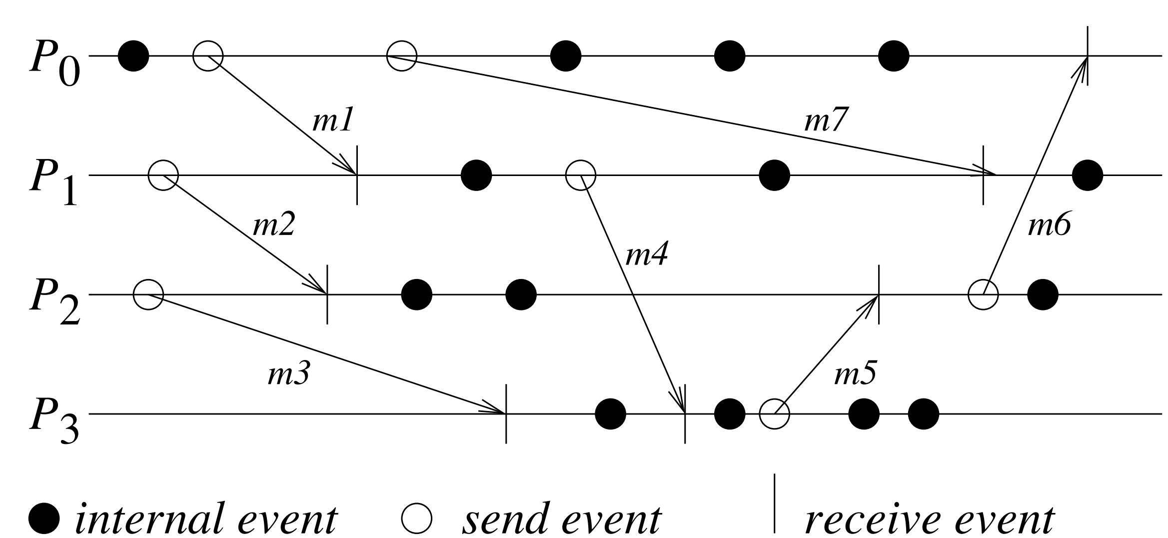
Logical Time

Causality and physical time

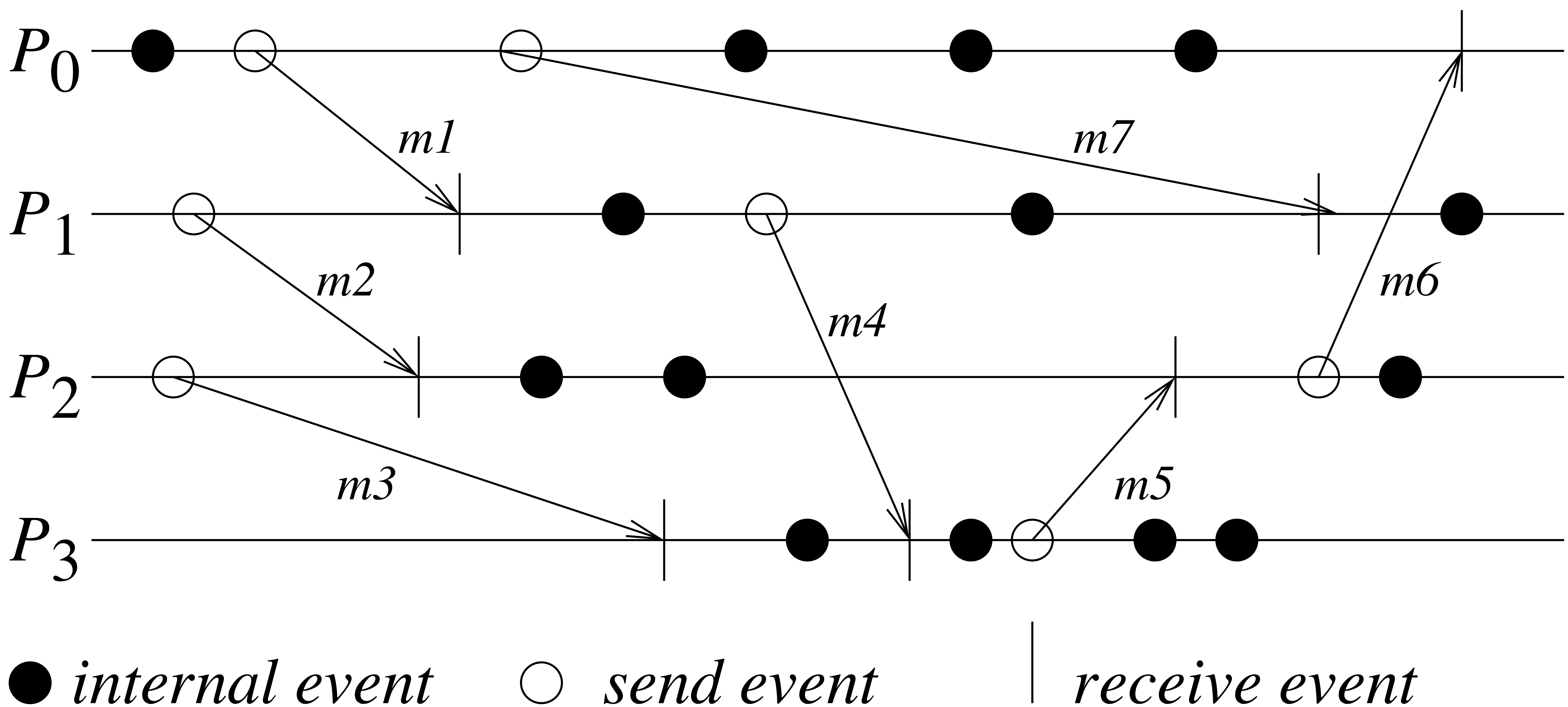
- **Causality** is fundamental to the design and analysis of parallel and distributed computing and OS.
    - Distributed algorithms design
    - Knowledge about the progress
    - Concurrency measure
- Usually causality is tracked using **physical time**.
- In distributed systems, it is **not possible** 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.



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 $\rightarrow$, that expresses **causal dependencies** between events in the distributed execution.
- $\rightarrow$ 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 → relation between two events can be **inferred** from their timestamps.
  - Timestamps obey a **monotonicity property**: if a → 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.

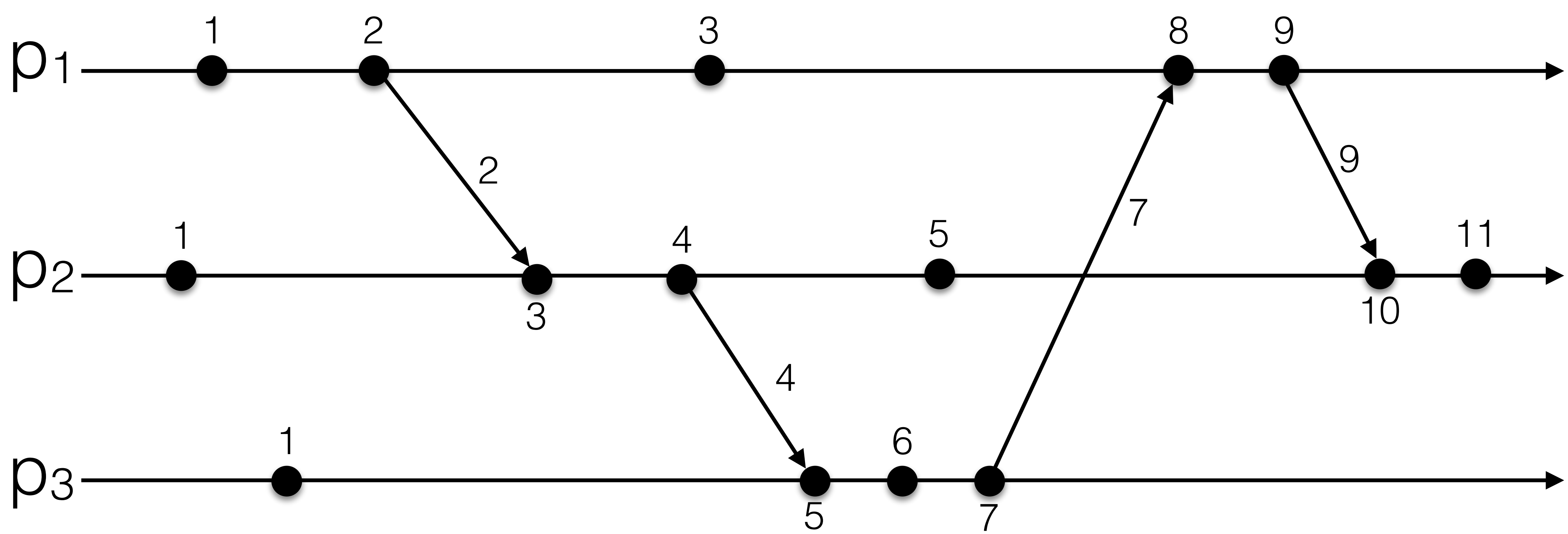
Scalar Clocks

- 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:

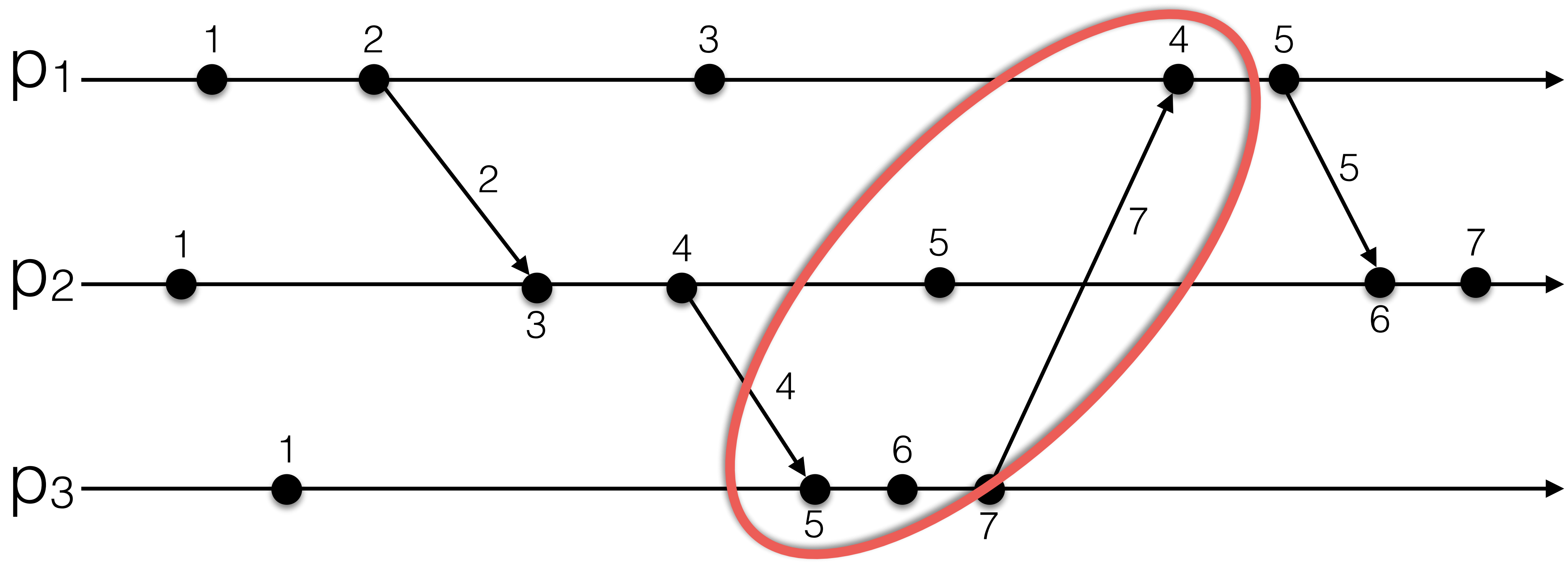
$$C_i = C_i + d \ (d > 0)$$

  - 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…

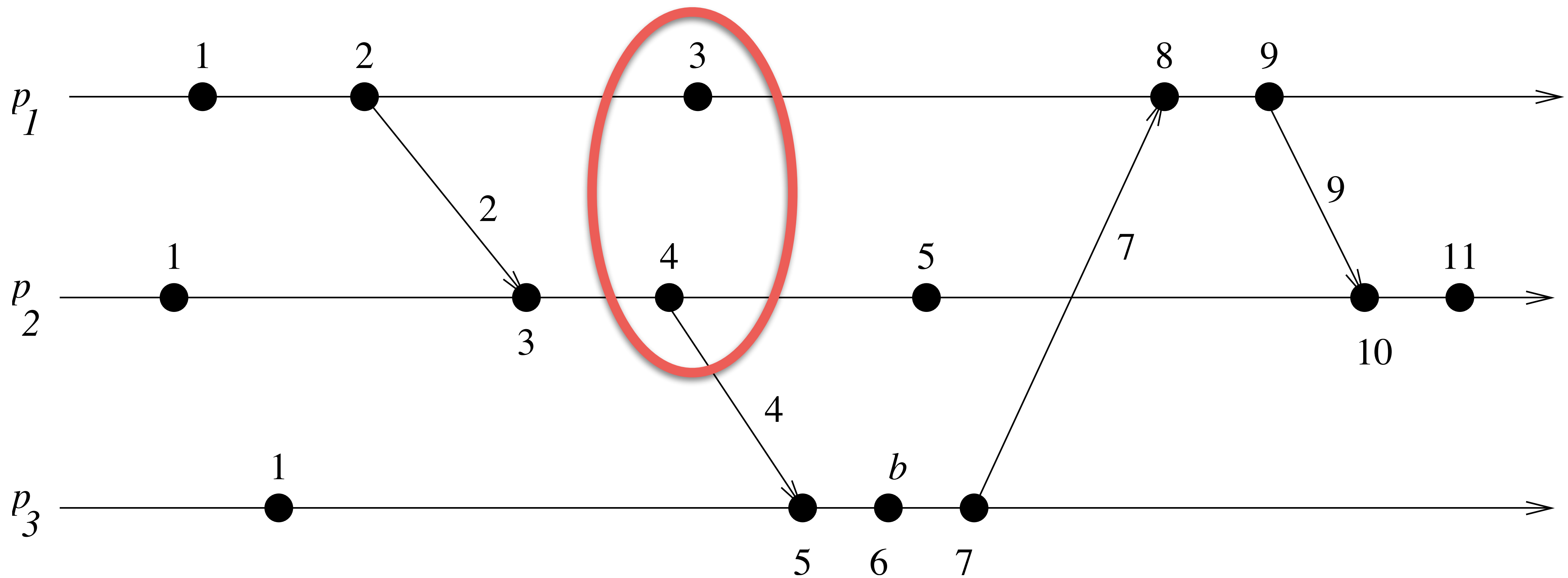


Basic Properties

- 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:

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

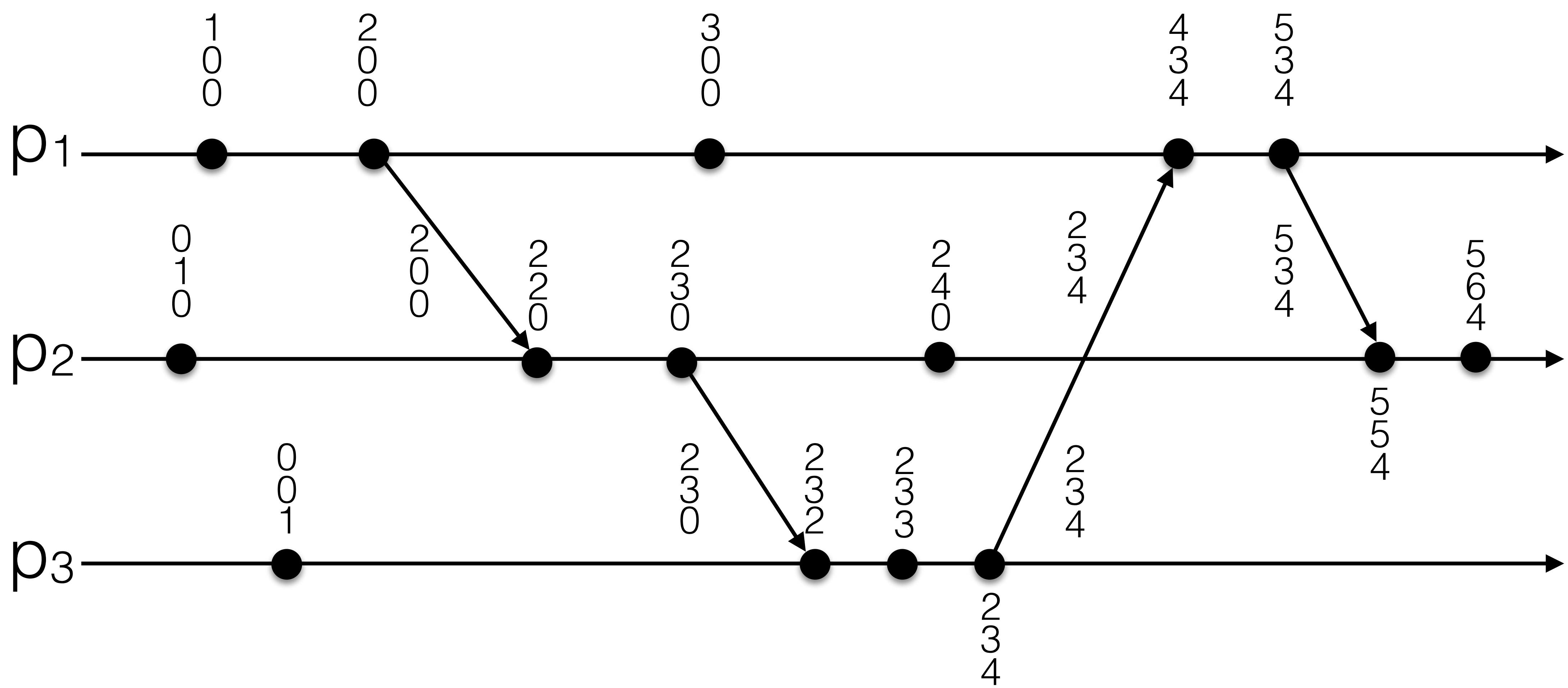
- R2: Each message m is piggybacked with the vector clock vt of the sender process at sending time. When a process p_i receives a message with (m,vt), it executes the following actions:

  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**.