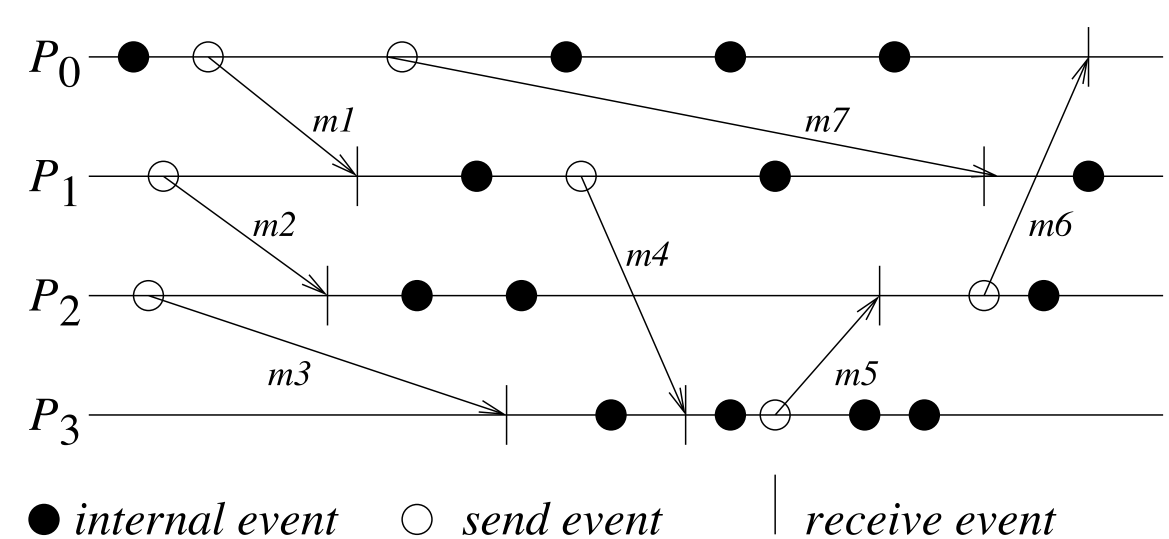
Logical Time

Causality and physical time

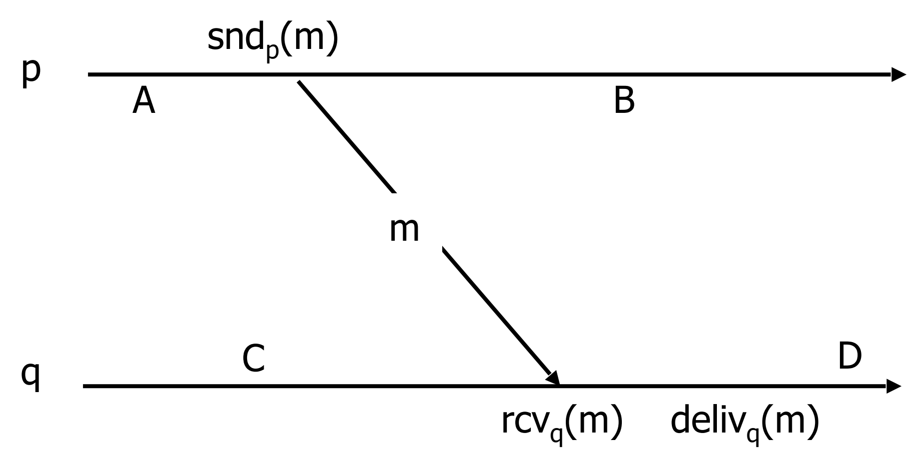
- **Causality** is fundamental to the design and analysis of parallel and distributed computing and OS.
    - Distributed algorithms design
    - Knowledge about the progress
    - Concurrency measure
- Usually causality is tracked using **physical time**.
- In distributed systems, it is **not possible** to have a **global physical time**, only an **approximation**.
    - **Network Time Protocol** (NTP) can maintain time accurate to a few tens of millisecond on the Internet
    - Not adequate to capture the causality relationship in distributed systems

Idea

- We **cannot sync** multiple clocks **perfectly**.
    - Thus, if we want to **order events** happened at different processes, we cannot rely on physical clocks.
- Then came **logical time**.
    - First proposed by Leslie **Lamport** in the 70's
    - Based on **causality** of events
    - Defined **relative time**, not absolute time
- **Critical observation**: time (ordering) only matters if two or more processes interact, i.e., send/receive messages.

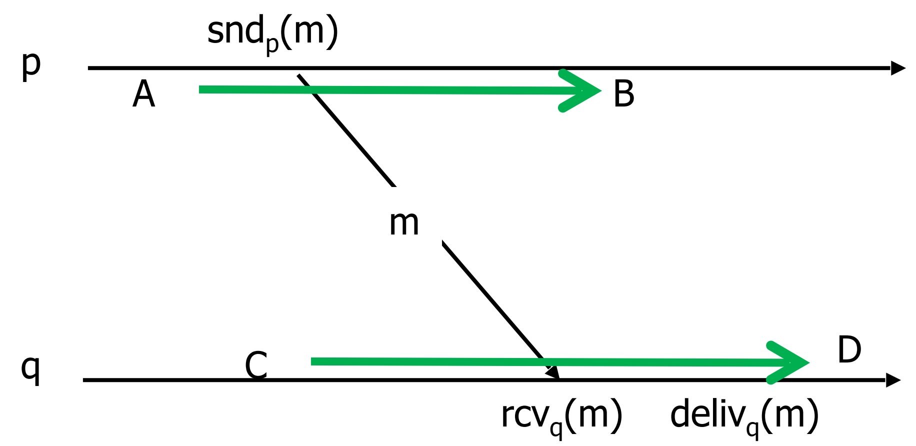


Time-line Diagrams



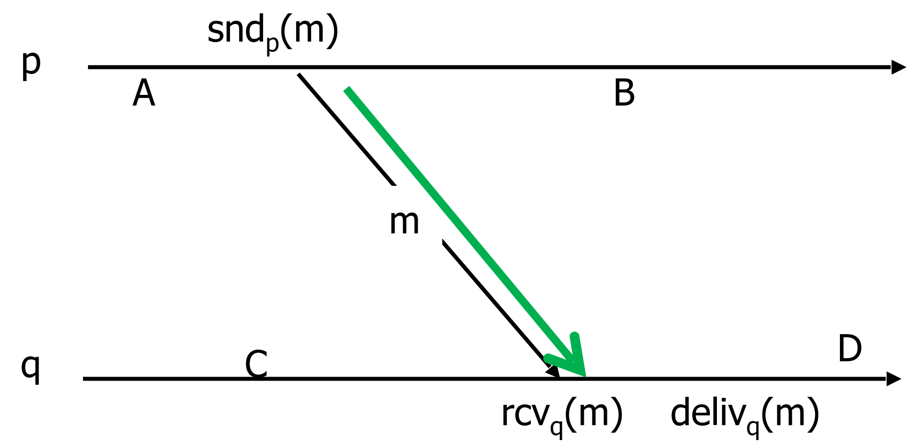
- A, B, C and D are “events”.
  - Could be anything meaningful to the application
  - So are snd(m) and rcv(m) and deliv(m)
  - What ordering claims are meaningful?

Time-line Diagrams



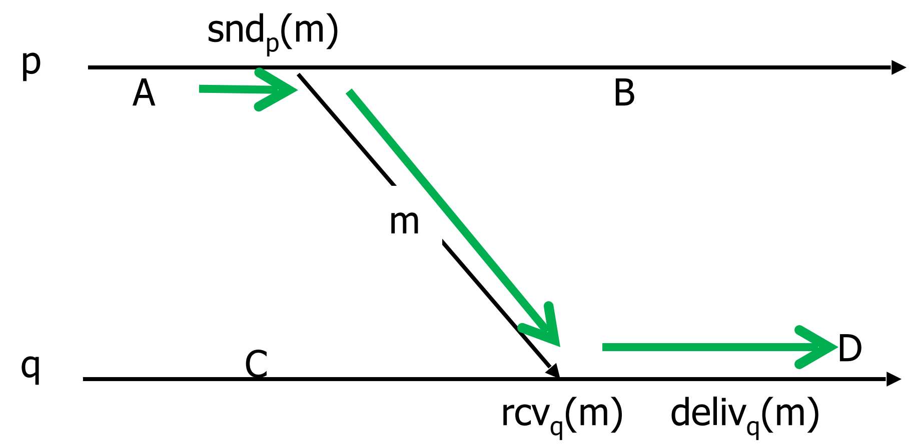
- A happens before B, and C before D
    - “Local ordering” at a single process
    - Write A → B and C → D

Time-line Diagrams



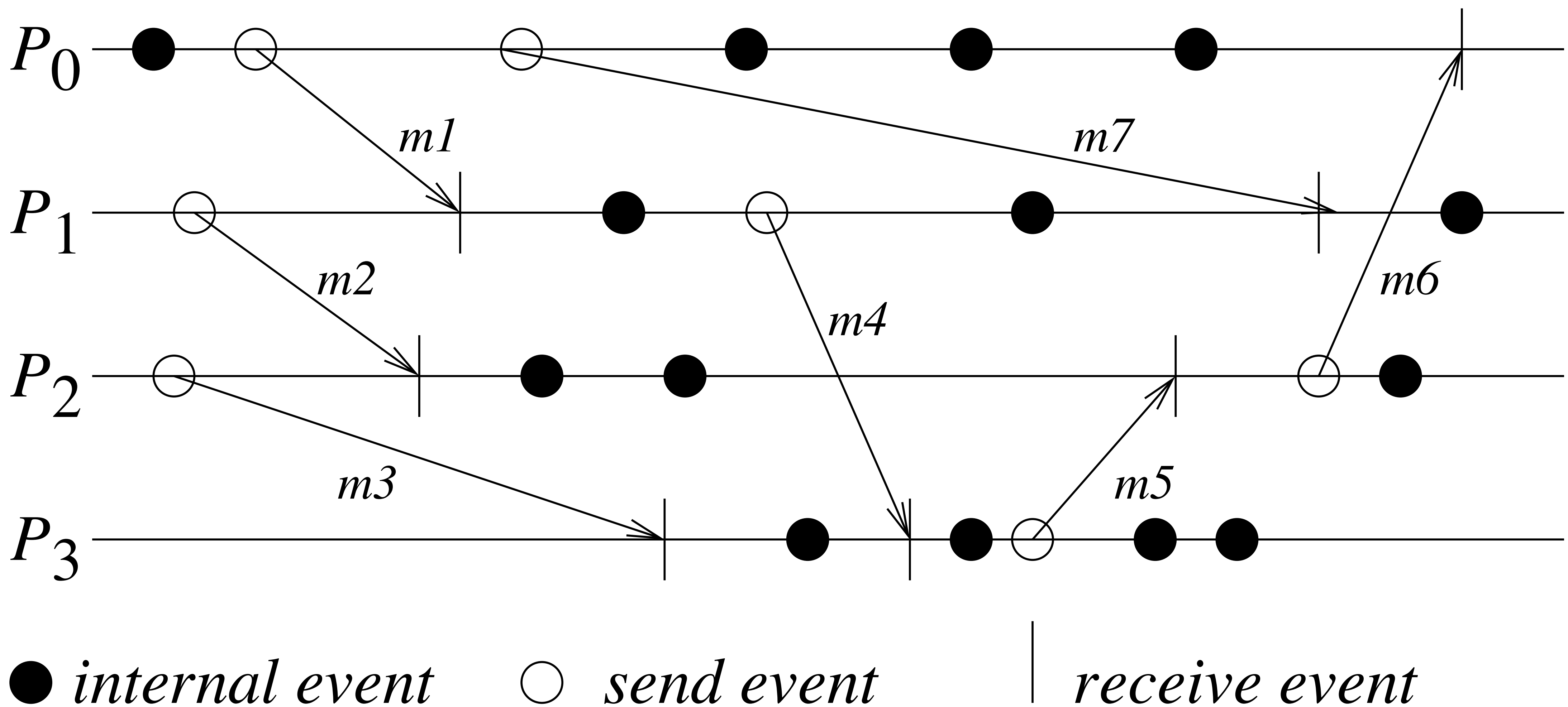
- $snd_p(m)$ also happens before $rcv_q(m)$
  - “Distributed ordering” introduced by a message
  - Write $snd_p(m) \rightarrow rcv_q(m)$

Time-line Diagrams



- A happens before D
  - Transitivity: A happens before $snd_p(m)$, which happens before $rcv_q(m)$, which happens before D
- B and D are concurrent
  - Looks like B happens first, but D has no way to know. No information flowed…

Events


P_0
P_1
P_2
P_3
m1
m2
m3
m4
m5
m6
m7
internal event
send event
receive event

Happens-Before Relation

- The **execution** of a distributed application results in a **set of** distributed **events** produced by the processes.
- Let **H** denote the set of events executed in a distributed computation.
- Define a **binary relation** on the set H, denoted as →, that expresses **causal dependencies** between events in the distributed execution.
- → is called **Happens-Before relation**.
- Properties:
  - On the same process: $a \rightarrow b$ if $realtime(a) < realtime(b)$
  - If p_1 sends m to p_2: $send(m) \rightarrow receive(m)$
  - Transitivity: if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

System of Logical Clocks

- Informally:
  - Every process has a **logical clock** that is advanced according to some rules.
  - Every event is **assigned** a logical timestamp.
  - The $\rightarrow$ relation between two events can be **inferred** from their timestamps.
  - Timestamps obey a **monotonicity property**: if a $\rightarrow$ b, then timestamp(a) < timestamp(b).
- Formally, a **system of logical clocks** is composed by:
  - a **time domain** T, whose elements form a partially ordered set over a relation <.
  - a **logical clock** C, that is a function mapping an event e in H to an element in the time domain T, denoted as C(e) and called **timestamp** of e.
  - a logical clock C must satisfy the **clock consistency condition**:

$$\text{for two events } e_i \text{ and } e_j,\ \ e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$

- The system of clocks (T,C) is said to be **strongly consistent** if the following condition is satisfied:

$$\text{for two events } e_i \text{ and } e_j,\ \ e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$$

Implementation

- Implementation of logical clocks require:
  - **data structures** local to every process to represent logical time
  - a **set of rules** to update the data structures to ensure the consistency condition
- The **data structures** of a process p_i must allow it to:
  - measure its own progress, with a (**logical**) **local clock** lc_i
  - represent its own view of the logical global time to assign consistent timestamps to its local events, with a (**logical**) **global clock** gc_i
  - typically lc_i is a part of gc_i
- The rules must:
  - R1: decide how the logical local clock is updated by a process when it executes an event (send, receive, internal)
  - R2: decide how a process updates its logical global clock to update its view of the global time and global progress.

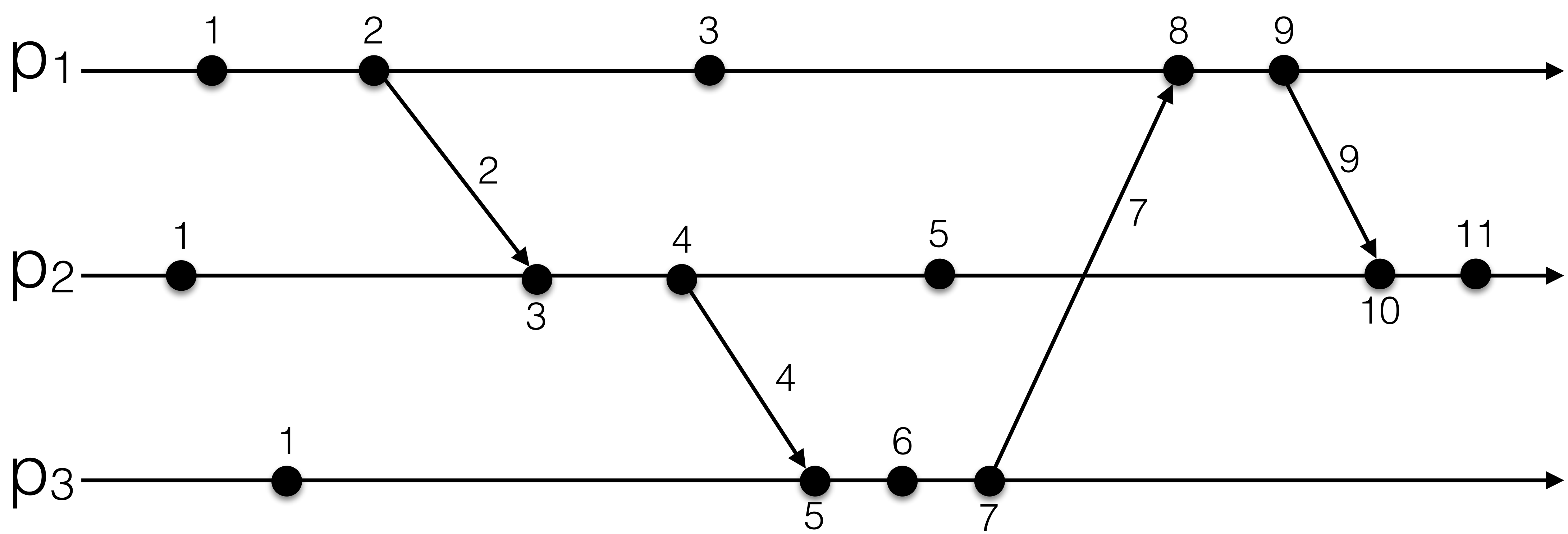
Scalar Clocks

- Proposed by Lamport in 1978.
- Time domain T is the set of **non-negative integers**.
- For each process p_i, the logical local clock and the logical global clock are squashed into **one integer variable** C_i.
- R1: before executing an event (send, receive, internal), process p_i executes the following:

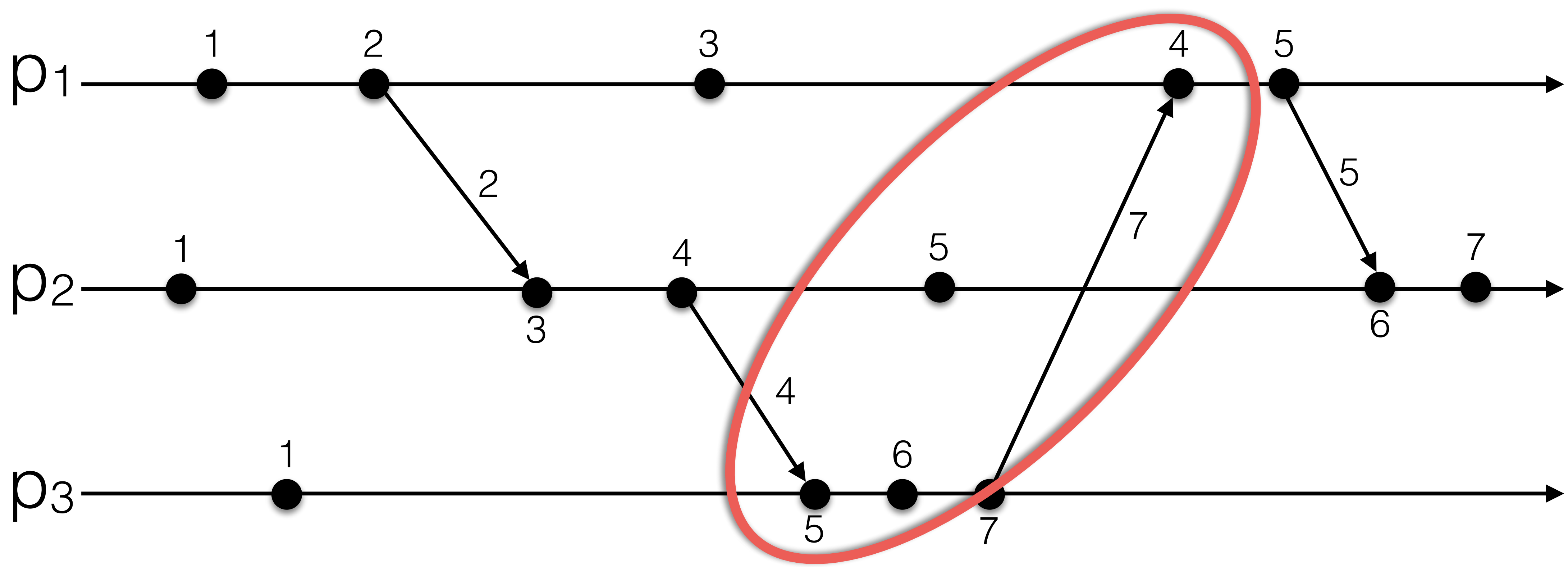
$$C_i = C_i + d \ (d > 0)$$

  - In general every time R1 is executed, d can have a different value.
  - Typically d is kept at 1 to keep the rate of increase of C_i's to its lowest values.
- R2: Each message piggybacks the clock value of it sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:
  1. $C_i = \max(C_i, C_{msg})$
  2. Execute R1
  3. Deliver the message to p_i

Example



Find the error…

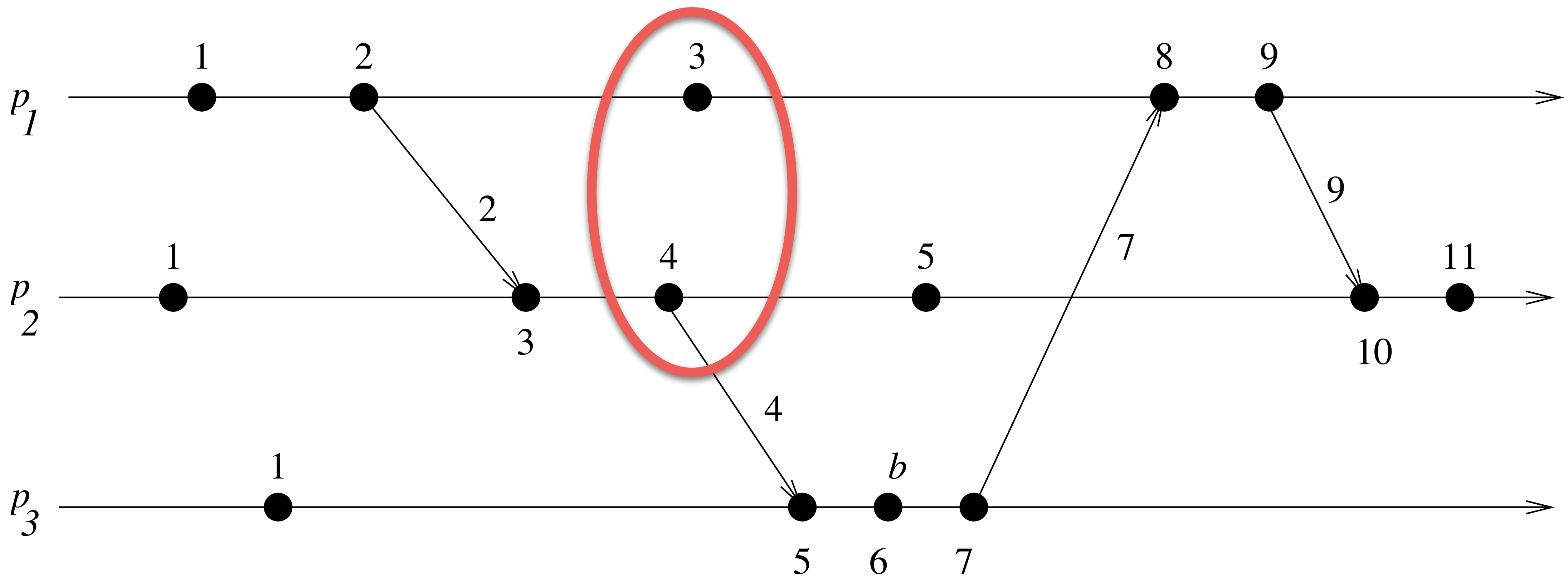


Basic Properties

- The **consistency** property is **satisfied**.
- If $C(e_i) = C(e_j)$ then e_i and e_j are concurrent events.
- To **totally order** events, we need a **tie-breaking mechanism** for concurrent events. This is typically done by augmenting the scalar timestamp with a **process identifier**, e.g., (t,i).
    - Process identifiers are linearly ordered and used to break ties.
- If d=1 we have that, if event e has a timestamp h, then h-1 represents the **minimum logical duration**, counted in units of events, required before producing event e.
- The **strong consistency** property is **NOT satisfied**.

Example

3 < 4 but the former did not happen before the latter



The lack of strong consistency is due to the squashing of logical local and global clocks into one

Vector Clocks (I)

- Proposed by **Fidge**, **Mattern** and **Schmuck** in 1988-1991.
- Time domain T is a set of n-dimension **non-negative integer vectors**.
- Each process p_i maintains a vector $vt_i[1..n]$.
- $vt_i[i]$ is the **logical local clock** of p_i.
- $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time. If $vt_i[j] = x$ then process p_i knows that local time at process p_j had progressed till x.

Vector Clocks (II)

- Initially $vt_i = [0, 0, 0, \ldots, 0]$
- R1: before executing an event (send, receive, internal), process p_i executes the following:

$$vt_i[i] = vt_i[i] + d \ (d > 0)$$

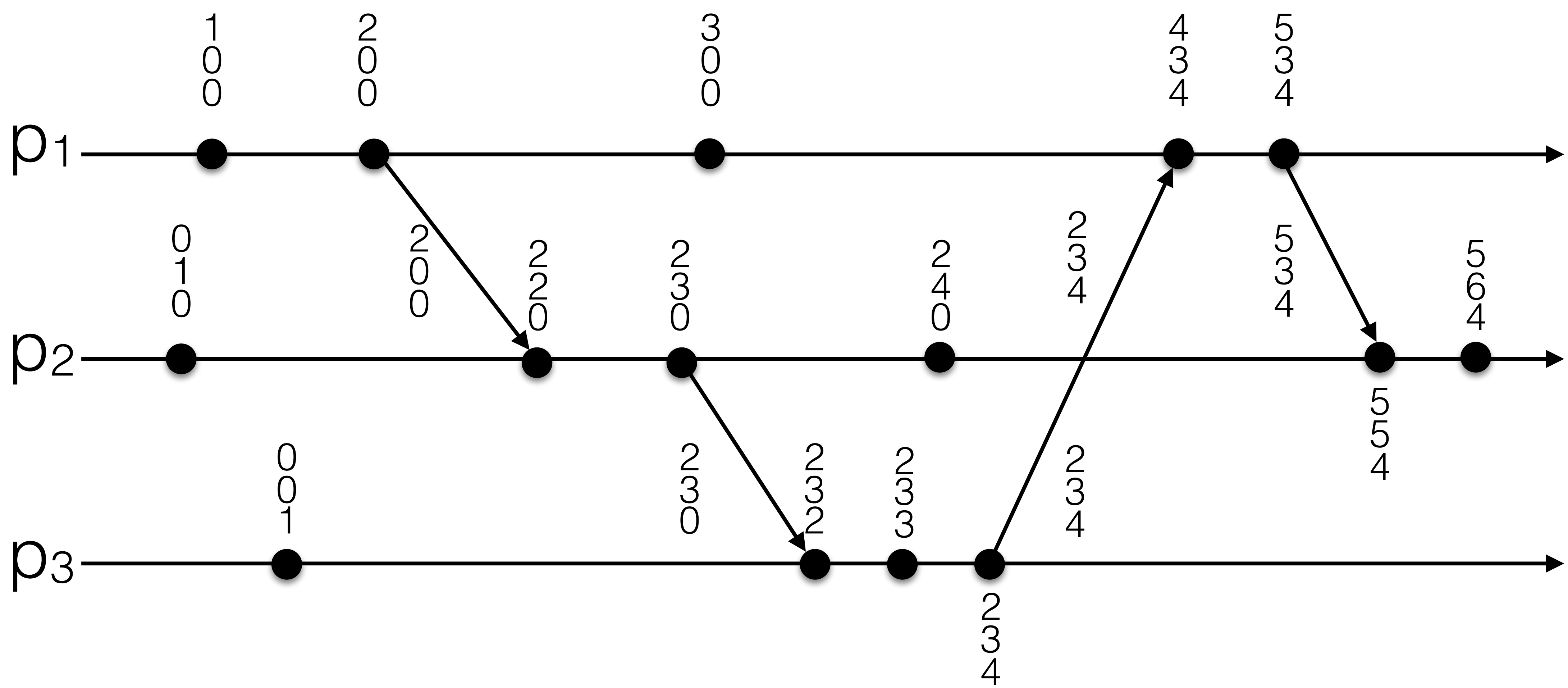
- R2: Each message m is piggybacked with the vector clock vt of the sender process at sending time. When a process p_i receives a message with (m,vt), it executes the following actions:

  1. Update its logical global time as follows:

  $$1 \leq k < n: vt_i[k] = \max(vt_i[k], vt[k])$$

  2. Execute R1
  3. Deliver the message m to p_i

Example



p1
p2
p3
1 0 0
2 0 0
3 0 0
4 3 4
5 3 4
0 1 0
2 0 0
2 2 0
2 3 0
2 4 0
2 3 4
5 3 4
5 6 4
5 5 4
0 0 1
2 3 0
2 3 2
2 3 3
2 3 4
2 3 4

Comparing Vector Clocks

- $VT_1 = VT_2$
  - iff $VT_1[i] = VT_2[i]$, for all $i = 1, \ldots, n$
- $VT_1 \leq VT_2$,
  - iff $VT_1[i] \leq VT_2[i]$, for all $i = 1, \ldots, n$
- $VT_1 < VT_2$,
  - iff $VT_1 \leq VT_2$ & $\exists$ j ($1 \leq j \leq n$ & $VT_1[j] < VT_2[j]$)
- $VT_1 \parallel VT_2$
  - iff $\neg(VT_1 \leq VT_2)$ & $\neg(VT_2 \leq VT_1)$

Basic Properties

- The **consistency** property is **satisfied**.
- The **strong consistency** property is **satisfied** (using always at least n elements).
- If two events x and y have timestamps vh and vk respectively, then we have the following **isomorphism**:

$$x \rightarrow y \Leftrightarrow vh < vk$$

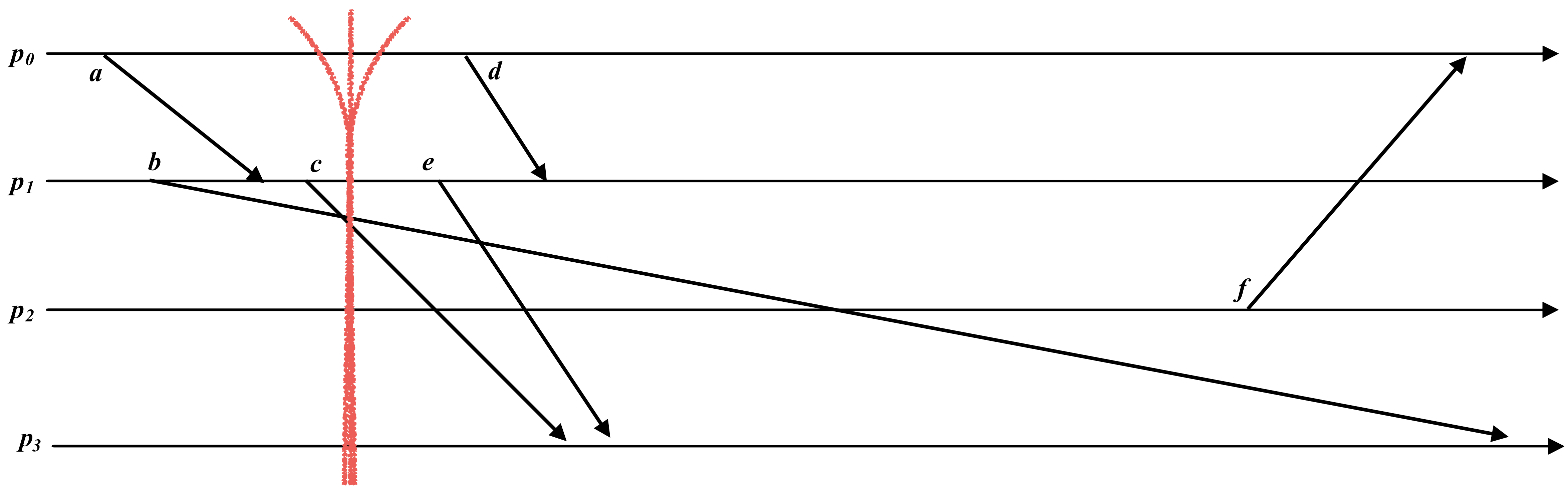
$$x \parallel y \Leftrightarrow vh \parallel vk$$

- If d = 1 then we have the **event counting** property of scalar clocks for logical local clocks.
- Since vector clocks are strongly consistent they can **track causal dependencies exactly**.

Temporal Distortions

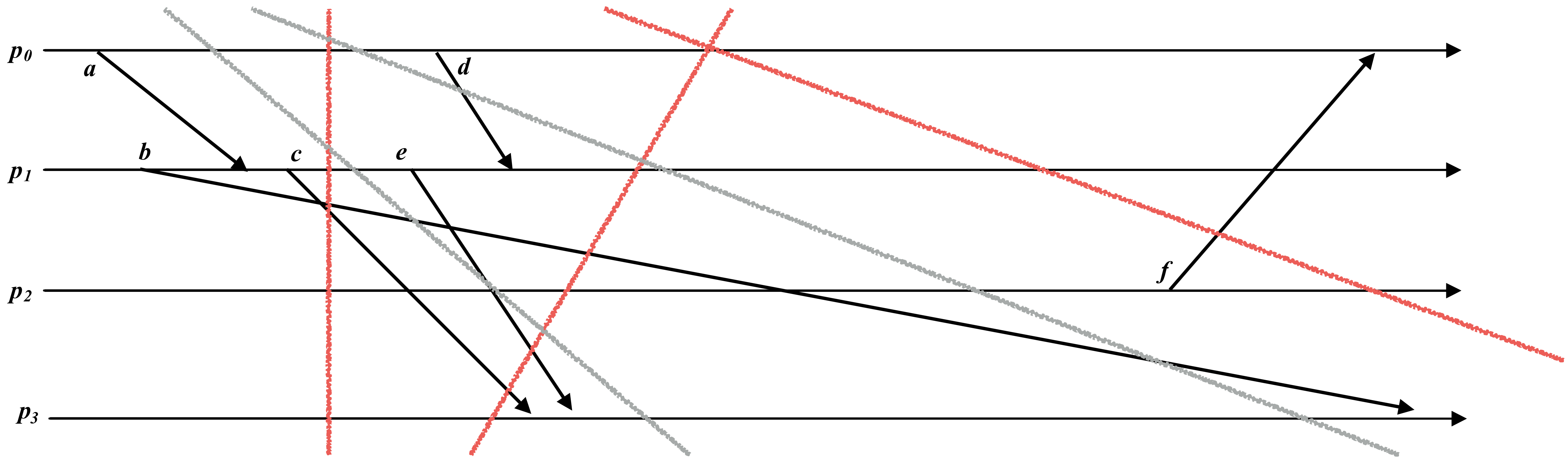
- Things can be complicated because we can’t predict
  - Message delays (they vary constantly)
  - Execution speeds (often a process shares a machine with many other tasks)
  - Timing of external events

What does "now" mean?



- Timelines can “stretch”…
  - … caused by scheduling effects, message delays, message loss…
- Timelines can “shrink”
  - E.g. something lets a machine speed up

Cuts

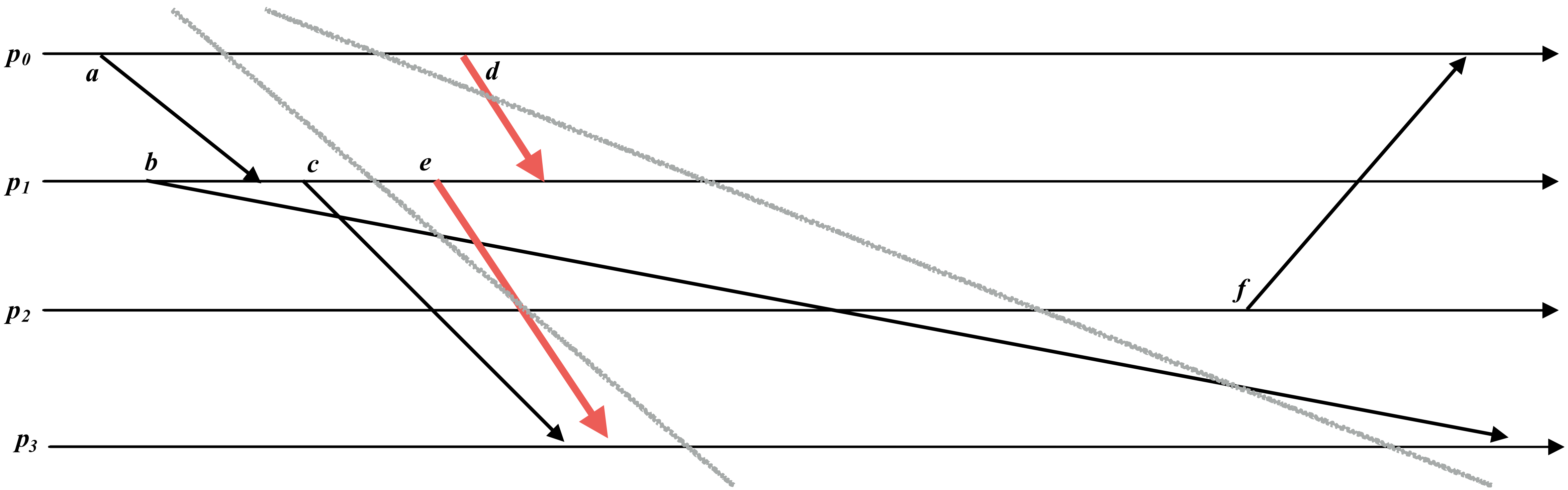


- Cuts represent instants of time.
  - But not every “cut” makes sense
    - Red cuts could occur but not gray ones.

Consistent Cuts

- Idea is to identify system states that “might” have occurred in real-life
  - Need to avoid capturing states in which a message is received but nobody is shown as having sent it
  - This is the problem with the gray cuts

Inconsistent Cuts



- Red messages cross grey cuts "backwards" in time

Consistent Cuts and Snapshots

- We want to draw a line across the system state such that
  - Every message “received” by a process is shown as having been sent by some other process
  - Some pending messages might still be in communication channels
- And we want to do this while running

System Assumptions

- Communication must be reliable
- Processes be failure-free.
- Point-to-point message delivery must be ordered (FIFO channels)
- Process/channel graph must be strongly connected
- No global shared memory and physical clock

- Under these assumptions, and after the algorithm completes, each process hold a copy of its local state and a set of messages that was in transit, with that process as their destination, during the snapshot.

System Assumptions

- At any instant, the state of process p_i , denoted by LS_i , is a result of the sequence of all the events executed by p_i till that instant.
- For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i.
- For an event e and a process state LS_i, $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process p_i to state LS_i.
- For a channel C_{ij} , the following set of messages can be defined based on the local states of the processes p_i and p_j

$$transit(LS_i, LS_j) = \{ m_{ij} \mid send(m_{ij}) \in LS_i \text{ and } recv(m_{ij}) \notin LS_j \}$$

Consistent Global State

- The **global state** of a distributed system is a collection of the local states of the processes (LS_i) and the channels (SC_{ij}).
- 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**: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij}$ XOR $recv(m_{ij}) \in LS_j$
  - **C2**: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij}$ AND $recv(m_{ij}) \notin LS_j$

Issues

- 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).

- 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.

Chandy-Lamport Algorithm

- The 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.

Chandy-Lamport Algorithm

- 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.
- 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.

Rules

- **Marker Sending Rule** for process p_i
  1. Process p_i *records* its state.
  2. For each outgoing channel C on which a marker has not been sent, p_i *sends* a marker along C before p_i sends further messages along C.
- **Marker Receiving Rule** for process p_j
  1. On *receiving* a marker along channel C:

     if p_j has not *recorded* its state then

     1. *Record* the state of C as the *empty set*
     2. Follow the **Marker Sending Rule**

     else

     1. *Record* the state of C as the *set of messages* received along C after p_j's state was recorded and before p_j received the marker along C

Correctness & 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 a 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 and O(d) time, where e is the number of edges in the network and d is the diameter of the network.