Distributed Systems

Monsoon 2024

Lecture 3

International Institute of Information Technology

Hyderabad, India

Time in Distributed Systems

- We have seen that NTP, the popular Internet time protocol, is not a good fit for distributed systems.
  - Especially at scale.
- There are ways to use similar protocols with heavy engineering effort.
  - Highly expensive and sophisticated.
  - Not for every system
- We will now see a simpler approach that works for most cases of distributed systems.

Moving Over to Distributed Systems

- Fortunately, however, asynchronous distributed computations make progress in spurts.
- Causality: Dictionary meaning is to involve a cause, or marked by cause and effect
  - Fundamental to the design and analysis of parallel and distributed computing and operating systems.
  - Allows reasoning, analyzing, and drawing inferences about a computation.
- Causal precedence relation among the events of processes helps in
  - distributed algorithms design,
  - tracking of dependent events,
  - knowledge about the progress of a computation, and
  - concurrency measures

Moving over to Distributed Systems

- Therefore, a logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.
- To define what logical time means, we have to first model distributed computation in terms of events.
  - Include communication channels.
- A bit of rough stretch with lots of notations and definitions.
  - Most definitions are intuitive but require the formal rigour.

A Distributed Program in Execution

- Goals
  - To understand the timeline of events across processors, and
  - To understand when two events are logically concurrent vs. when the two events are physically concurrent
  - To model message delivery of the communication channels.
  - To model the state of a distributed system

Modeling Distributed Systems

- Can think of a distributed system as a collection of processors, and
- A communication network.
- The processors
  - May fail
  - Share no global memory
- We also assume that the network
  - Has delays that are finite but unpredictable.
  - May not respect message ordering
  - Can lose messages, garble messages, duplicate messages
  - May have links that become unavailable/may fail.

A Distributed Program

- We will write such programs in the coming weeks too.
- To model, we say that a distributed program is
  - A set of n processes, P_1, P_2, …, P_n, typically each running on a different processor.
  - A set of channels C_{ij} such that C_{ij} connects Processor i to Processor j.
- The state of a process P_i includes
  - The local memory of P_i
  - Also depends on messages sent, the context.
- The state of a channel C_{ij} includes
  - The messages in transit on this channel.

A Distributed Program in Execution

- We will assume that local actions are spontaneous.
- Sending a message is non-blocking.
  - Processor sending a message does not wait for its delivery.
- Any distributed program has three types of events
  - Local actions
  - Message Send
  - Message Receive
- Can then view a distributed program as a sequential execution of the above events.
- When one talks about events across processors, things get tricky.

A Distributed Program in Execution

- Events can change the state of one or more process as follows.
  - A local action: aka an internal event
    - changes the state of the process where the event occurs
  - A send event: denoted send(m) for message m,
    - changes the state of the process sending the message, and
    - the state of the channel that is carrying the message m.
  - A receive event: denoted recv(m) for message m,
    - changes the state of the process that receives the message, and
    - the state of the channel on which the message is received.

A Distributed Program in Execution

- Think of events happening at a process P_i.
- One can order these events sequentially,
- In other words, place a linear order, $\rightarrow_i$,on these events.
- Let $E_i = \{e^1_i, e^2_i, \ldots\}$, be the events at process P_i.
- A linear order H_i is a binary relation $\rightarrow_i$ on the events h_i such that $e^k_i \rightarrow_i e^j_i$ if and only if the event e^k_i occurs before the event e^j_i at Process P_i.
- The dependencies captures by $\rightarrow_i$ are often called as causal dependencies among the events h_i at P_i.

A Distributed Program in Execution

- What about events generated due to messages?
- Consider a binary relation $\rightarrow_{msg}$ across messages exchanged.
- Clearly, for any message m, send(m) $\rightarrow_{msg}$ recv(m).
- These two relations $\rightarrow_i$ and $\rightarrow_{msg}$ allow us to view the execution of a distributed program in the picture shown next.

A Distributed Program in Execution

- One can now view the execution of a distributed program as a collection of events.
- Consider the set of events $E = U_i\ E_i$, where h_i is the events that occurred at process P_i.
- Define a binary relation $\rightarrow$ expressing causality among pairs of events that possibly occur at different processes. $\rightarrow$ is defined as:

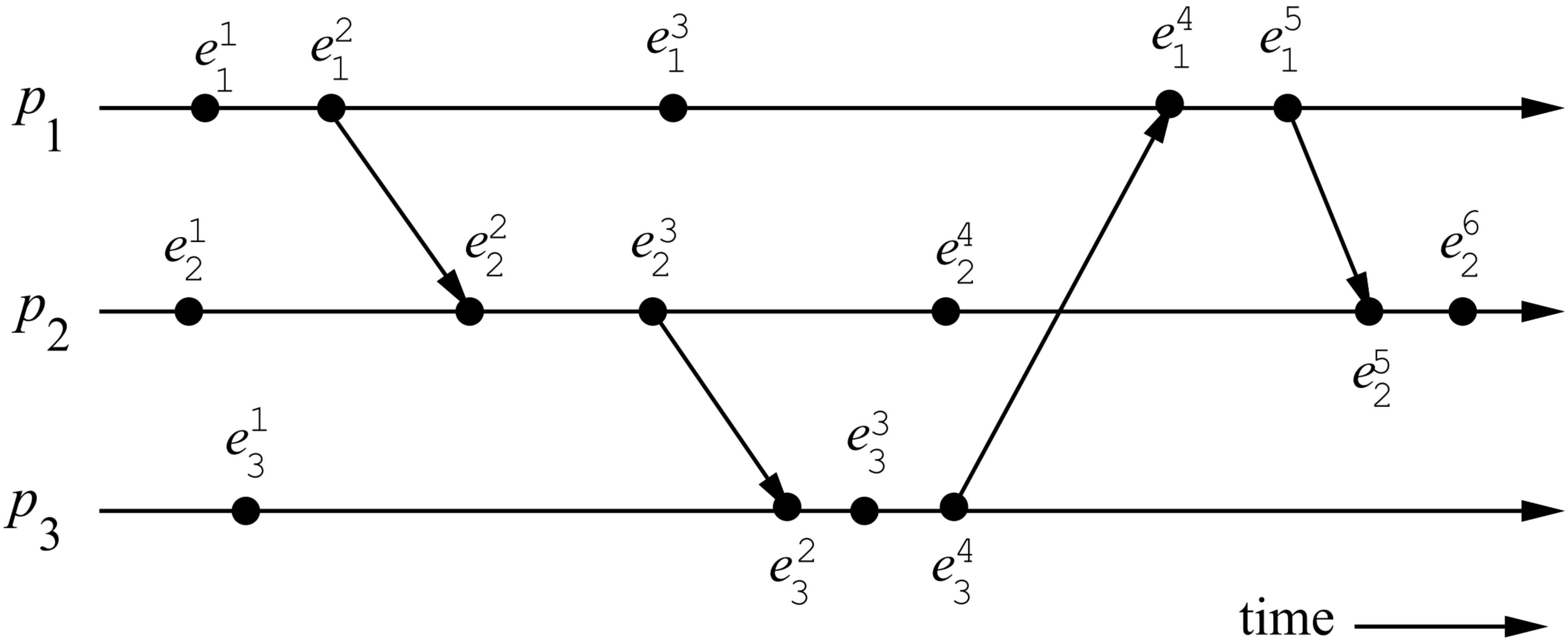
$$e^x_i \rightarrow e^y_j \text{ iff } \begin{cases} i = j \text{ and } x < y, \quad \text{or} \\ e^x_i \rightarrow_{msg} e^y_j, \quad \text{or} \\ \text{There exists } e^z_k \text{ in E s.t. } e^x_i \rightarrow e^z_k \text{ and} \\ e^z_k \rightarrow e^y_j \end{cases}$$

A Distributed Program in Execution

- In light of the above relation $\rightarrow$, we can now define Logical Concurrency
- Two events are logically concurrent if and only if the events do not causally affect each other. In other words, $e_i \parallel e_j \leftrightarrow Not(e_i \rightarrow e_j)$ and $Not(e_j \rightarrow e_i)$.
- Note that for logical concurrency of two events, the events may not occur at the same time.

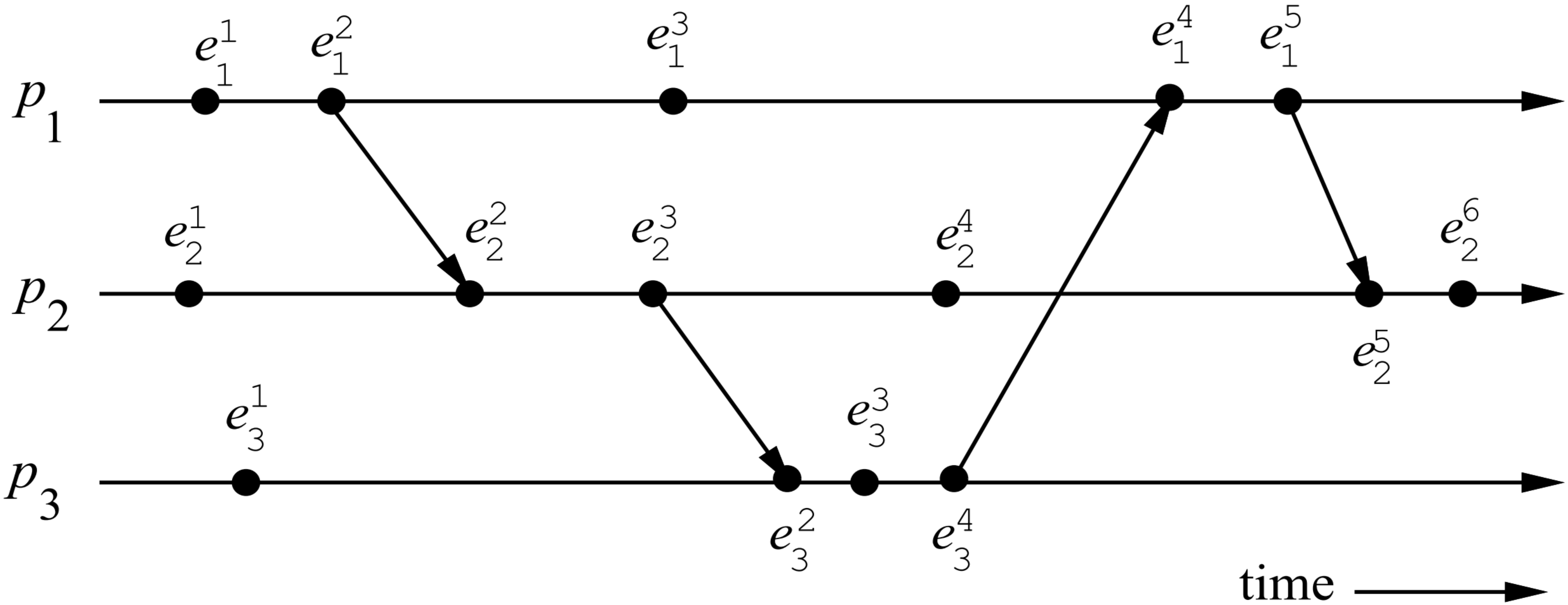
A Distributed Program in Execution

- Examples – Find logically concurrent events, and some non-trivial (across P_is) precedence among



A Distributed Program in Execution

- Examples – Find logically concurrent events, and some non-trivial (across P_is) precedence among



- Concurrent: e^4_2 and e^3_3, e^1_3 and e^3_1, e^4_2 and e^1_3
- Causal: $e^3_3 \rightarrow e^5_1$, $e^2_1 \rightarrow e^3_2$, $e^4_3 \rightarrow e^5_1$

Logical Time

- Armed with our notation of events and precedences amongst events, we now study logical time.
- We now see how logical time can be maintained in a distributed system.
- Three ways to implement logical time -
  - scalar time,
  - vector time, and
  - matrix time

Logical Time

- Consider a distributed system with each processor having a logical clock.
- These logical clocks are updated according to a common set of rules.
- Events are assigned timestamps
- Causality between events inferred via the timestamps associated.
- Rule: If an event e_1 causally affects another event e_2, then the timestamp of e_1 is smaller than that of e_2.

Logical Time

- Consider a distributed system with each processor having a logical clock.
- These logical clocks are updated according to a common set of rules.
- Events are assigned timestamps
- Causality between events inferred via the timestamps associated.
- Rule: If an event e_1 causally affects another event e_2, then the timestamp of e_1 is smaller than that of e_2.

Logical Time

- A formal definition to start with.
- A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation $<$.
- Relation $<$ is called the happened before or causal precedence.
  - Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time.
- The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T, denoted as C(e) and called the timestamp of e, and is defined as follows:

$$C : E \rightarrow T$$

such that the following property is satisfied:

for two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

Logical Time

- A formal definition to start with.
- A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation <.
- Relation < is called the happened before or causal precedence.
  - Intuitively, this relatio
    relation provided by the phy e.
- The logical clock C is a functio t maps an event e in a distributed system to an ele nt in the time domain T, denoted as C(e) and called t e timestamp of e, and is defined as follows:

What is E here?

$$C : E \rightarrow T$$

such that the following property is satisfied:

for two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

Logical Time

- This monotonicity property is called the clock consistency condition.

for two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

- Strongly Consistent: When T and C satisfy the condition:

for two events e_i and e_j , $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$.
then the system of clocks is said to be strongly consistent.

Logical Time

- Each processor needs some data structure to represent logical time.
  - A local logical clock, denoted by lc_i, that helps process p_i measure its own progress.
  - A logical global clock, denoted by gc_i, that is a representation of process p_i's local view of the logical global time.
    - Allows p_i to assign consistent time stamps to its local events.
    - Typically, lc_i is a part of gc_i.
- Each processor needs a protocol to update the data structures to ensure the consistency condition.

Logical Time

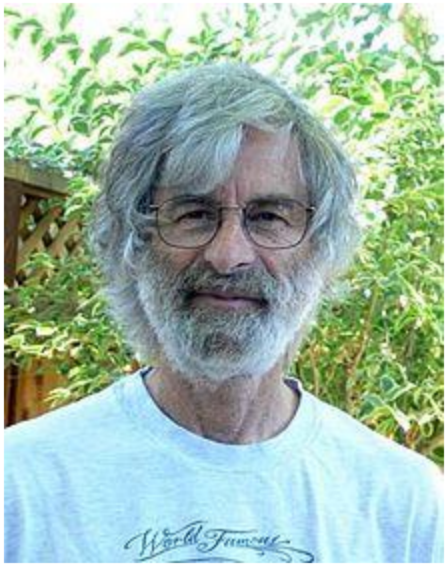
- Each processor needs some data structure to represent logical time.
- Each processor needs a protocol to update the data structures to ensure the consistency condition.
- The protocol is specified by two rules:
  - Rule 1: Specify how the local logical clock is updated by a process when it executes an event.
  - Rule 2: Specify how a process updates its logical global clock to update its view of the global time and global progress.

Logical Time

- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.
- However, all logical clocks systems implement Rule 1 and Rule 2 to maintain the monotonicity property of causality.

Scalar Time

- Proposed by Lamport in 1978
  - Lamport won the 2013 Turing award for "fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, ...”.



Scalar Time

- Time domain is the set of non-negative integers.
- Data structure: The logical local clock of a process p_i and its local view of the global time are combined into one integer variable C_i.
- Rule1 and Rule2 to update the clocks are as follows:
- Rule1: Before executing an event (send, receive, or internal), process p_i executes the following:

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

- In general, every time Rule1 is executed, d can have a different value; however, typically d is kept at 1.

Scalar Time

- Data structure: The logical local clock of a process p_i and its local view of the global time are combined into one integer variable C_i.
- Rule1: Before executing an event (send, receive, or internal), process p_i executes the following

  $C_i := C_i + d \ (d > 0)$
- Rule 2: Each message piggybacks the clock value of its 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 Rule1.
  3. Deliver the message.

Scalar Time

- Example. Use d = 1 and assign time stamps for the events.

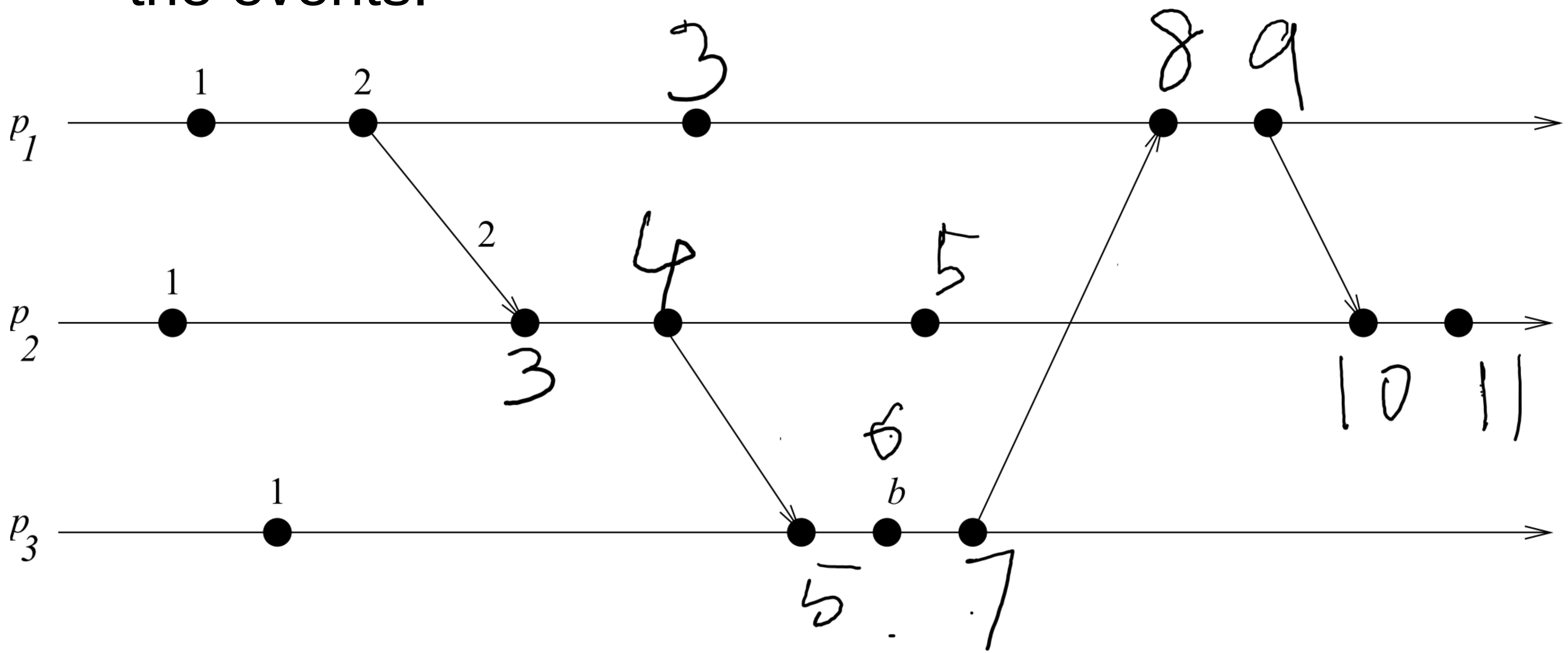


Figure 3.1: The space-time diagram of a distributed execution.

Scalar Time

- Example

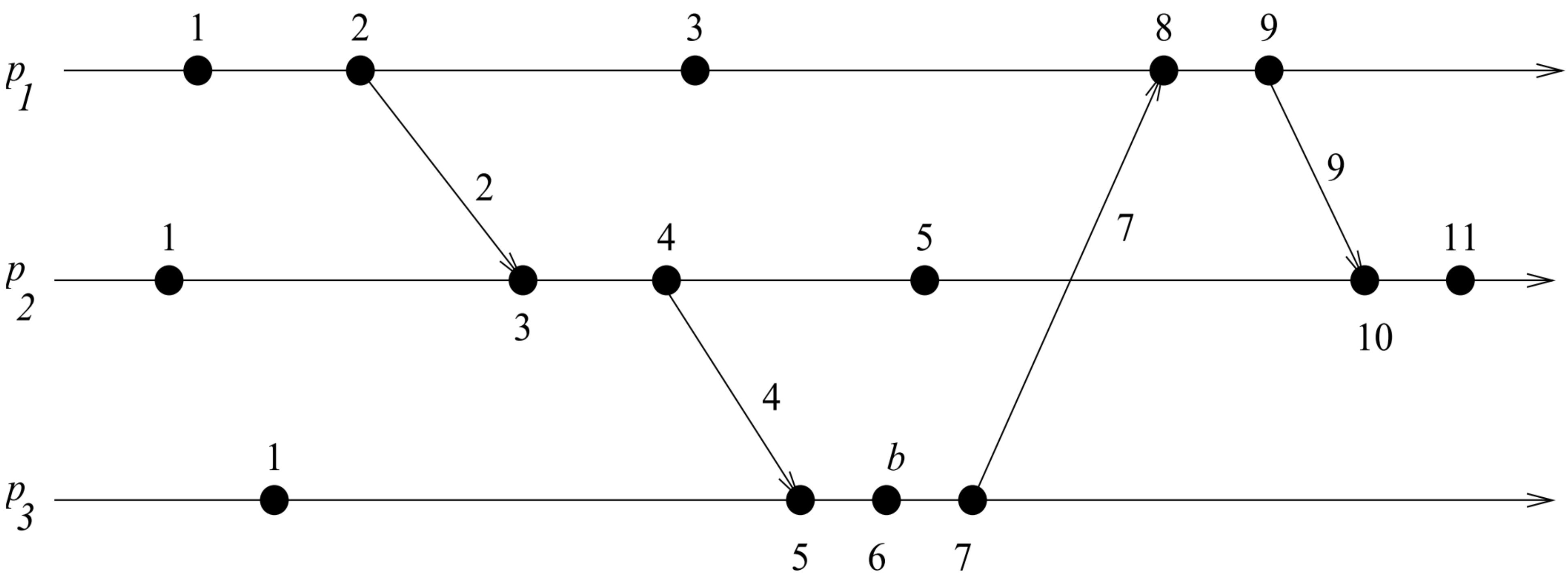


Figure 3.1: The space-time diagram of a distributed execution.

Scalar Time

- Let us study the basic properties of the scalar time.
- Monotonicity
  - For two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

- Total Ordering: Can use the logical time given by the scalar clocks to induce a total order .
  - Note that the timestamps alone do not induce a total order.
  - Two events at different processors can have an identical timestamp.
  - But the tuple (t, i) for each event with $(t_1, i_1) < (t_2, i_2)$ if either $t_1 < t_2$ or ($(t_1 == t_2)$ and $i_1 < i_2$) is a total order.
  - This total order is consistent with the relation $\rightarrow$.

Scalar Time

- Event Counting: Set the increment d to 1 always.
- If some event e has a timestamp t, then e is dependent on t – 1 other events to occur.
- This can be called as the height of event e.

- Strong Consistency: Note that scalar time does not provide strong consistency. [Strong consistency requires that $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$.]
  - No proof required. Example suffices. Refer to the timeline again.

Scalar Time

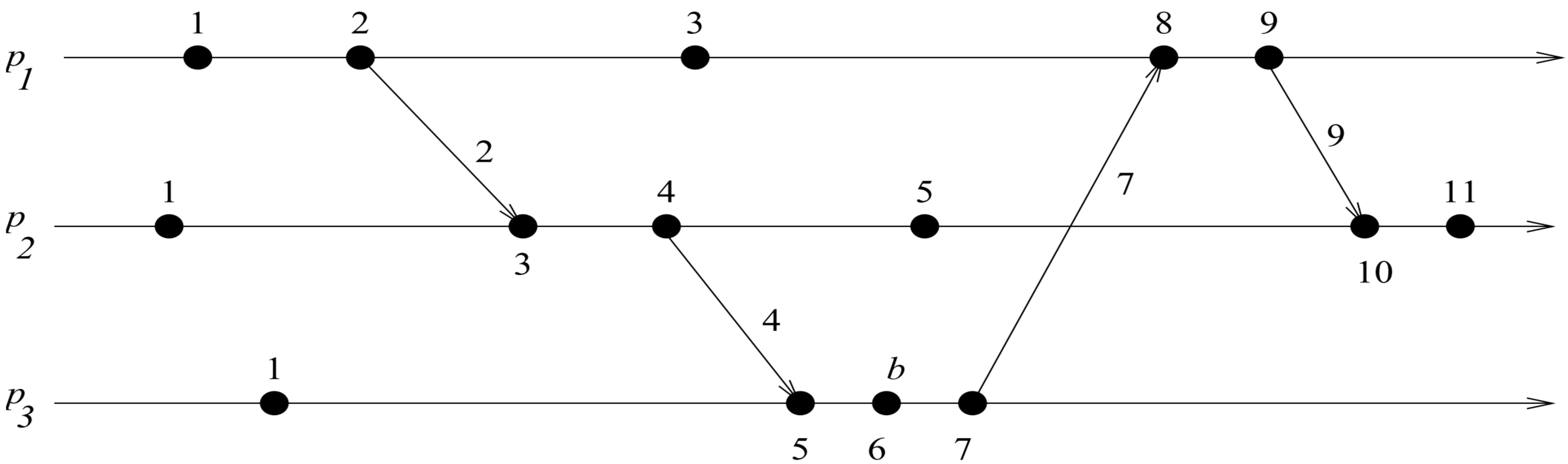


Figure 3.1: The space–time diagram of a distributed execution.

- Strong Consistency: Note that scalar time does not provide strong consistency. [Strong consistency requires that $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$.]
  - No proof required. Example suffices. Refer to the timeline again.

Scalar Time

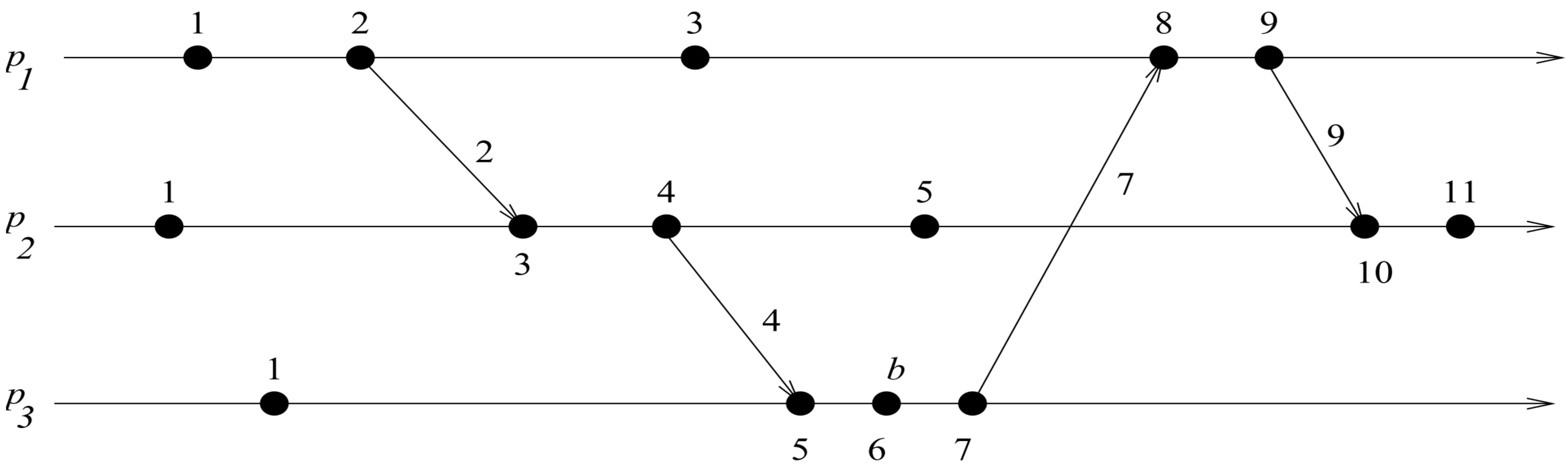


Figure 3.1: The space-time diagram of a distributed execution.

- Strong Consistency: Note that scalar time does not provide strong consistency. [Strong consistency requires that $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$.]
  - No proof required. Example suffices. Refer to the timeline again. Consider the third event at p_1 and p_2.

Vector Time

- One of the limitations of scalar time is the lack of strong consistency.
- Strong consistency not achieved as the data structure used in scalar time – a single time – that represents the logical local clock is reused to represent the logical global clock.
  - This means that the causality of events across processors is lost.
- Vector time solves this problem but with big data structures.

Vector Time

- Time is represented by an n-dimensional non-negative integer vector.
- Each process p_i maintains a vector vt_i [1..n], where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process 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 has progressed till x.
- The entire vector vt_i constitutes p_i 's view of the global logical time and is used to timestamp events.

Vector Time – Update Rules

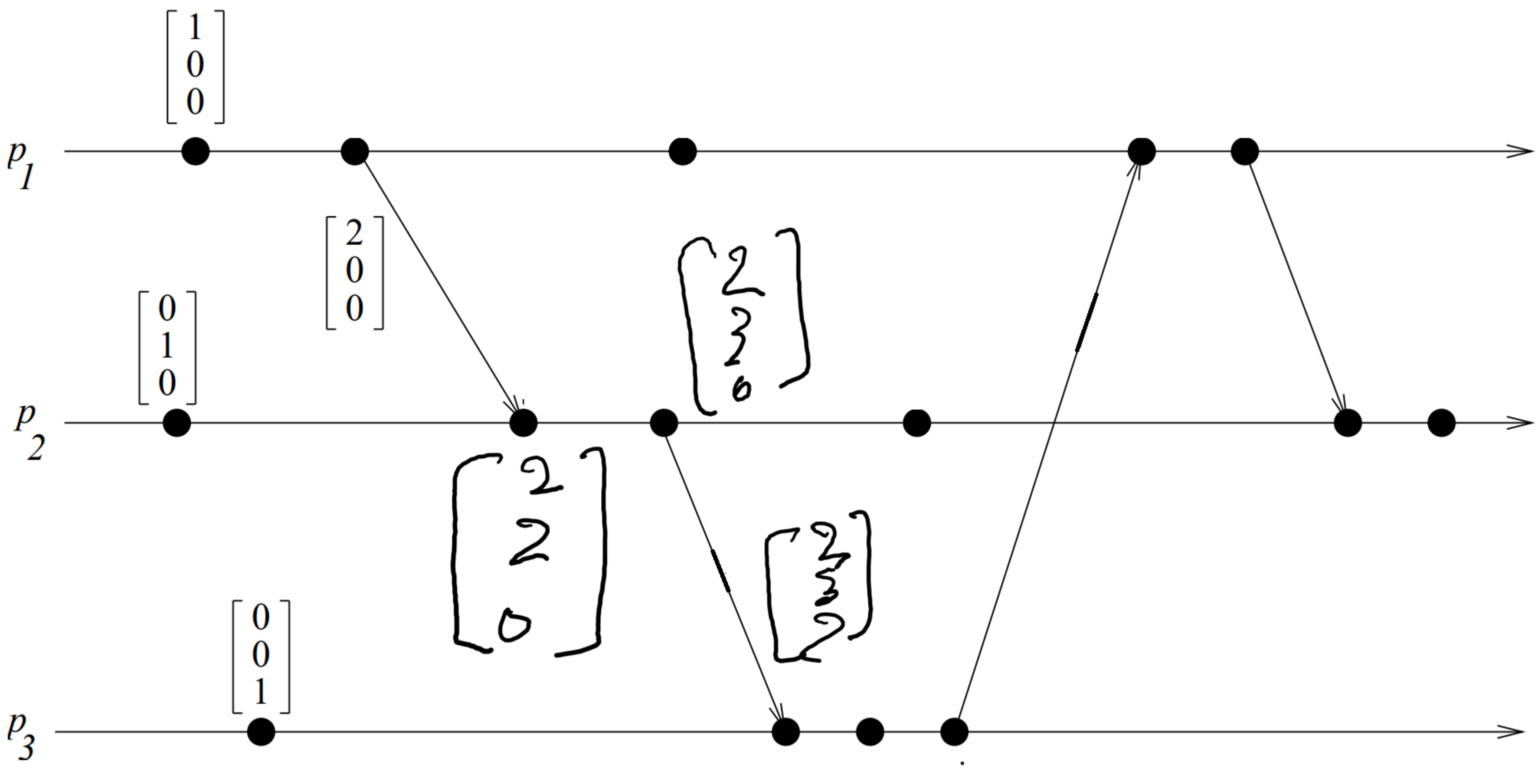
- Process p_i uses the following two rules to update its clock:
- Rule 1: Before executing an event, process p_i updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \quad \text{where } (d > 0)$$

- Rule2: Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message (m,vt), process p_i executes the following sequence of actions:
  1. Update its global logical time as follows:
  2. $1 \leq k \leq n : vt_i[k] := \max(vt_i[k], vt[k])$
  3. Execute Rule1.
  4. Deliver the message m.
- The timestamp of an event is the value of the vector clock of its process when the event is executed.

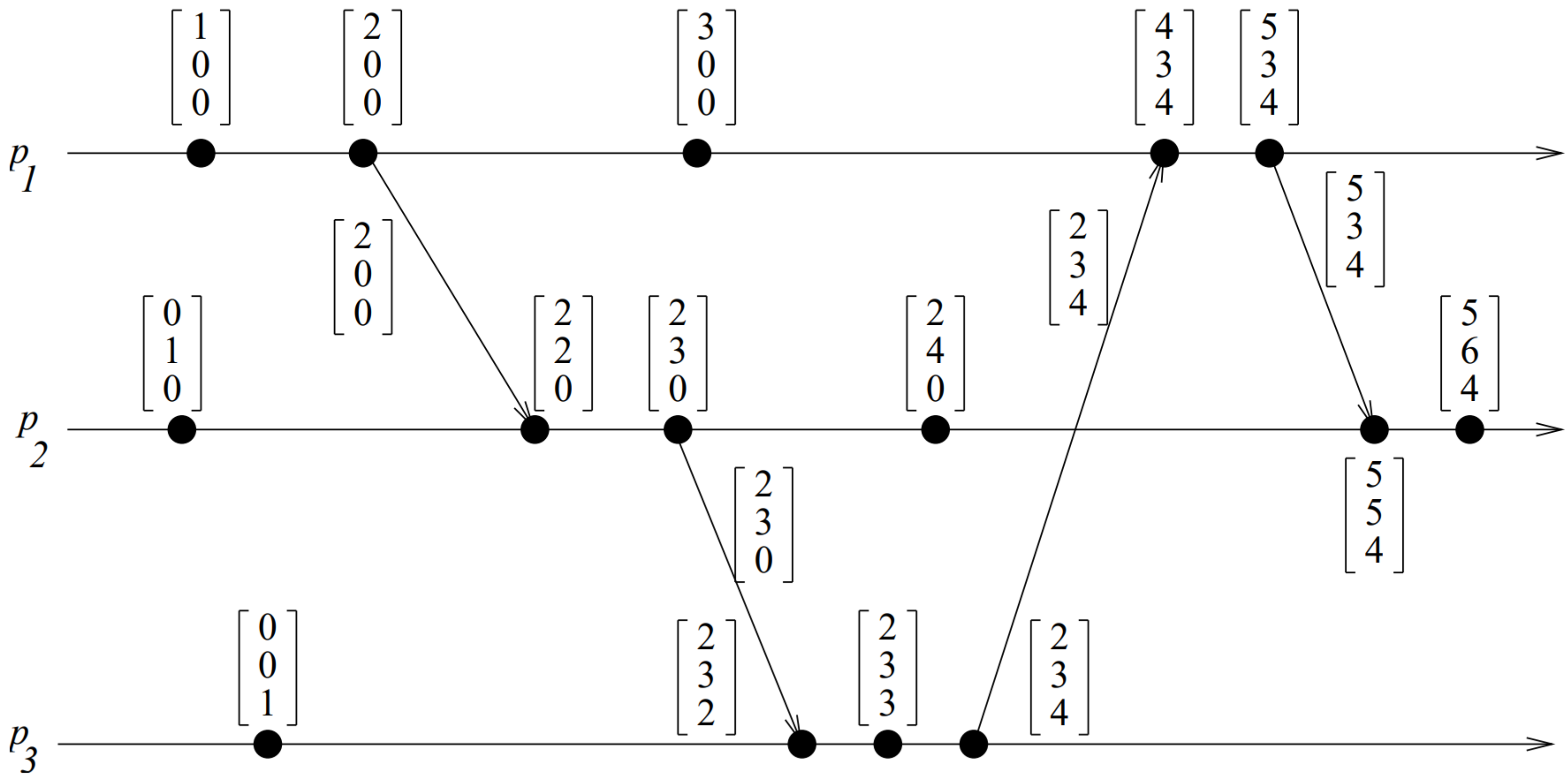
Vector Time – Example

- Initially, the vector clock is set to [0, 0, 0, ..., 0] and d = 1.
- Fill the timestamps for the other events.



Vector Time – Example

- Initially, the vector clock is set to [0, 0, 0, ..., 0] and d = 1.
- Fill the timestamps for the other events.



Vector Time

- Using vector clocks, two timestamps vh and vk are compared as follows.
  - vh == vk iff for all indices i, vh[i] == vk[i]
  - vh $\leq$ vk iff for all indices i, vh[i] $\leq$ vk[i]
  - vh < vk iff vh $\leq$ vk and there exists an index i where vh[i] < vk[i].
  - vh || vk iff not(vh < vk) and not(vk < vh)

- So, the vector [1, 3, 4] is less than the vector [1, 5, 6].
- The vectors [2, 5, 3] and [3, 4, 4] are concurrent.

Properties of Vector Time

- Event Counting: Use d = 1 always.
- Then the i th component of vector clock at process p_i,$vt_i[i]$, denotes the number of events that have occurred at p_i until that instant.
- So, if an event e has timestamp vh, vh[j] denotes the number of events executed by process p_j that causally precede e.
- Further, $\left(\sum_j vh[j]\right) - 1$ represents the total number of events that causally precede e in the distributed computation.

Properties of Vector Time

- Strong Consistency: Vector clocks are strongly consistent.
- For any two events, we can determine if the events are causally related.
- Proof: Exercise.

Limitations of Vector Time

- Large message sizes owing to the vector being piggybacked on each message.
- The message overhead grows linearly with the number of processors in the system.
- When there are thousands of processors in the system, the message size becomes huge even if there are only a few events occurring in few processors.
- Few techniques exist to reduce the overhead.