

Lecture: 04

Models of Distributed Computation, Causality & Logical Time



Rajiv Misra

Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture we have discussed about the leader election problem in message-passing systems for a ring topology.
- Different algorithms for leader election problem were presented by taking the cases like anonymous/non-anonymous rings, uniform/non-uniform rings and synchronous/ asynchronous rings.

Preface

Content of this Lecture:

- In this lecture, we will discuss about the models of distributed computation, causality and a general framework of logical clocks in distributed systems.
- Also In the absence of global physical time in DS, we present three systems of logical time, namely, scalar, vector, and matrix time to capture causality between events of a distributed computation .

Models of Distributed Computation

Introduction

- A distributed system consists of a set of processors that are connected by a communication network. The communication network provides the facility of information exchange among processors.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.
- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down.

Distributed Program: Definition

- Distributed program is composed of a set of n asynchronous processes, $p_1, p_2, \dots, p_n$.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let C_{ij} denote the **channel** from **process p_i** to **process p_j** and let m_{ij} denote a **message** sent by p_i to p_j .
- The message transmission delay is finite and unpredictable.

1. Model of Distributed Executions

- The execution of a process consists of a sequential execution of its actions.
- The actions are atomic and the actions of a process are modeled as three types of events, namely, **internal events**, **message send events**, and **message receive events**. Let e_i^x denote the **x th event at process p_i** .
- For a **message m** , let **$send(m)$** and **$rec(m)$** denote its send and receive events, respectively.
- The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state.
- An internal event changes the state of the process at which it occurs. A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).

Contd...

- The events at a process are linearly ordered by their order of occurrence.
- The execution of **process** p_i produces a sequence of **events** $e_i^1, e_i^2, \dots, e_i^x, \dots$ and is denoted by H_i where
$$H_i = (h_i, \rightarrow_i)$$
- h_i is the set of events produced by p_i and
- **binary relation** $\rightarrow_i$ defines a linear order on these events.
- Relation $\rightarrow_i$ expresses **causal dependencies** among the events of p_i .

Contd...

- The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process.
- A **relation** $\rightarrow_{msg}$ that captures the **causal dependency** due to message exchange, is defined as follows. For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m)$$

- Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events.

Contd...

- The evolution of a distributed execution is depicted by a space-time diagram.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.
- Since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to denote it as a dot on a process line.
- In the Figure 4.1, for process p_1 , the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

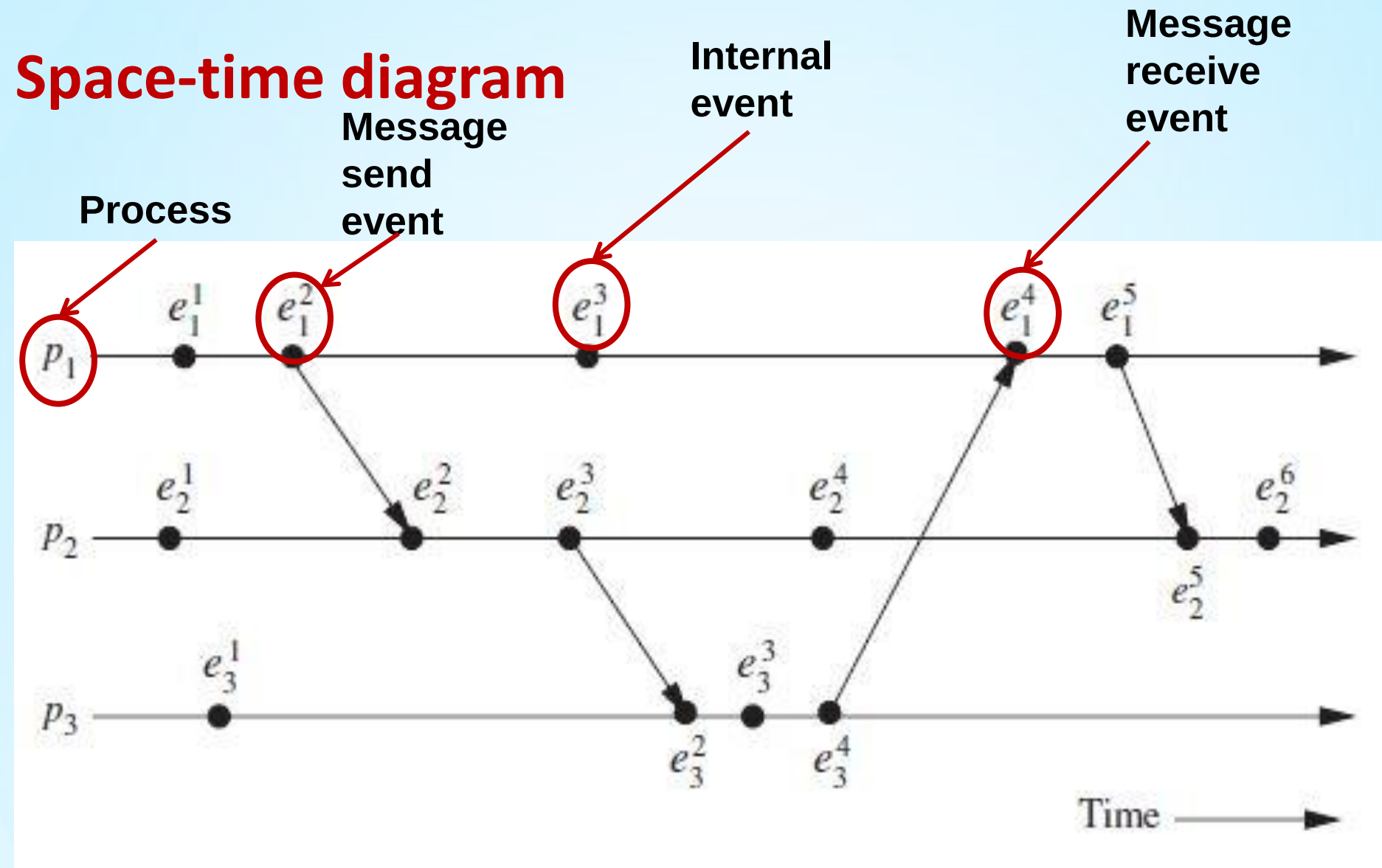


Figure 4.1: The space-time diagram of a distributed execution.

Preliminaries: Partial Order Relation

Definition:

A binary relation R on a set A is a **partial order** if and only if it is

- (i) reflexive,
- (ii) antisymmetric, and
- (iii) transitive.

The ordered pair $\langle A, R \rangle$ is called a **poset** (partially ordered set) when R is a partial order.

Example 1: The less-than-or-equal-to relation on the set of integers $\mathbb{Z}$ is a partial order, and the set $\mathbb{Z}$ with this relation is a poset.

Preliminaries: Total Order Relation

Definition:

A binary relation R on a set A is a **total order** if and only if it is

- (i) a partial order, and
- (ii) for any pair of elements a and b of A , $\langle a, b \rangle$ in R or $\langle b, a \rangle$ in R .

That is, every element is related with every element one way or the other.

A total order is also called a **linear order**.

Example 2: The less-than-or-equal-to relation on the set of integers I is a total order.

Causal Precedence Relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let $H = \cup_i h_i$ denote the set of events executed in a distributed computation.
- Define a binary relation $\rightarrow$ on the set H as follows that expresses causal dependencies between events in the distributed execution.
- The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as $H = (H, \rightarrow)$

$$\forall e_i^x, \forall e_j^y \in H, \quad e_i^x \rightarrow e_j^y \Leftrightarrow \left\{ \begin{array}{l} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{array} \right.$$

Contd...

- Note that the relation $\rightarrow$ is nothing but **Lamport's "happens before"** relation.
- For any two events e_i and e_j , if $e_i \rightarrow e_j$, then event e_j is directly or transitively dependent on event e_i . (Graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space-time diagram that starts at e_i and ends at e_j)
- For example, in Figure 4.1, $e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$.
- The relation $\rightarrow$ denotes flow of information in a distributed computation and $e_i \rightarrow e_j$ dictates that all the information available at e_i is potentially accessible at e_j .
- For example, in Figure 4.1 event e_2^6 has the knowledge of all other events shown in the figure.

Space-time diagram

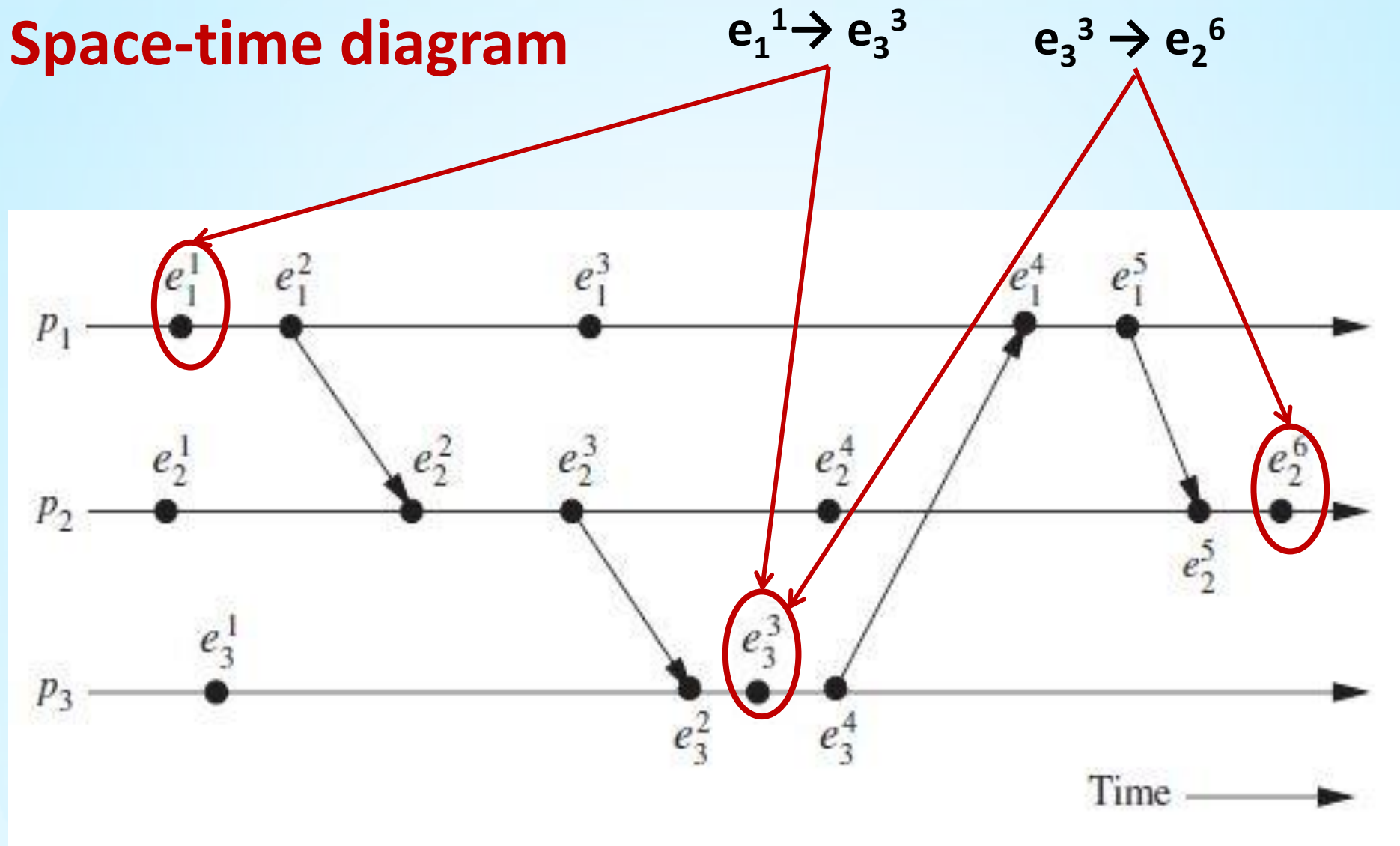


Figure 4.1: The space-time diagram of a distributed execution.

Concurrent events

- For any two events e_i and e_j , if $e_i \not\rightarrow e_j$ and $e_j \not\rightarrow e_i$, then events e_i and e_j are said to be concurrent (denoted as $e_i \parallel e_j$)
- In the execution of Figure 4.1, $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$
- The relation $\parallel$ is not transitive; that is, $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$
- For example, in Figure 4.1, $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$, however, $e_3^3 \not\parallel e_1^5$
- **For any two events e_i and e_j in a distributed execution,**
 $e_i \rightarrow e_j$ or $e_j \rightarrow e_i$, or $e_i \parallel e_j$.

Space-time diagram

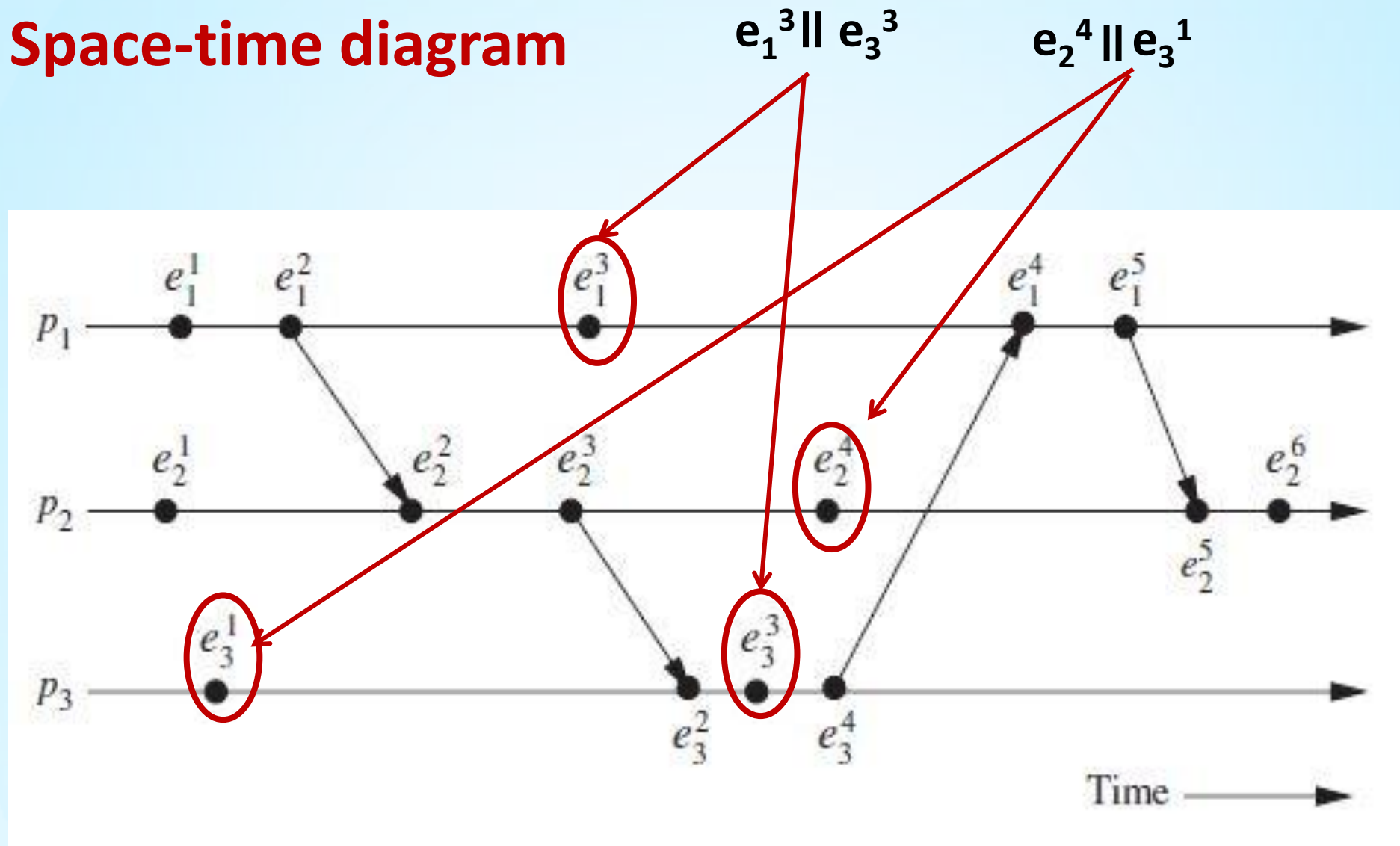


Figure 4.1: The space-time diagram of a distributed execution.

Logical vs. Physical Concurrency

- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other.
- Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time.
- Note that two or more events may be logically concurrent even though they do not occur at the same instant in physical time.

Contd...

- For example, in Figure 4.1, events in the set $\{e_1^3, e_2^4, e_3^3\}$ are logically concurrent, but they occurred at different instants in physical time. However, note that if processor speed and message delays had been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or in what order in the physical time they occur does not change the outcome of the computation.

Space-time diagram

events in the set $\{e_1^3, e_2^4, e_3^3\}$ are logically concurrent

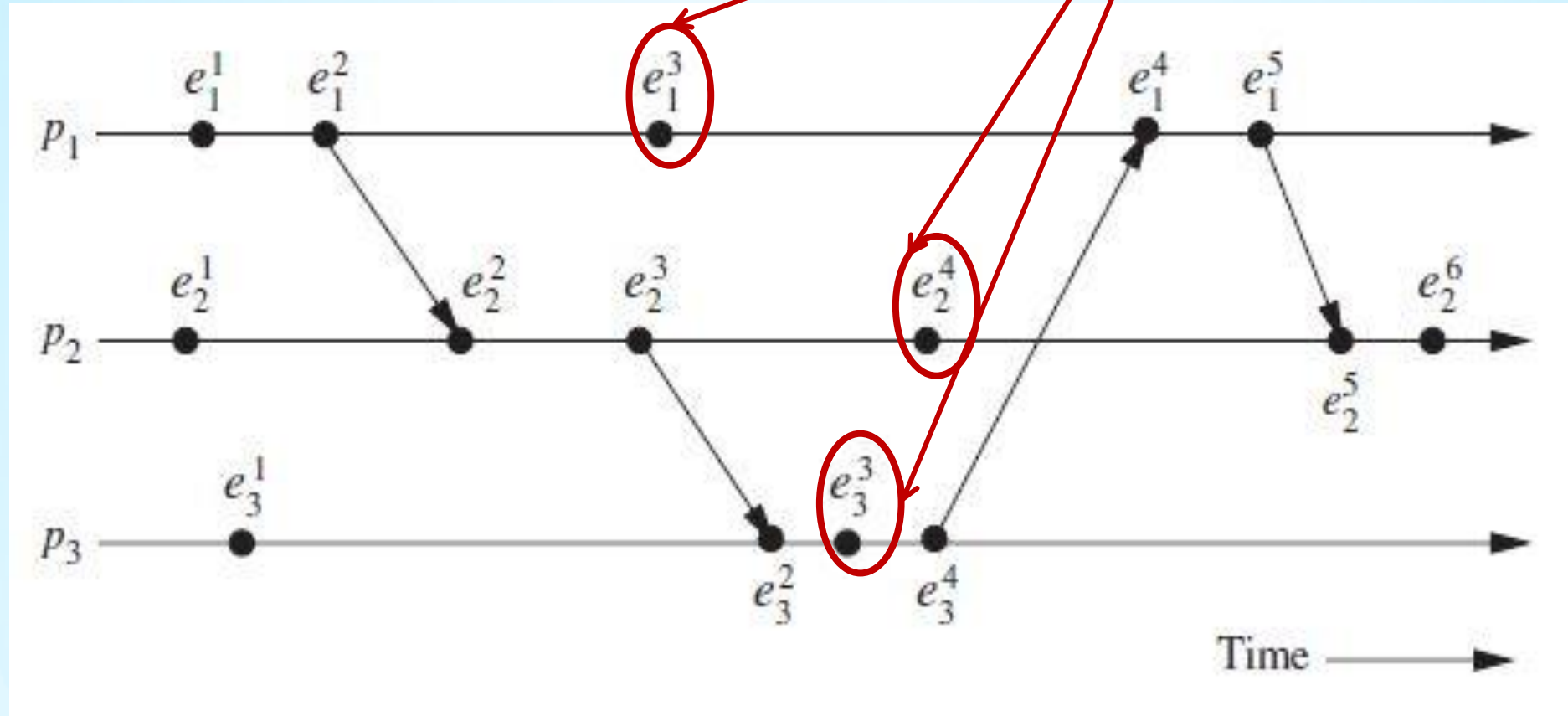


Figure 4.1: The space-time diagram of a distributed execution.

2. Models of Communication Networks

- There are several models of the service provided by communication networks, **namely, FIFO, Non-FIFO, and causal ordering.**
- In the **FIFO model**, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the **non-FIFO model**, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

2. Models of Communication Networks contd...

- The “causal ordering” model is based on Lamport’s “happens before” relation. A system that supports the causal ordering model satisfies the following property:
- **CO:** For any two messages m_{ij} and m_{kj} , if $send(m_{ij}) \rightarrow send(m_{kj})$, then $rec(m_{ij}) \rightarrow rec(m_{kj})$
- This property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that **CO $\subset$ FIFO $\subset$ Non-FIFO**)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

Causality & Logical Time

2. Concept of Causality: Introduction

- The **concept of causality** between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have a global physical time; it is possible to realize only an approximation of it.
- As asynchronous distributed computations make progress in spurts, the logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.

Contd...

- This lecture discusses three ways to implement logical time: scalar time, vector time, and matrix time.
- Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.
- The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems, such as distributed algorithms design(mutual exclusion, replicated databases, deadlock detection), tracking of dependent events, knowledge about the progress of a computation, and concurrency measures.

Framework for a System of Logical Clocks

- 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:

$$C : H \mapsto T$$

such that the following property is satisfied:

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

This monotonicity property is called the clock consistency condition.

Contd...

- When T and C satisfy the following condition,
for two events e_i and e_j , $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$
the system of clocks is said to be **strongly consistent**.

Implementing Logical Clocks

- Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol to update the data structures to ensure the consistency condition.
- Each process p_i maintains data structures that allow it the following two capabilities:
 - (i) A **local logical clock**, denoted by l_{C_i} , that helps process p_i measure its own progress

Implementing Logical Clocks

(ii) A *logical global clock*, denoted by gc_i , that is a representation of process p_i 's local view of the logical global time.

Typically, lc_i is a part of gc_i .

- The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

R1: This rule governs how the local logical clock is updated by a process when it executes an event.

R2: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.

- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.

Scalar Time

- Proposed by **Lamport in 1978** as an attempt to **totally order events** in a distributed system.
- Time domain is the set of non-negative integers.
- The logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i .
- Rules $R1$ and $R2$ to update the clocks are as follows:

$R1$: Before executing an event (send, receive, or internal), process p_i executes the following:

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

- In general, every time $R1$ is executed, d can have a different value; however, typically d is kept at 1.

Scalar Time

R2: 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:

- $C_i := \max(C_i, C_{msg})$
- Execute *R1*
- Deliver the message

Figure 4.2 shows evolution of scalar time.

Evolution of Scalar Time

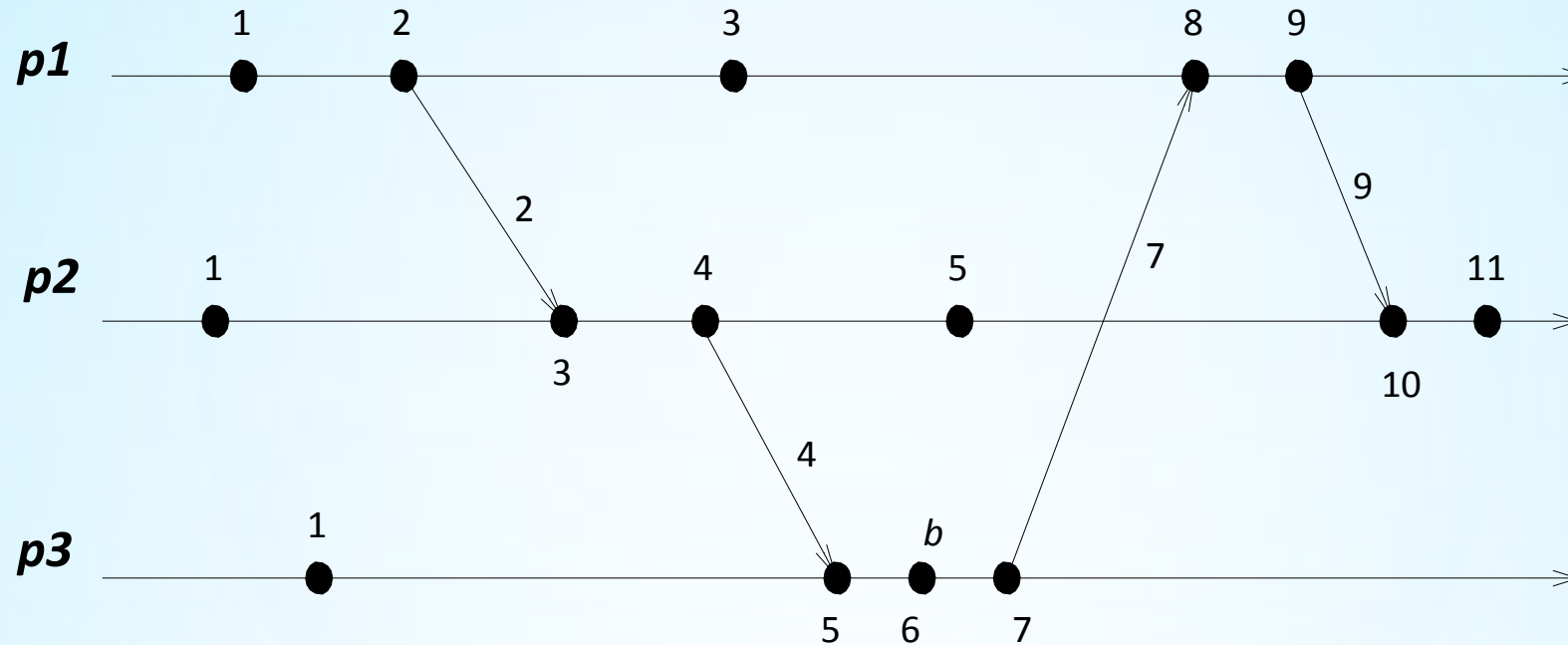


Figure 4.2: The space-time diagram of a distributed execution.

Basic Properties

Consistency Property

- Scalar clocks satisfy the monotonicity and hence the consistency property: for two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure 4.2, the third event of process P_1 and the second event of process P_2 have identical scalar timestamp.

Identical Scalar Time Stamp

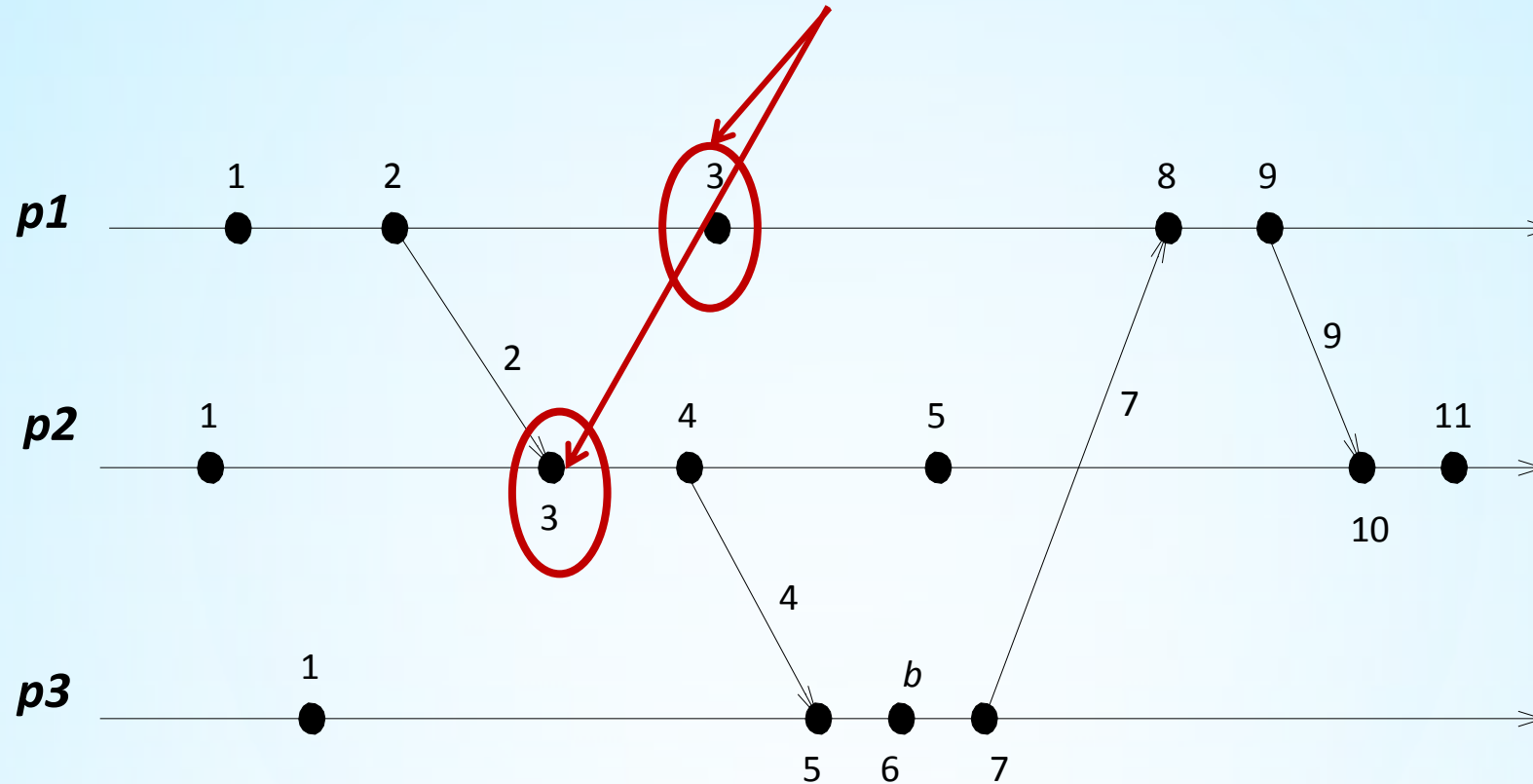


Figure 4.2: The space-time diagram of a distributed execution.

Total Ordering

A ***tie-breaking mechanism*** is needed to order such events. A tie is broken as follows:

- Process identifiers are linearly ordered and tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple (t, i) where t is its time of occurrence and i is the identity of the process where it occurred.
- The total order relation $<$ on two events x and y with timestamps (h, i) and (k, j) , respectively, is defined as follows:

$$x < y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

Properties...

Event counting

- If the increment value d is always 1, the scalar time has the following interesting property: if event e has a timestamp h , then $h-1$ represents the minimum logical duration, counted in units of events, required before producing the event e ;
- We call it the height of the event e .
- In other words, $h-1$ events have been produced sequentially before the event e regardless of the processes that produced these events.
- For example, in Figure 4.2, five events precede event b on the longest causal path ending at b .

Five events precede event b on the longest causal path ending at b

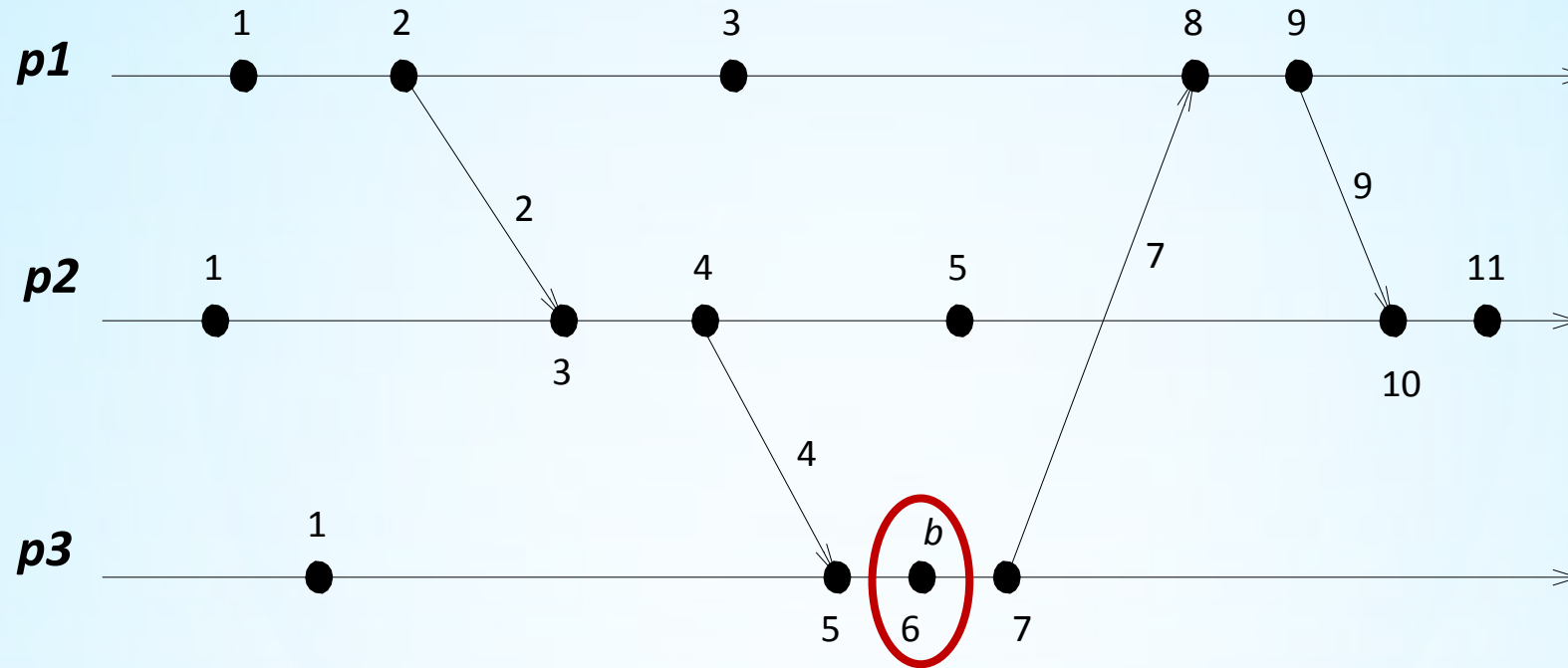


Figure 4.2: The space-time diagram of a distributed execution.

Properties...

No Strong Consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j , $C(e_i) < C(e_j) \Rightarrow e_i \not\rightarrow e_j$
- For example, in Figure 4.2, the third event of process P_1 has smaller scalar timestamp than the third event of process P_2 . However, the former did not happen before the latter.
- The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.
- For example, in Figure 4.2, when process P_2 receives the first message from process P_1 , it updates its clock to 3, forgetting that the timestamp of the latest event at P_1 on which it depends is 2.

Clock Updation

smaller scalar timestamp

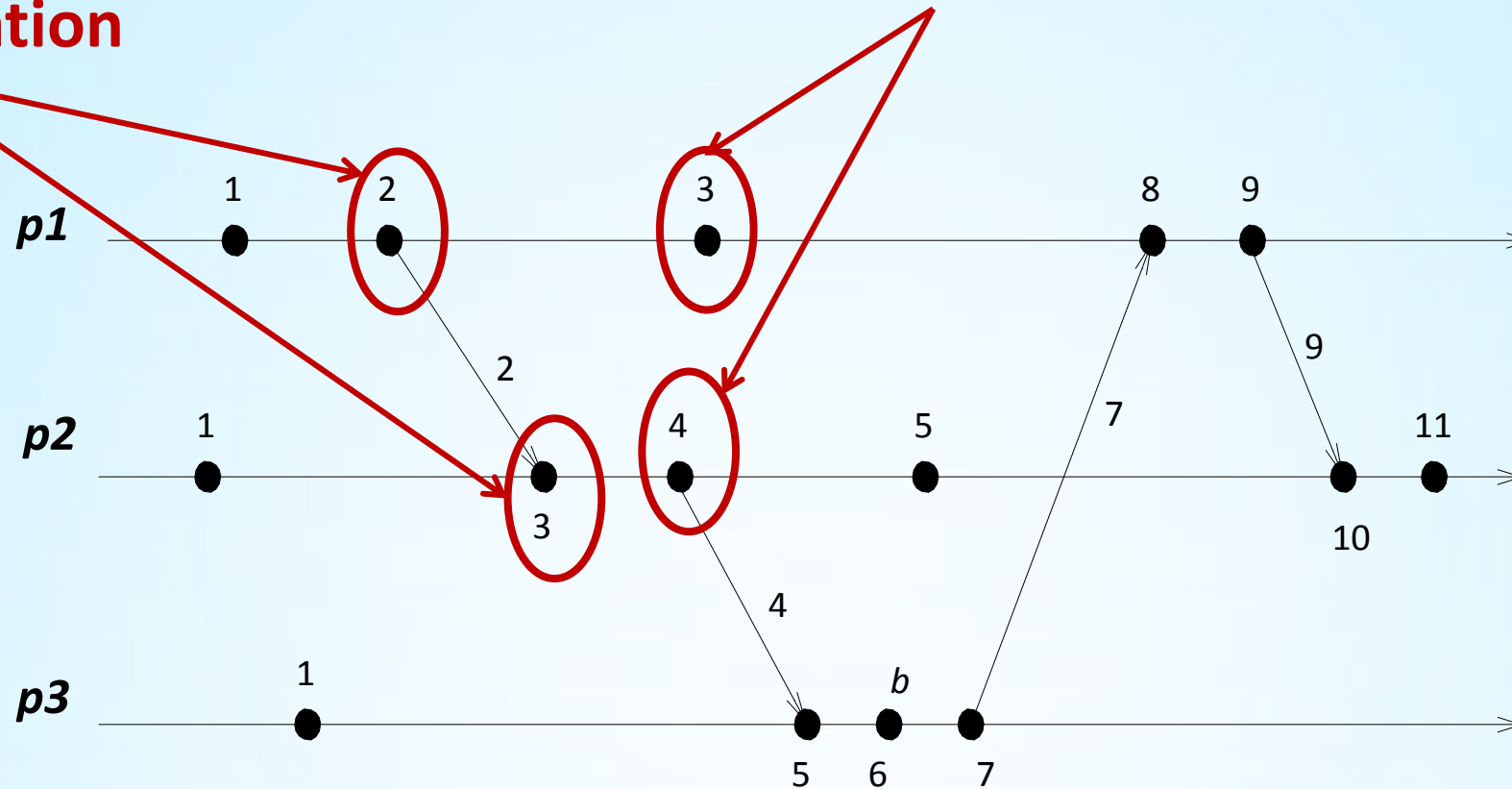


Figure 4.2: The space-time diagram of a distributed execution.

Vector Time

- The system of vector clocks was developed independently by **Fidge, Mattern and Schmuck**.
- In the system of vector clocks, the time domain is represented by a set of n -dimensional non-negative integer vectors.
- 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

Process p_i uses the following two rules $R1$ and $R2$ to update its clock:

$R1$: Before executing an event, process p_i updates its local logical time as follows:

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

$R2$: 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:

- Update its global logical time as follows:

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

- Execute $R1$

- Deliver the message m

Vector Time

- The timestamp of an event is the value of the vector clock of its process when the event is executed.
- Figure 4.3 shows an example of vector clocks progress with the increment value $d=1$.
- Initially, a vector clock is $[0, 0, 0, \dots, 0]$.

Example of Vector Clock

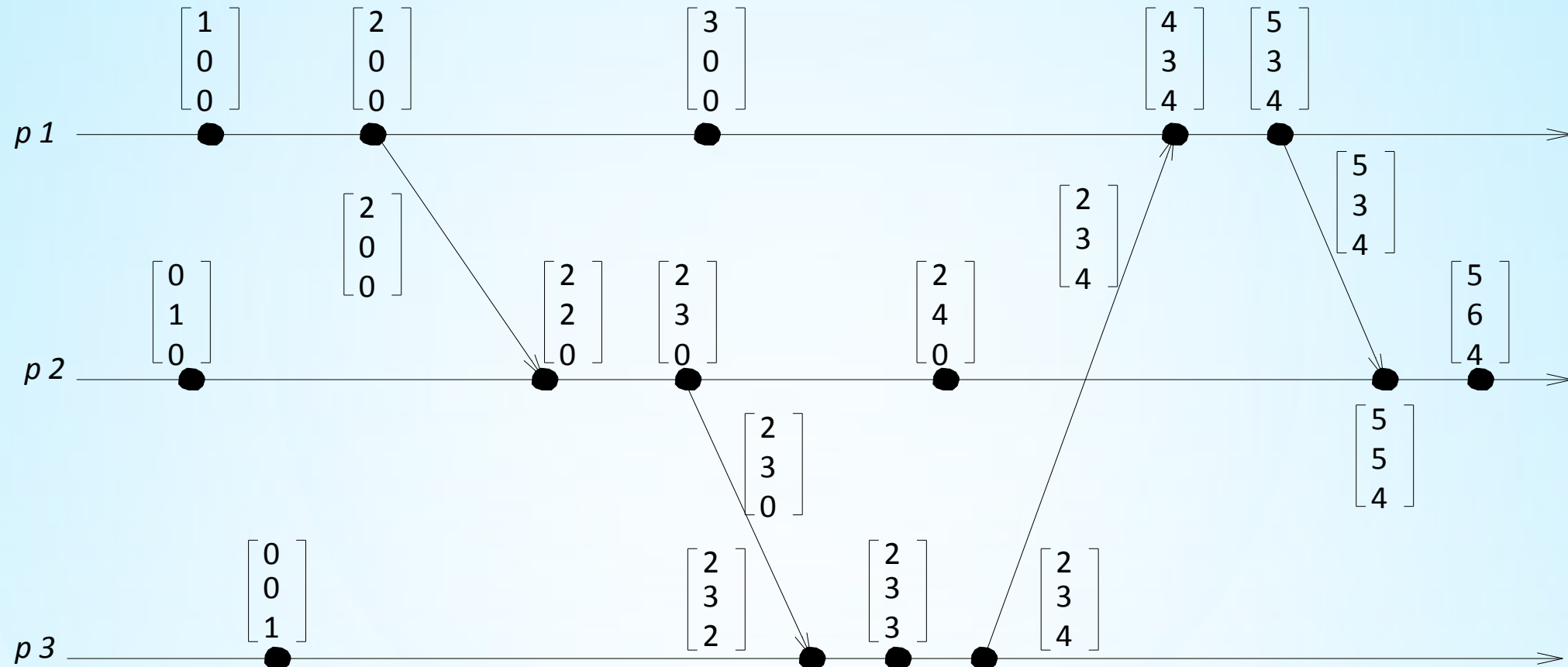


Figure 4.3: Evolution of vector time.

Comparing Vector Time Stamps

- The following relations are defined to compare two vector timestamps, vh and vk :

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh || vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)$$

- If the process at which an event occurred is known, the test to compare two timestamps can be simplified as follows: If events x and y respectively occurred at processes p_i and p_j and are assigned timestamps vh and vk , respectively, then

$$x \rightarrow y \Leftrightarrow vh[i] \leq vk[j]$$

$$x || y \Leftrightarrow vh[i] > vk[j] \wedge vh[j] < vk[i]$$

Properties of Vector Time

Isomorphism

- If events in a distributed system are timestamped using a system of vector clocks, we have the following property.
- If two events x and y have timestamps vh and vk , respectively, then

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

$$x || y \Leftrightarrow vh || vk$$

- Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.

Properties of Vector Time

Strong Consistency

- The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.
- However, Charron-Bost showed that the dimension of vector clocks cannot be less than n , the total number of processes in the distributed computation, for this property to hold.

Event Counting

- If $d=1$ (in rule $R1$), 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 . Clearly, $\sum vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation.

Conclusion

- In a distributed system, a set of processes communicate by exchanging messages over a communication network. A distributed computation is spread over geographically distributed processes. The processes do not share a **common global memory or a physical global clock**, to which processes have instantaneous access.
- In this lecture we have presented the ***idea of logical time*** that was proposed by **Lamport in 1978 in an attempt to order events in distributed systems**.
- We have discussed two systems of logical clocks, namely, **scalar and vector clocks** to capture causality between events of a distributed computation.
- In upcoming lecture, we will discuss about the size of vector clock, matrix clocks, virtual time and physical clock synchronization.