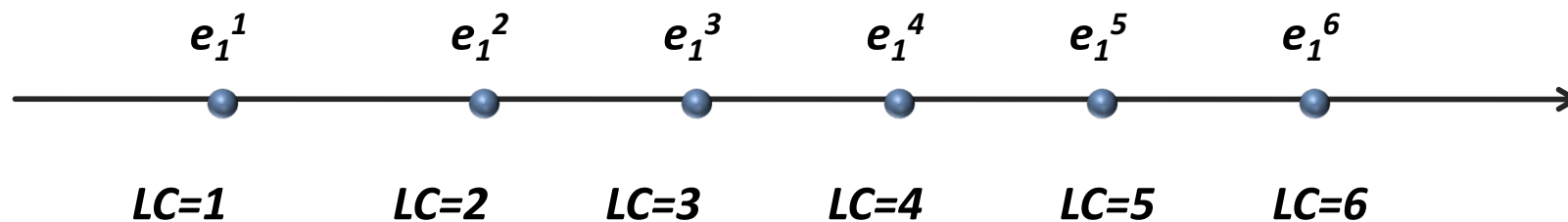


Constructing a Consistent Cut

- We must know the causal ordering of events
- If there was a global clock, we would detect this as follows:
- $RC(e)$ – timestamp of event e given by a global clock
- Clock condition:
$$e' \rightarrow e \Rightarrow RC(e') < RC(e)$$
- **What to do if there is no global clock?**
- We will use **logical clocks** and **vector clocks**

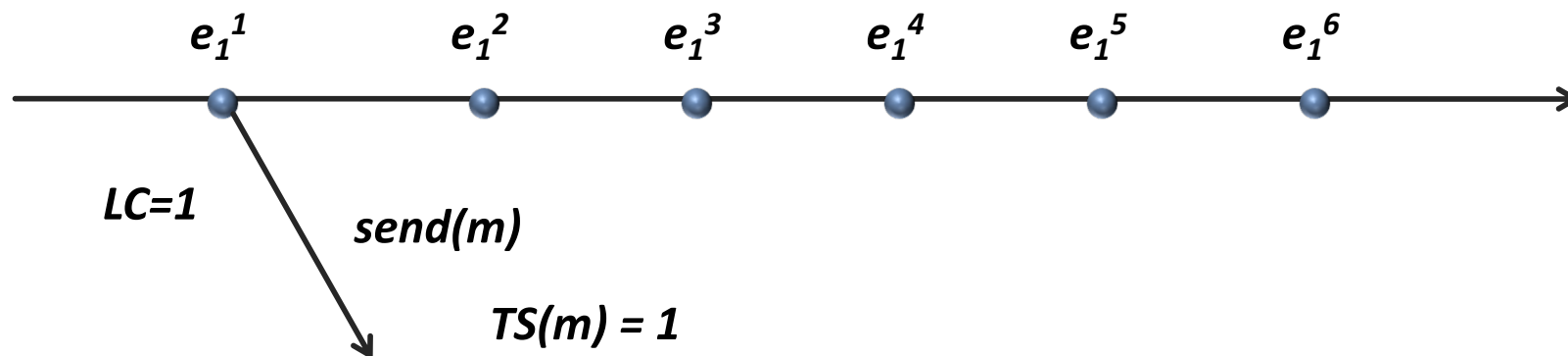
Logical Clocks

- Each process maintains a local value of a logical clock **LC**
- Logical clock of process p counts **how many events in a distributed computation causally preceded the current event at p** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



Logical Clocks (cont.)

- In a system with more than one process logical clocks are updated as follows:
- Each message m that is sent contains a timestamp $TS(m)$
- $TS(m)$ is the logical clock value associated with sending event at the sending process



Logical Clocks (cont)

- When the receiving process receives message m , it updates its logical clock to:

$$\max\{LC, TS(m)\} + 1$$

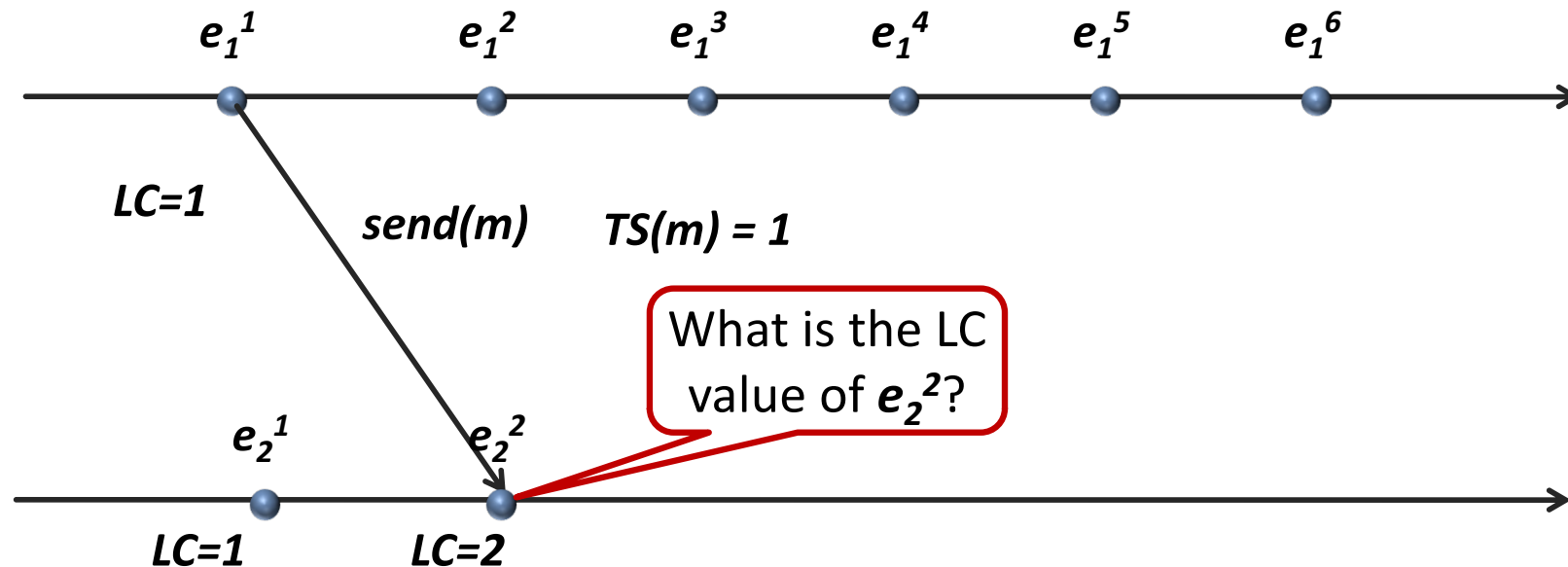
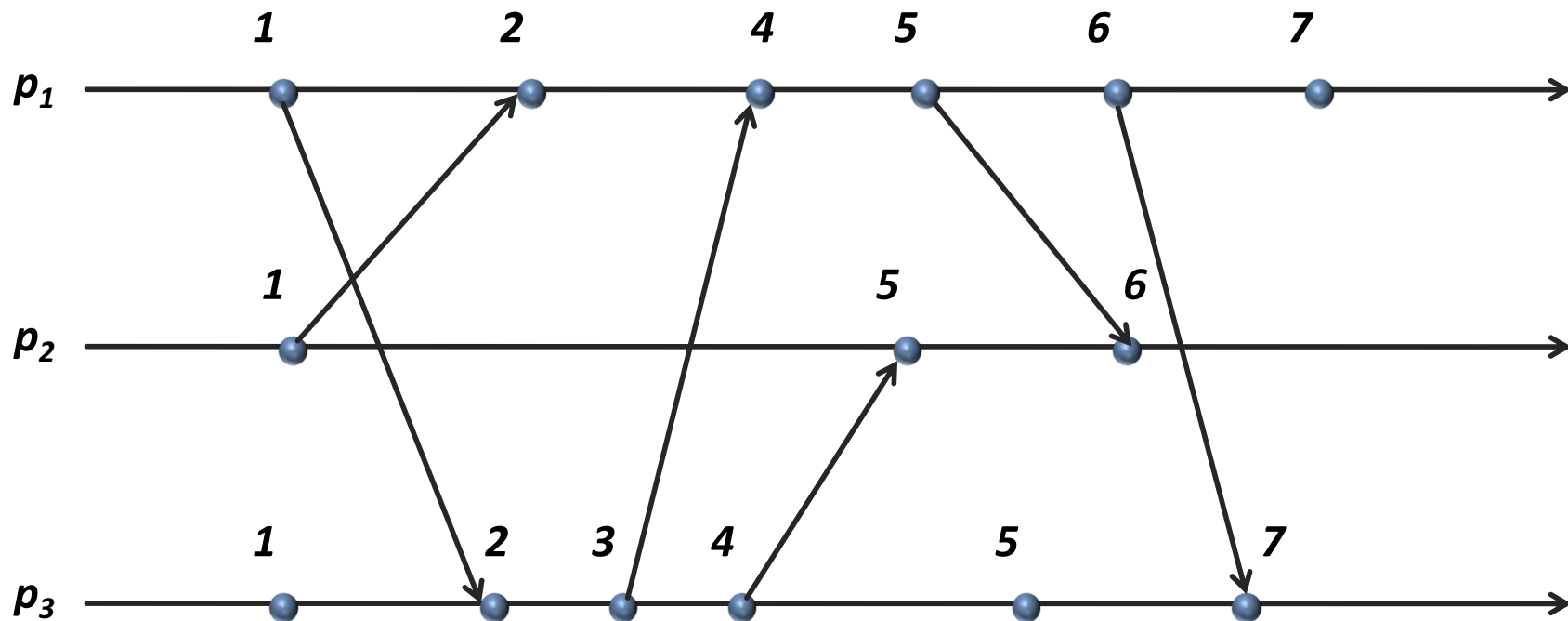


Illustration of a Logical Clock



Questions

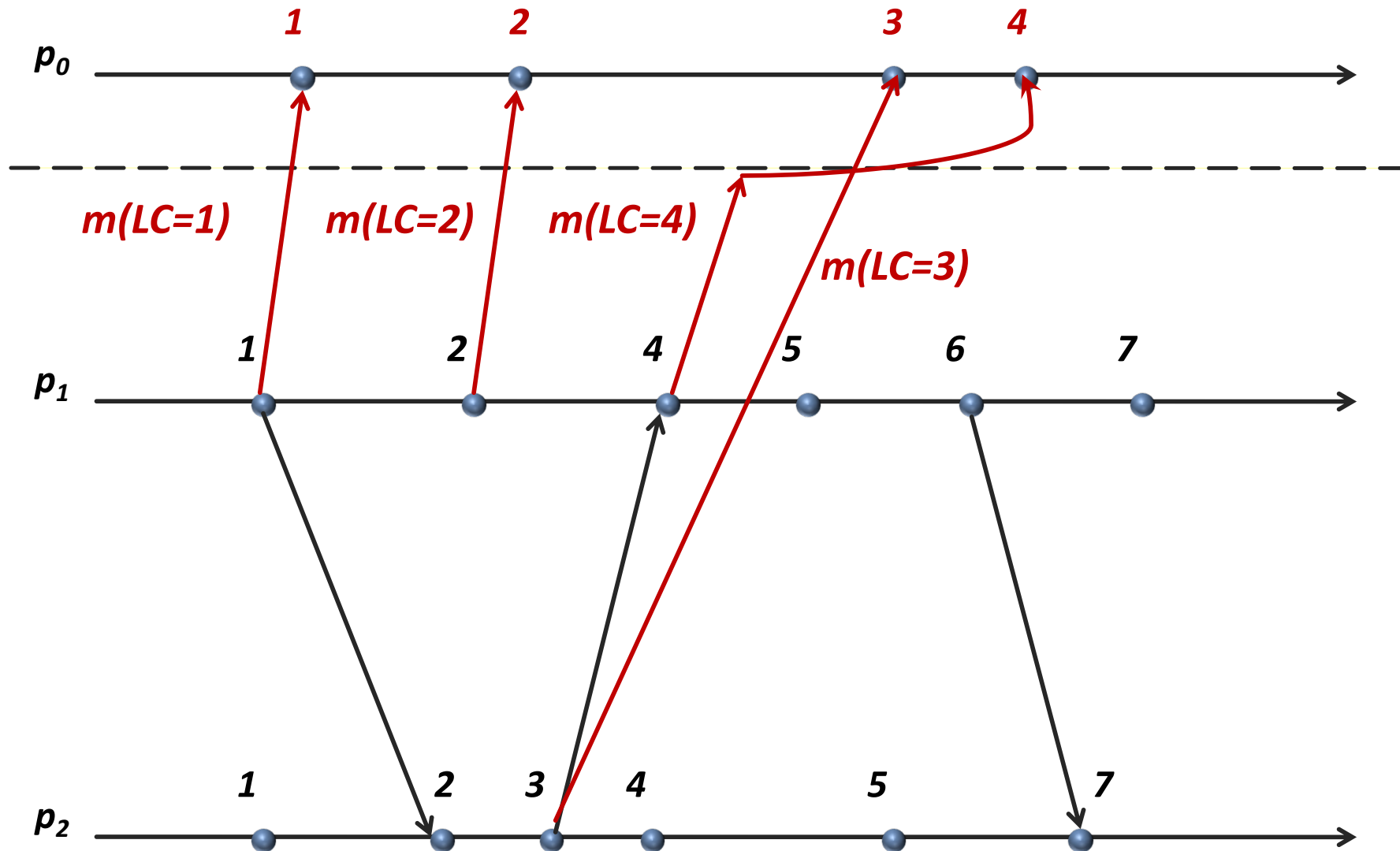
- Logical clock: when a node receives a msg?
 - Max of local and received value, inc. by one
- 2 nodes, by which value can their logical clocks differ the most?
 - Arbitrary large

Causal Delivery with Logical Clocks

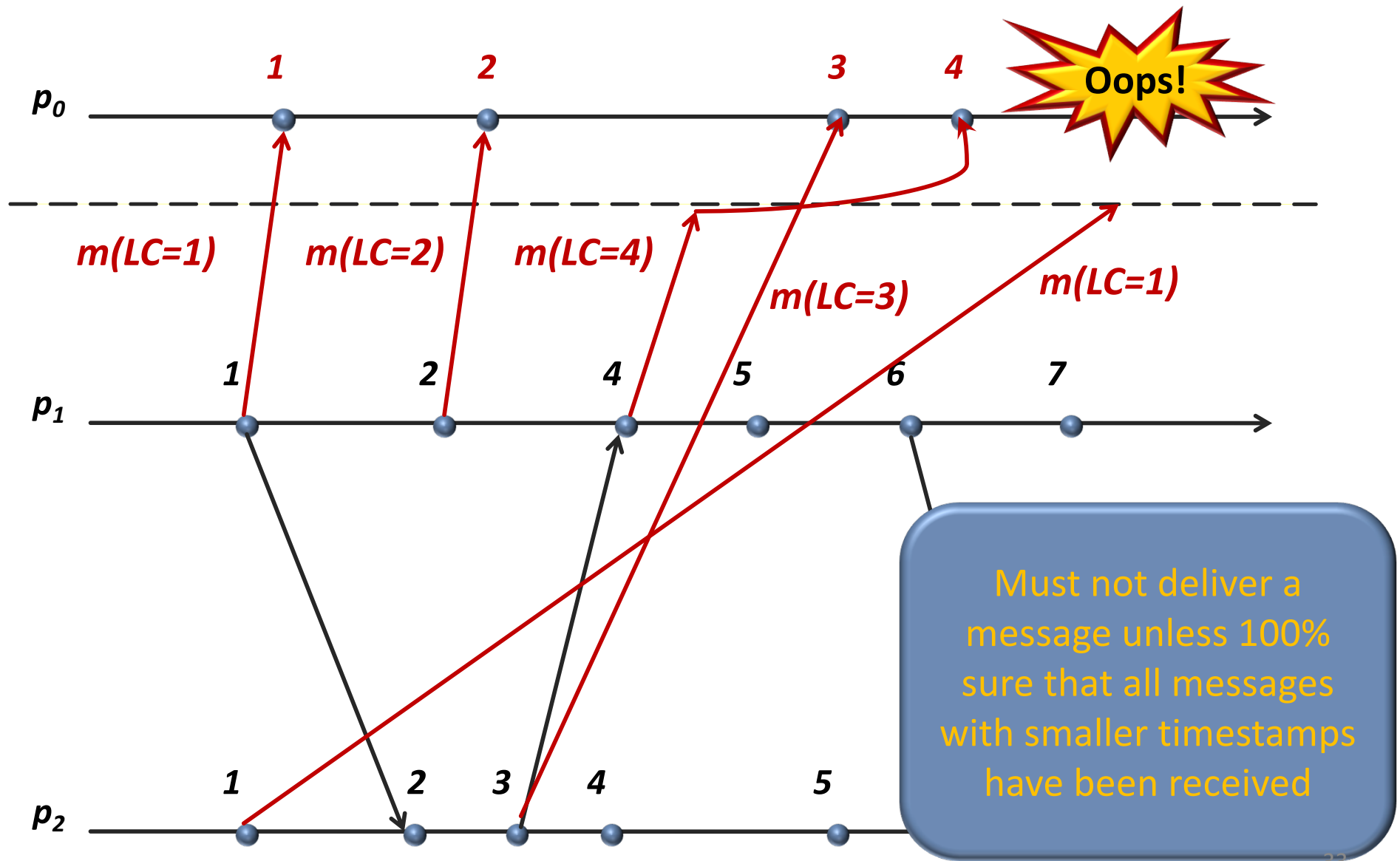
- We have a external monitor process p_0
- Each process p_i sends an **event notification** message m to p_0 after each event e_i
- To let p_0 construct a correct causal ordering of events, we define the following delivery rule for event notification messages:

Causal delivery rule: Event notification messages are delivered to p_0 in the increasing logical clock timestamp order

Causal Delivery Rule



Gap Detection Rule



Rules to Achieve Causal Delivery

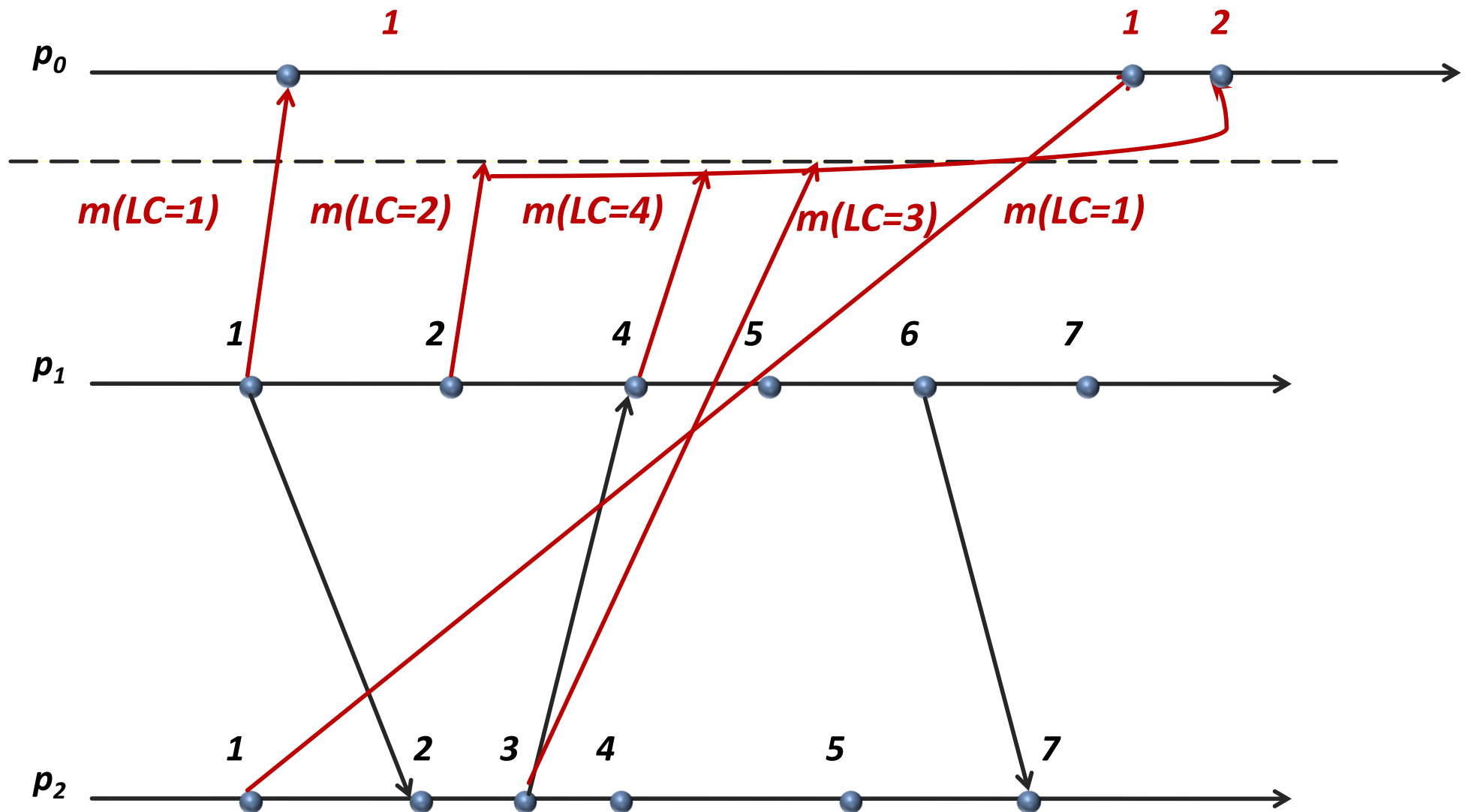
- **Rule #1:**

Do not deliver event notification message from a process p_i unless all messages with smaller timestamps have been delivered (FIFO delivery)

- **Rule #2:**

Do not deliver to p_0 event notification message with a timestamp $TS(m)$ unless p_0 received messages with timestamps one smaller, equal or greater than $TS(m)$ from all other processes

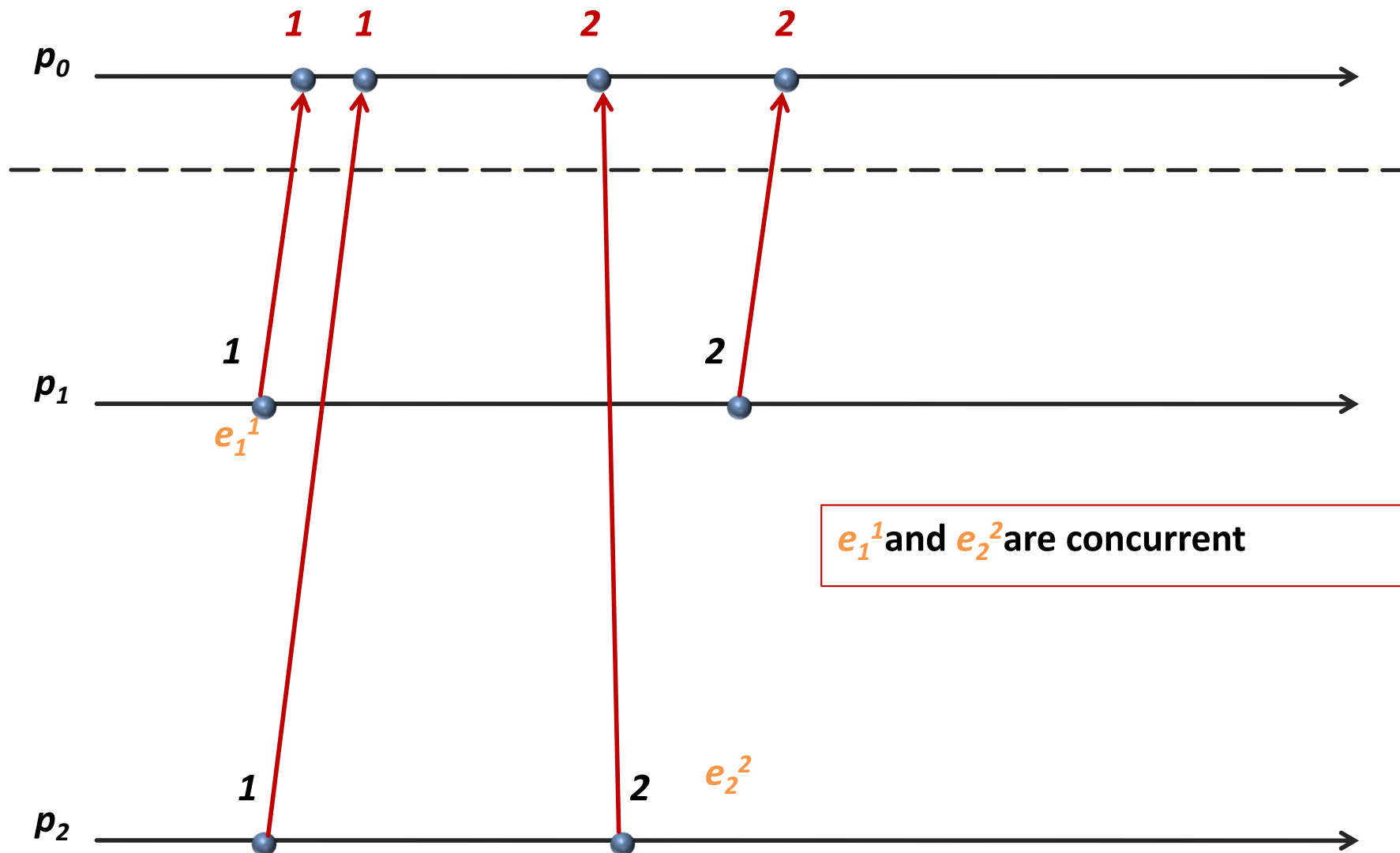
Causal Delivery Rules



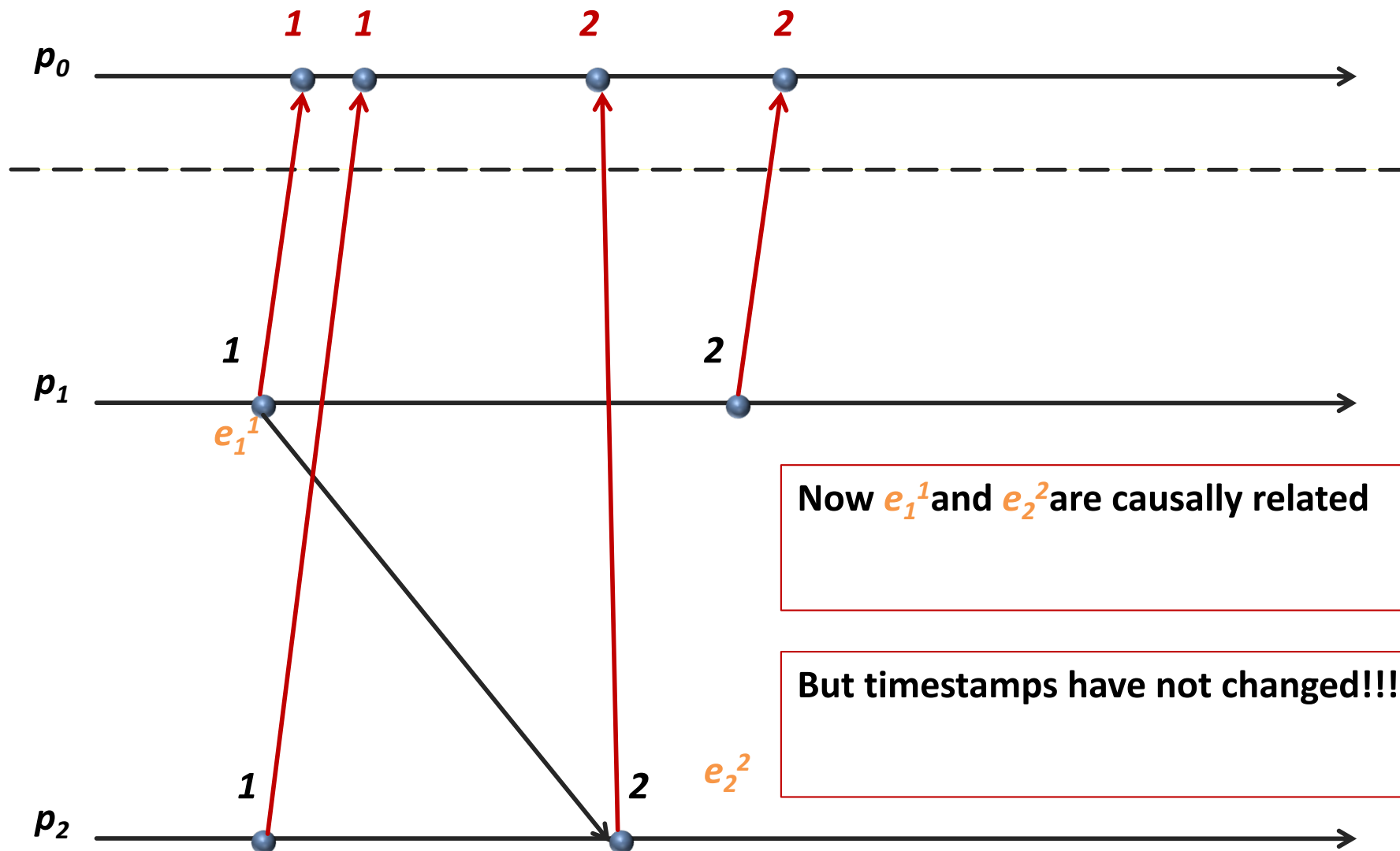
Causal Precedence Relation

- Detecting causal precedence relationship with logical clocks requires sending more than just timestamps!

Causal Precedence or Concurrency?



Causal Precedence or Concurrency?



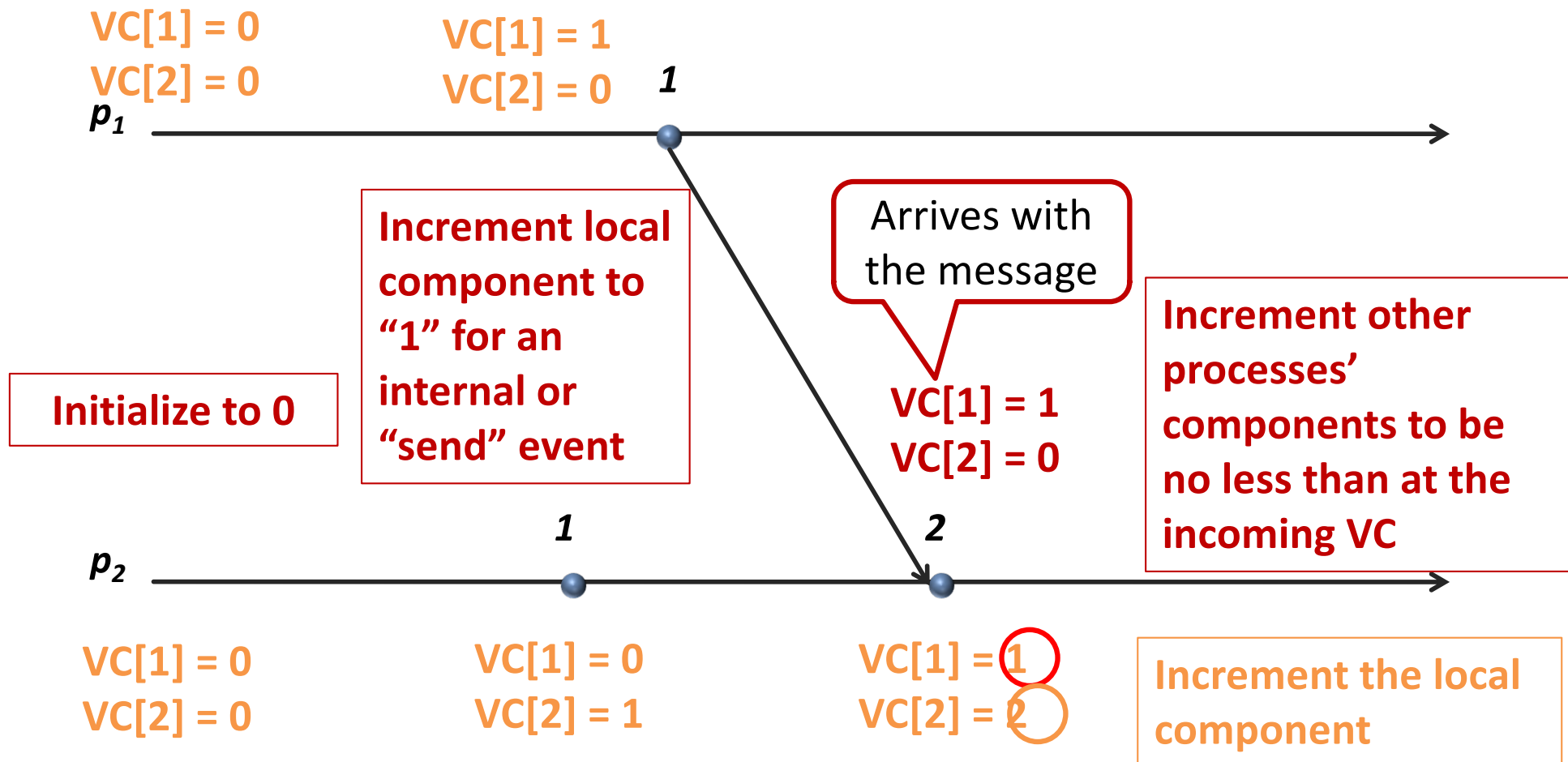
Causal Precedence Relation

- Using just timestamps from logical clocks does not help us detect whether two events are causally related or concurrent
- We could fix this by including a local history in the event notification message
- Not a good idea: messages will become large
- A better idea: **vector clocks**

Introduction to Vector Clocks

- The clock of each process is represented by a vector
- The vector dimension is the number of processes in the system
- Each entry of a vector clock corresponds to a process
- Suppose there are three processes: p_1 , p_2 , and p_3
- Then the vector clock at each process will look like this:
 - VC[1] – entry for process p_1
 - VC[2] – entry for process p_2
 - VC[3] – entry for process p_3

Using Vector Clocks



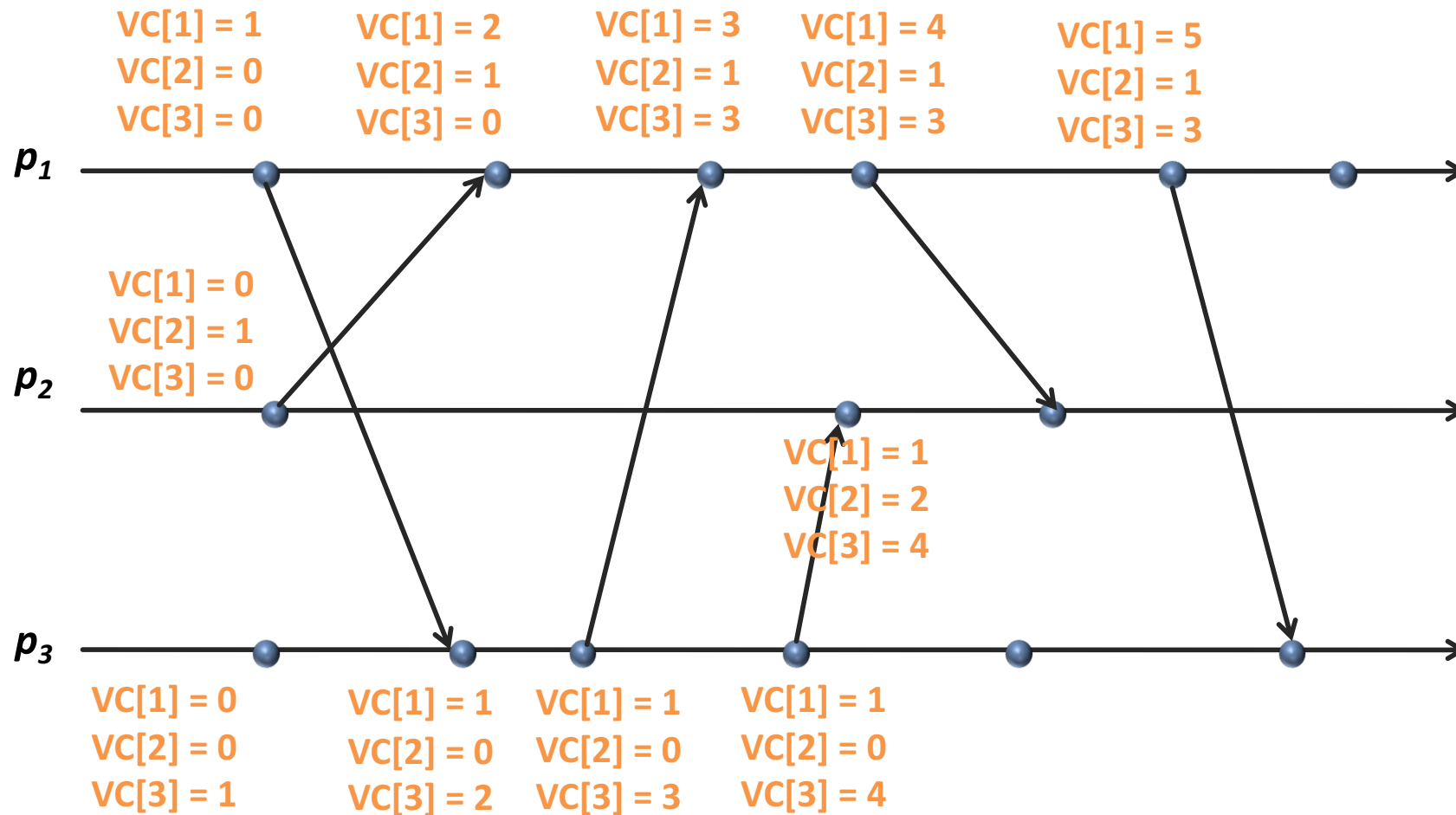
The Meaning of Vector Clocks

- An entry of a vector clock for process p_i is **the number of events at p_i**
VC[1] = 3 – number of events at p_1
VC[2] = 0 – number of events at p_2
VC[3] = 1 – number of events at p_3
- p_1 knows for sure how many events it executed
- But how can it be sure about p_2 and p_3 ? It can't!
- So p_1 's entries for p_2 and p_3 are...
what p_1 **thinks** about p_2 and p_3

Meaning of Vector Clocks (cont)

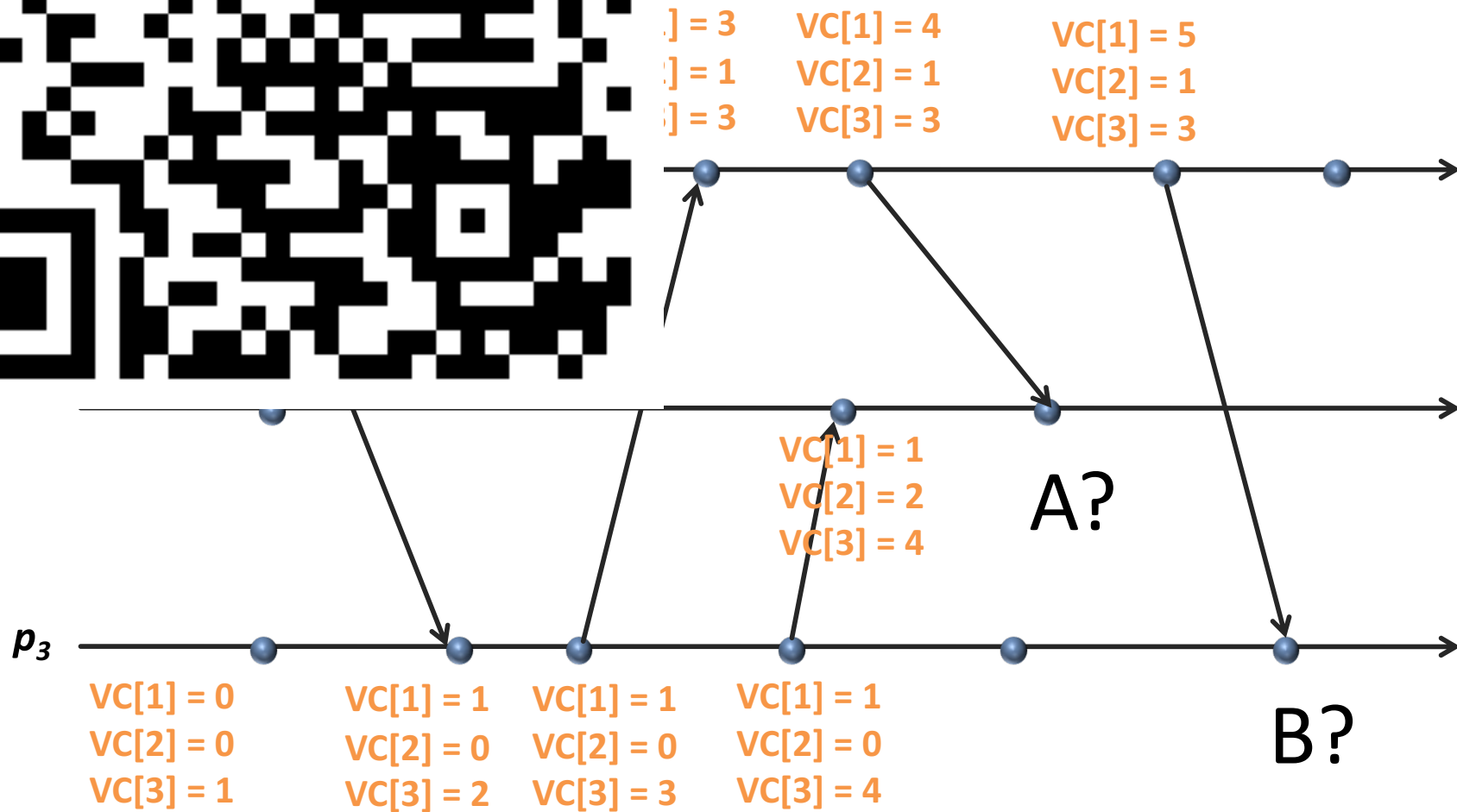
- p_1 thinks incorrectly about other processes if p_1 does not communicate with other processes
- Once p_1 exchanges messages with other processes, it updates its information about other processes using vector clock timestamps received from other processes

Another Vector Clock Example



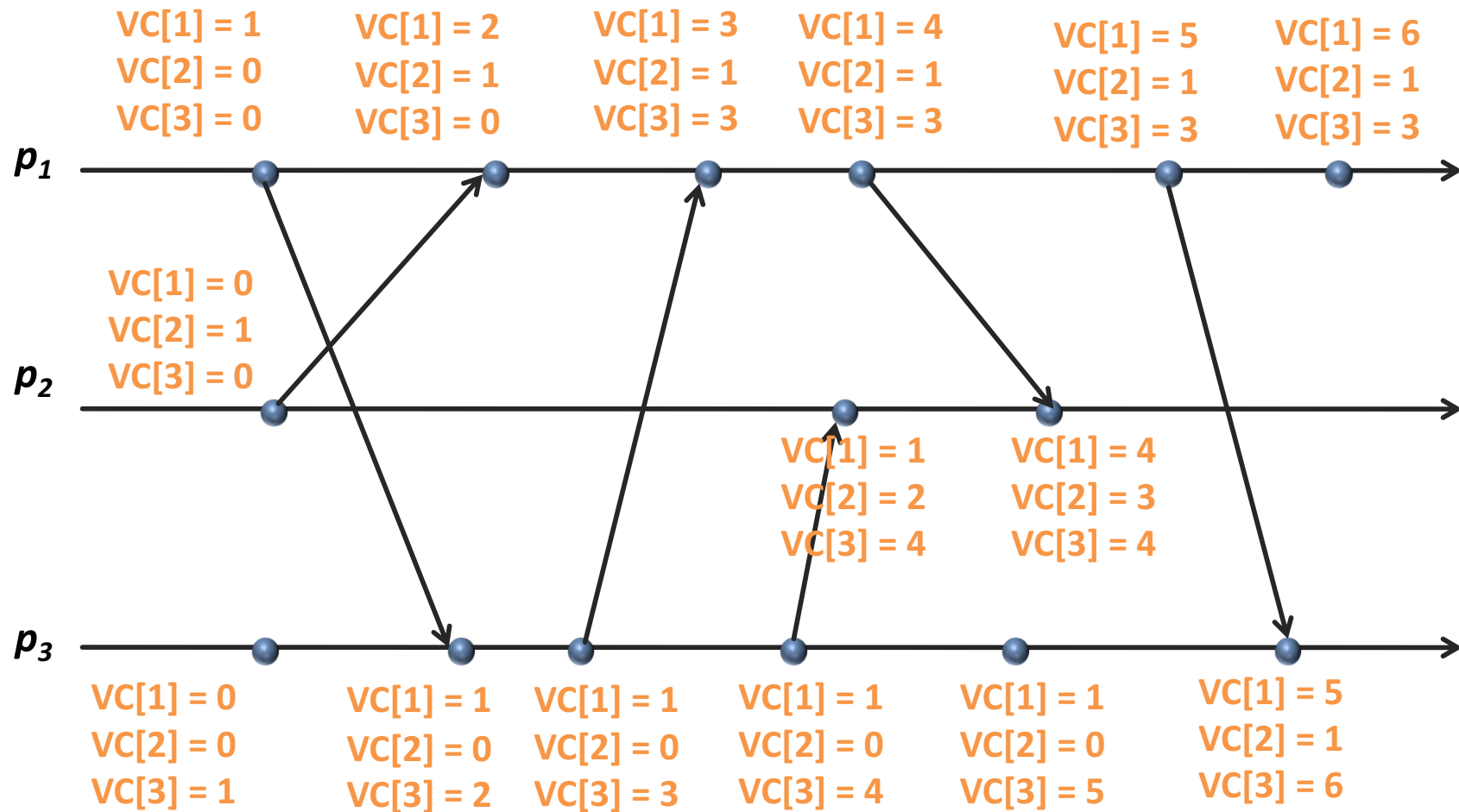


tor Clock Example

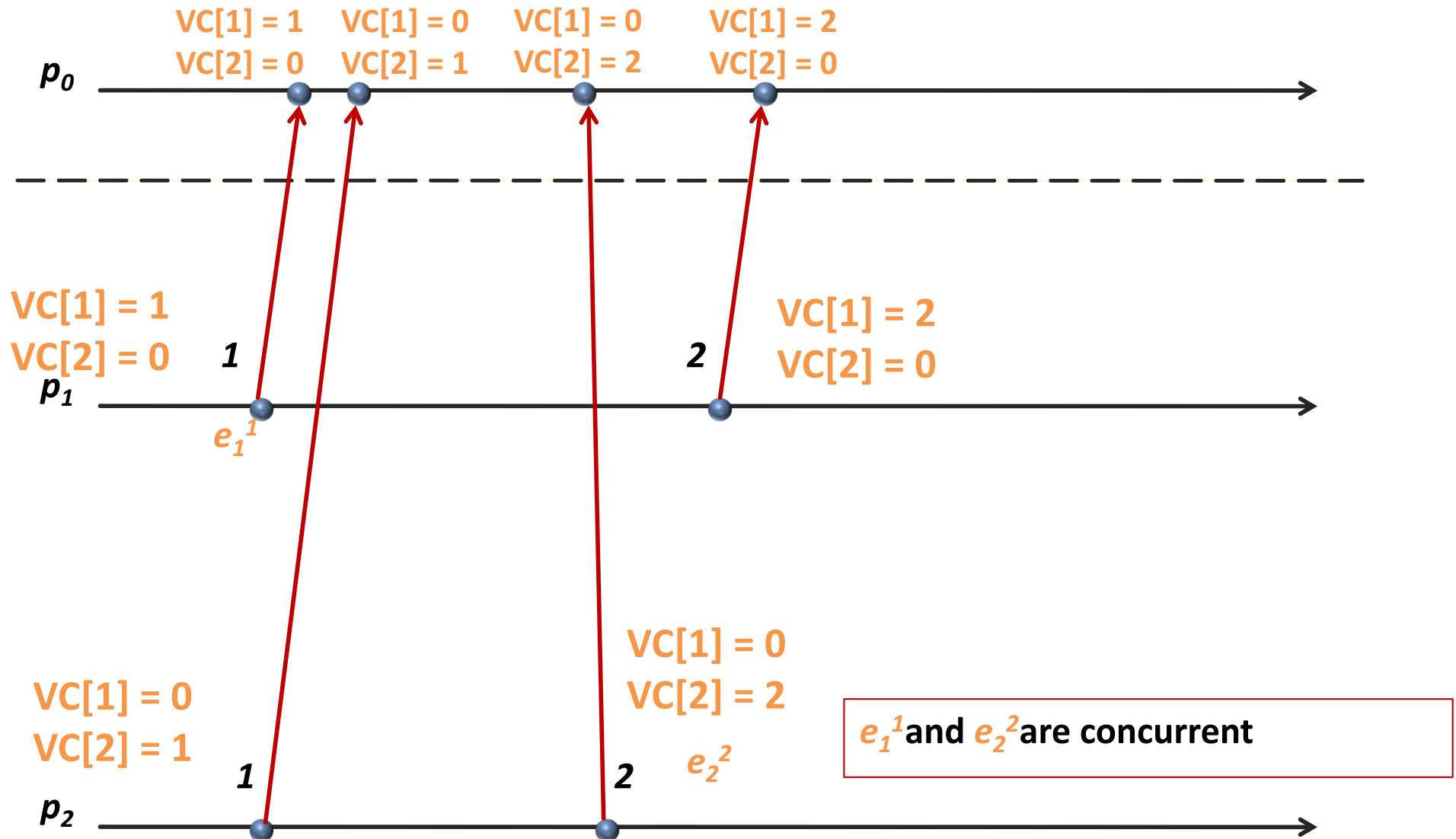




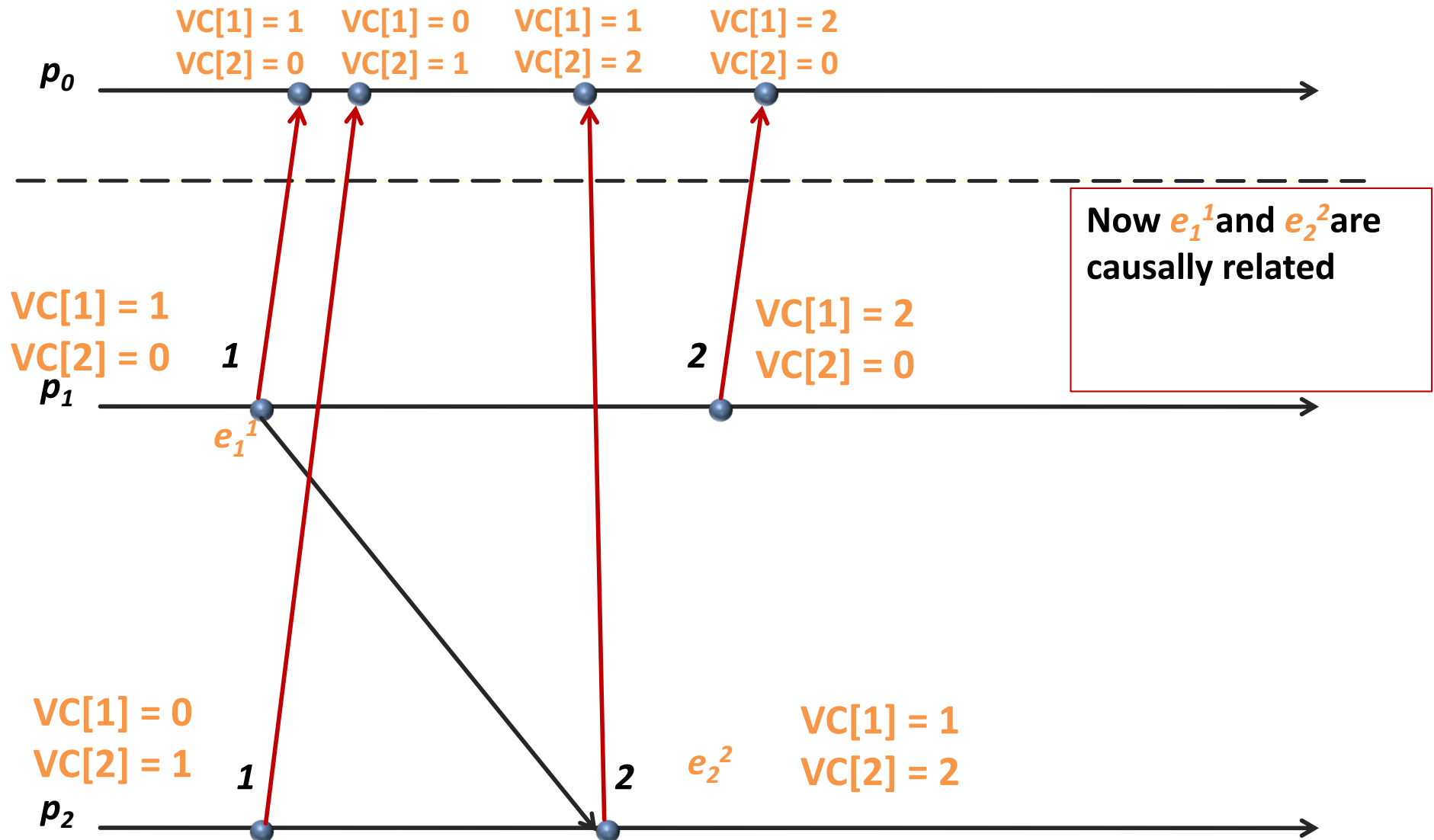
Another Vector Clock Example



Causal Precedence or Concurrency?



Causal Precedence or Concurrency?



Vector Clocks Detect Causal Precedence

- Let us look at timestamps received by observer p_0 :
- **In case when events were concurrent:**
 - From p1: $VC[1] = 2$
 $VC[2] = 0$
 - From p2: $VC[1] = 0$
 $VC[2] = 2$
- **In case when events were causally related:**
 - From p1: $VC[1] = 2$
 $VC[2] = 0$
 - From p2: $VC[1] = 1$
 $VC[2] = 2$

Now the timestamps help detect causal precedence

Use Vector Clocks to Construct Consistent Cuts

- A consistent cut is a cut that does not include *pairwise inconsistent events*
- So the first step is to understand pairwise inconsistency

Informal Definition of Pairwise Inconsistency

- Look at vector timestamps of two events: e_i at p_i and e_j at p_j
- If at event e_i process p_i thinks that process p_j executed more events than process p_j thinks it has executed at event e_j , the events are pairwise inconsistent

Example:

timestamp at p_3

$VC[1] = 5$
 $VC[2] = 1$
 $VC[3] = 6$

p_3 thinks that p_1
has executed 5
events

timestamp at p_1

$VC[1] = 4$
 $VC[2] = 1$
 $VC[3] = 3$

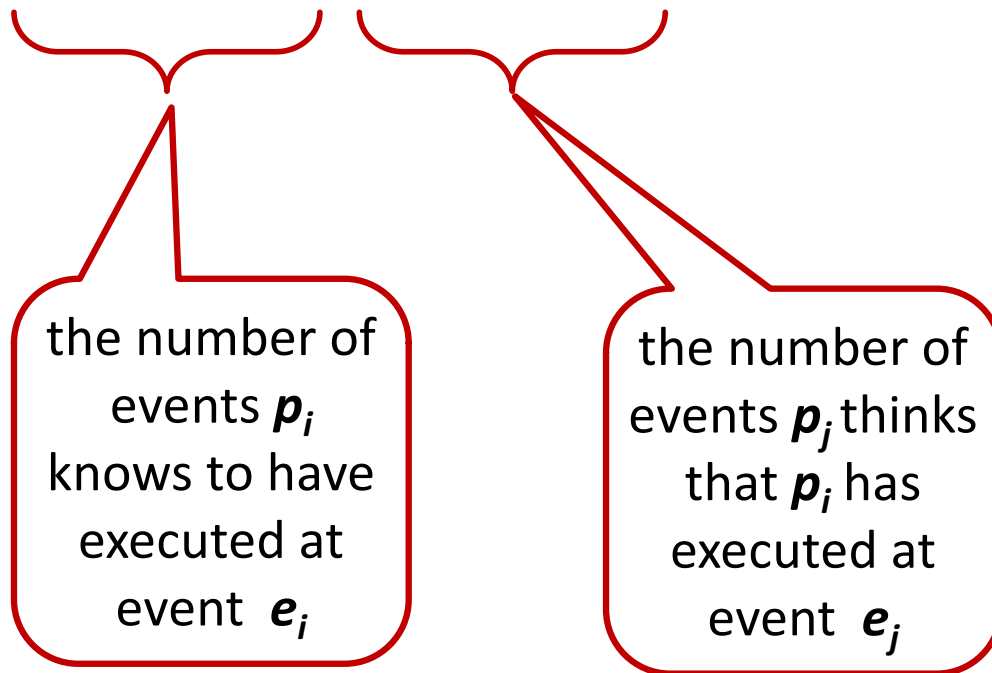
p_1 has
executed 4
events!

- The events are pairwise inconsistent
- Process p_1 must know better about what it's doing than process p_3 could ever know.

Formal Definition of Pairwise Inconsistency

- Two events e_i and e_j , are pairwise inconsistent if:

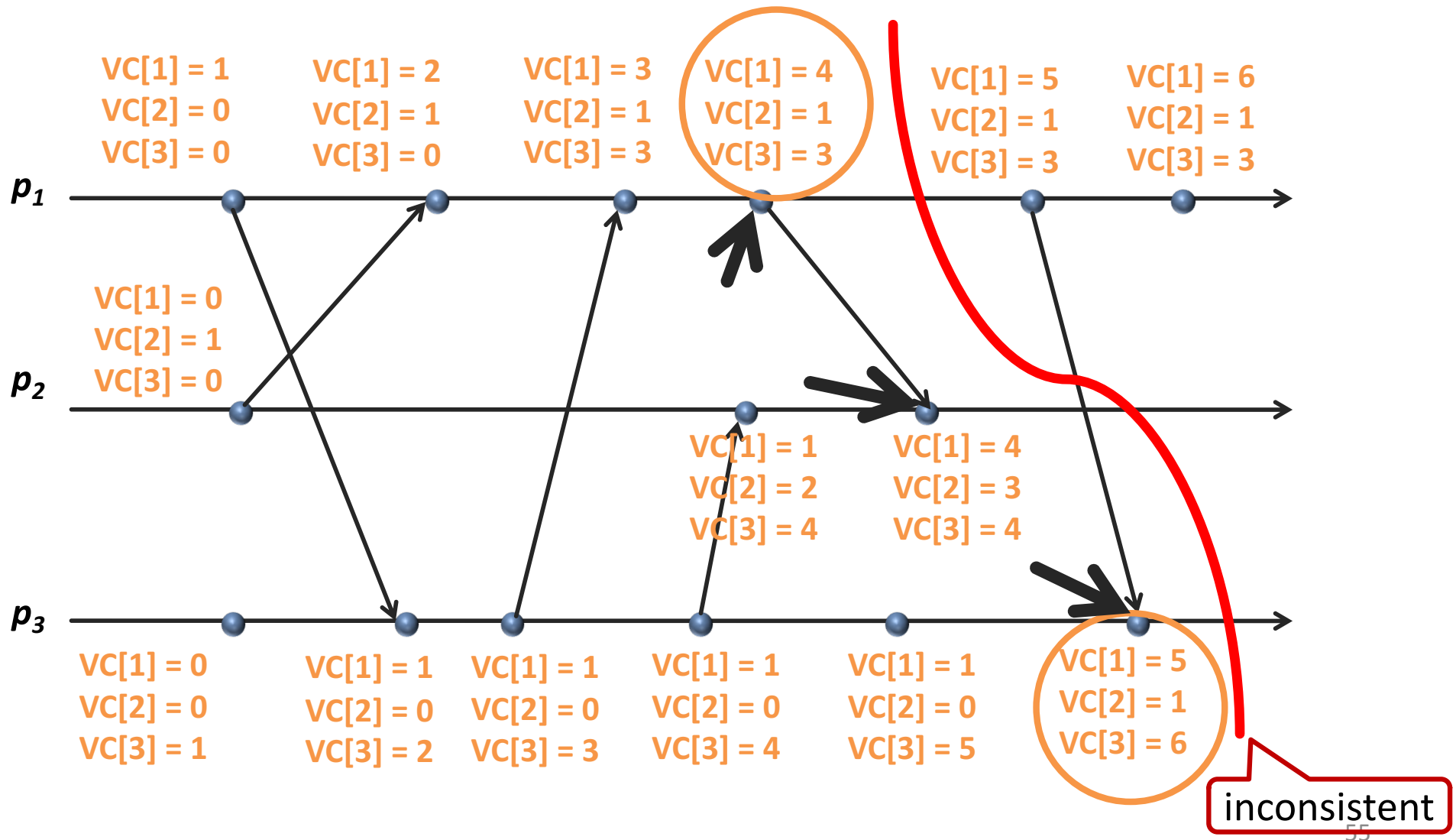
$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$



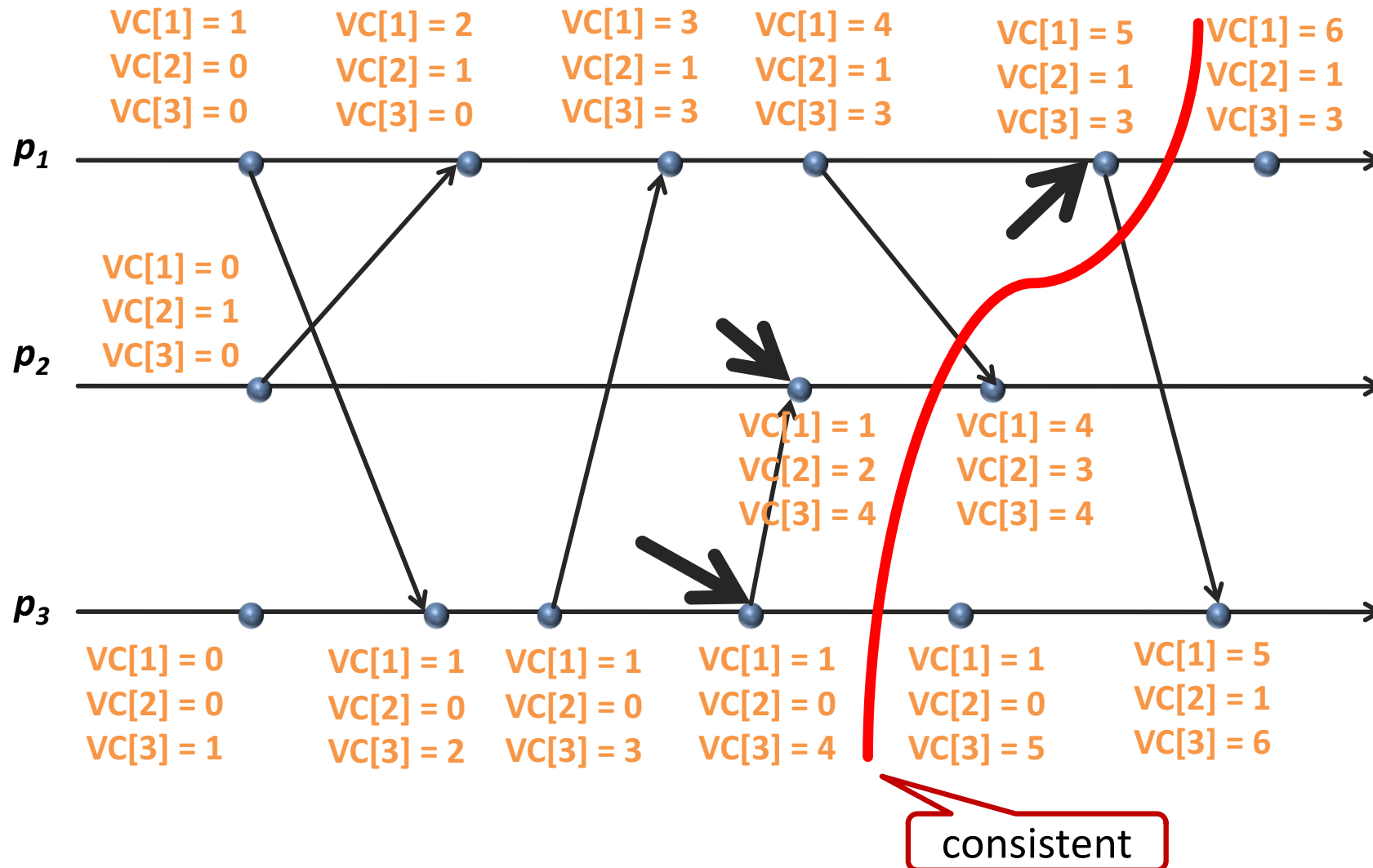
Cut Consistency

- Among all events in the frontier of the cut, check each pair of events for pairwise inconsistency
- If at least two events are pairwise inconsistent, then the cut is inconsistent

Is this Cut Consistent?



And What About This One?



Questions

- Vector Clocks: How many clocks at each node?
 - One per node in the system
- How are they incremented?
 - Local clock: on each event
 - Remote clocks: merged on receive event

Active Monitoring

- Recall: we used vector clocks so we can construct consistent global states (or cuts) from processes' local histories via passive monitoring
- Remember, there is another type of monitoring – **active monitoring**
 - Active monitoring is also referred to as constructing a ***distributed snapshot***

Distributed Snapshot Protocol (Chandy and Lamport)

- Monitor p_0 sends “take snapshot” message to all processes p_i
- If p_i receives such “take snapshot” message for the first time, it:
 - Records its state
 - Stops doing any activity related to the distributed computation
 - Relays the “take snapshot” message on all of its outgoing channels
 - Starts recording a state of its incoming channels – records all messages that have been delivered after the receipt of the first “take snapshot” message

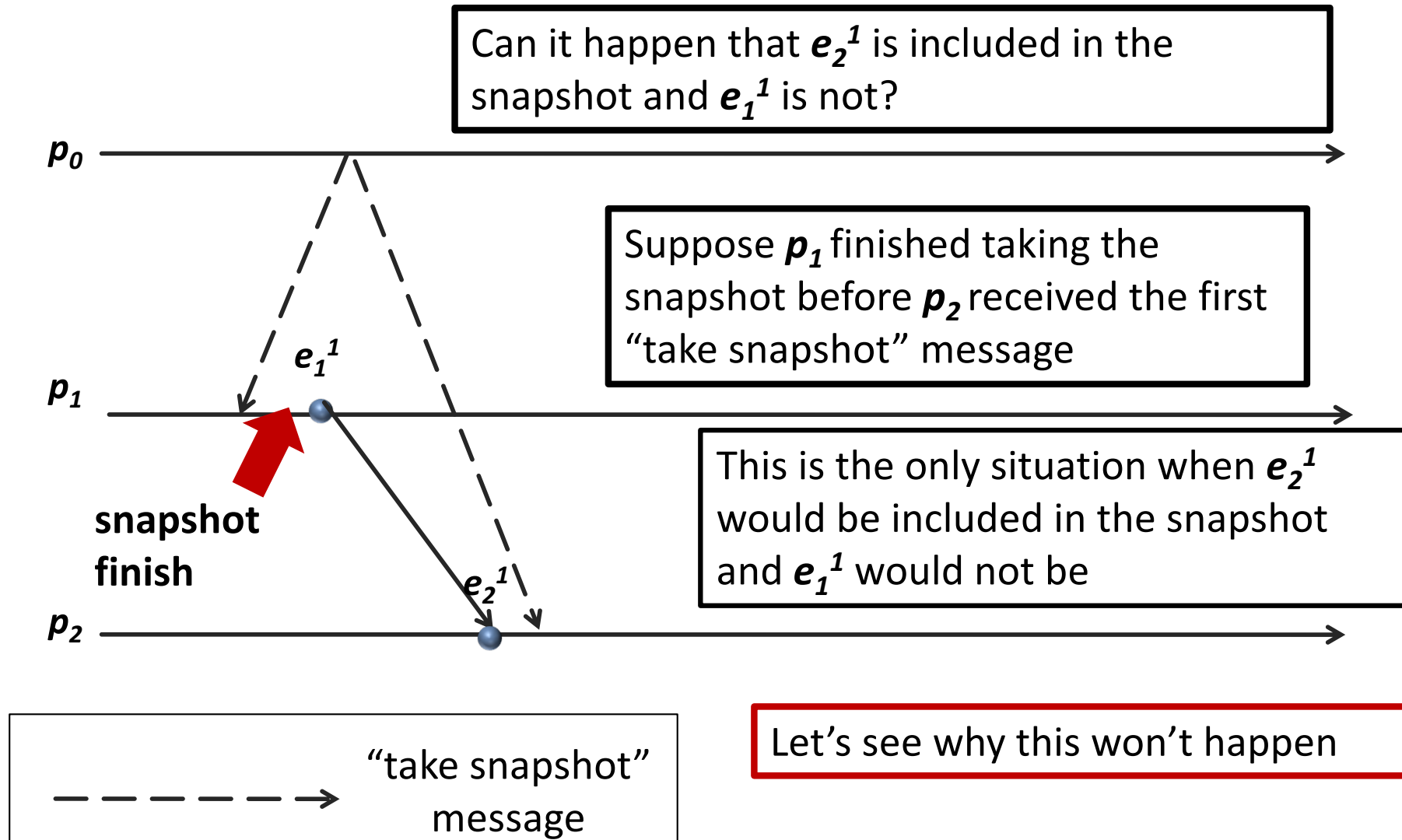
Distributed Snapshot Protocol (Chandy and Lamport)(cont.)

- When p_i receives a “take snapshot” message from process p_j , it stops recording the state of the channel between itself and p_j
- When p_i received “take snapshot” message from all processes and from p_o , it stops recording the snapshot and sends it to p_o

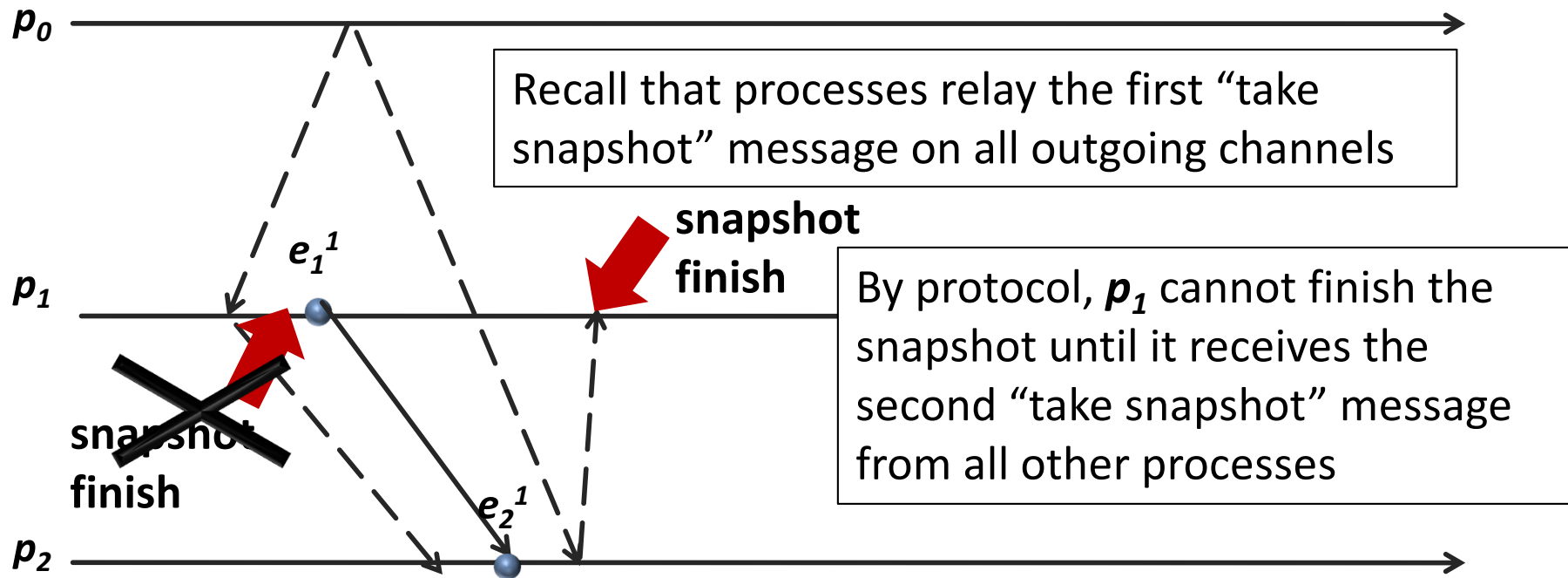
Properties of Chandy-Lamport Protocol

- Let's see why this protocol constructs only consistent snapshots, provided that FIFO channels are used
- Recall the key property of a consistent snapshot (i.e., consistent cut):
 - If event $(e \rightarrow e'')$ and e'' is included in the snapshot, then e must also be included in that snapshot
- Let's see why this property holds if we use the Chandy-Lamport distributed snapshot protocol

Properties of Chandy-Lamport Protocol



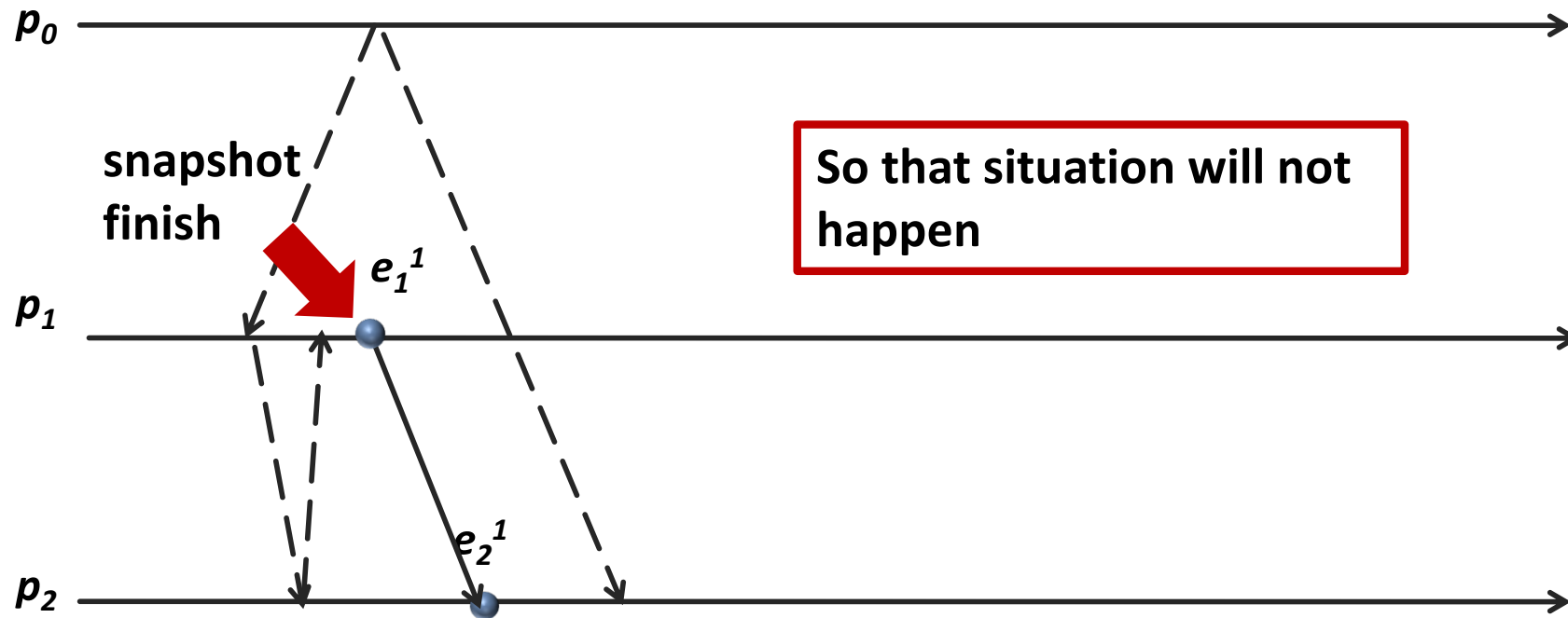
Chandy-Lamport Protocol: Example 1



So p_1 cannot send e_1^1 and finish taking the snapshot before p_2 received the first "take snapshot" message.

So that situation will not happen

Chandy-Lamport Protocol: Example 2



But p_2 must **stop recording events** from p_1 after it receives the "take snapshot" message from p_1 .

Questions

- Active monitoring vs. passive monitoring?
 - Active monitoring: query the system
- Goal of Chandy and Lamport protocol?
 - Global snapshot

Summary

- Constructing a consistent global state is difficult in absence of global clocks
- Consistent global states can be constructed using
 - Active monitoring (distributed snapshots)
 - Passive monitoring (local event histories with vector clock timestamps)

Questions?

In part, inspired from / based on slides from
– Alexandra Fedorova