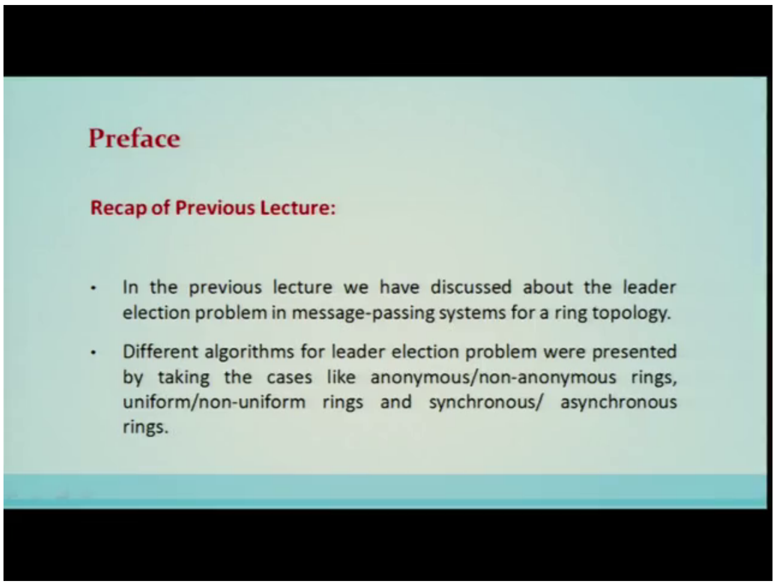
Distributed Systems
Dr. Rajiv Misra
Department of Computer Science and Engineering
Indian Institute of Technology, Patna

Lecture - 04
Models of Distributed Computation, Causality and Logical Time

Lecture 4: Models of Distributed Computation, Causality and Logical Time. Preface: recap of previous lecture.

(Refer Slide Time: 00:26)



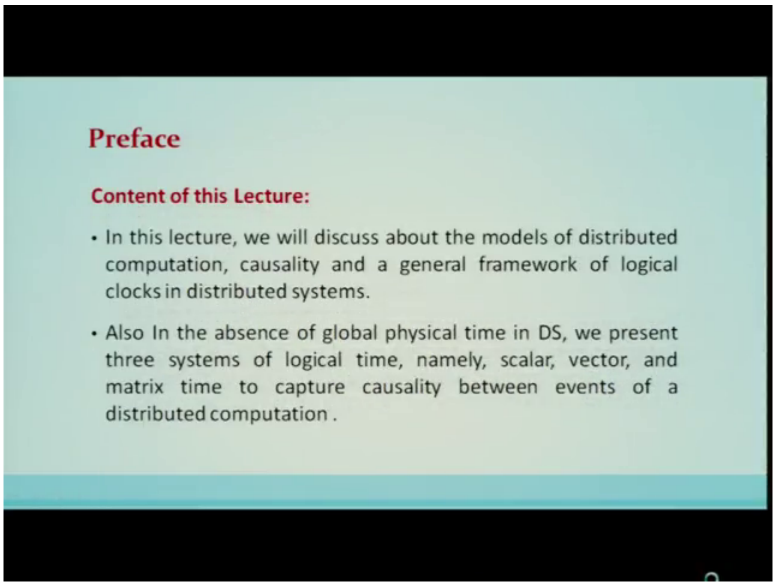
Preface

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.

In the previous lecture we have discussed about the leader election problem in the message passing system for a ring topology. We have also seen different algorithm for leader election problem by taking different cases of a topology, like anonymous, oblique, non-anonymous rings, uniform, oblique, non-uniform rings, synchronous and asynchronous rings.

(Refer Slide Time: 00:52)



Preface

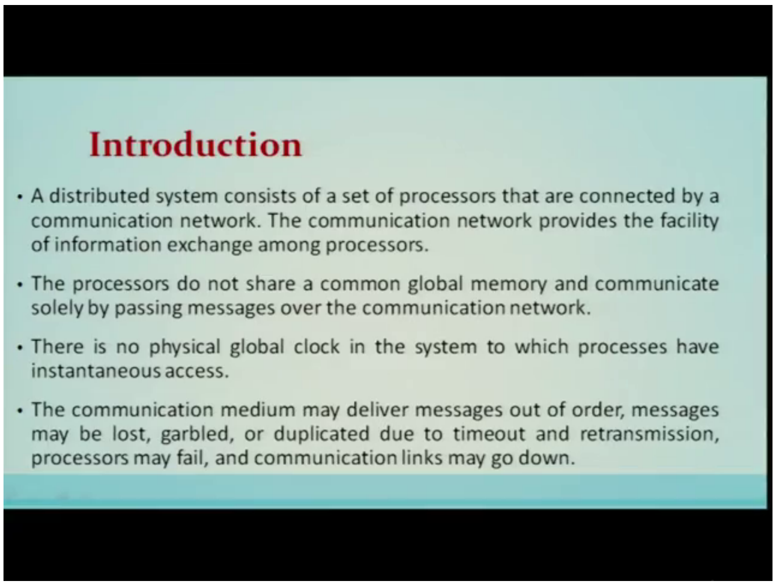
Content of this Lecture:

- 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 .

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

Before we start I should mention that causality is the concept of causality is the fundamental for the design of distributed systems. Usually causality is tract using physical time since you know that distributed system do not have a global physical time. So, there is a possibility to realize it using an approximation of it. So, logical clock is basically able to capture the fundamental monotonicity property associated with the causality of a distributed system. So, that is we are going to cover up in this part of the lecture models of distributed computation.

(Refer Slide Time: 02:10)



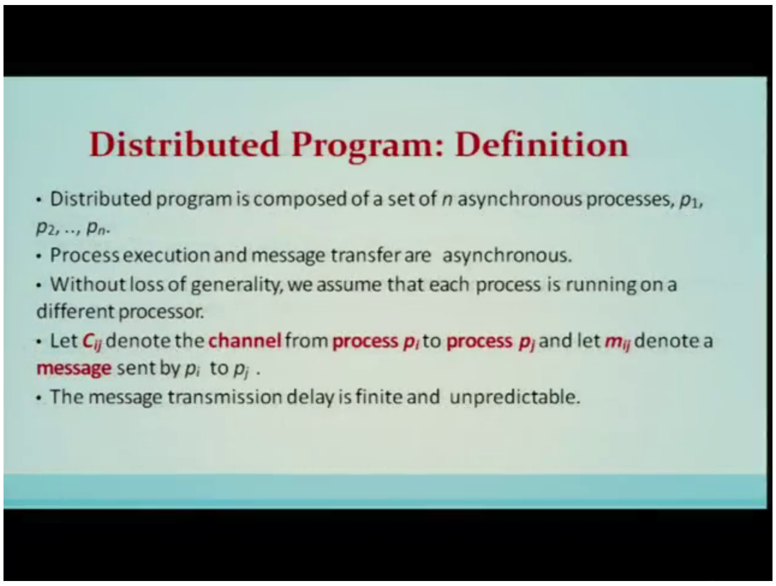
Introduction

- 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.

Introduction; 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 the processors. The processors do not share a common global memory and communicates only by passing the messages over the communication network.

There is no physical global clock in the system to which the processes instantaneous access. The communication medium may deliver the messages out of order messages may be lost garbled or duplicated due to the timeout entry transmission, processors may fail and communication link may go down.

(Refer Slide Time: 02:57)



Distributed Program: Definition

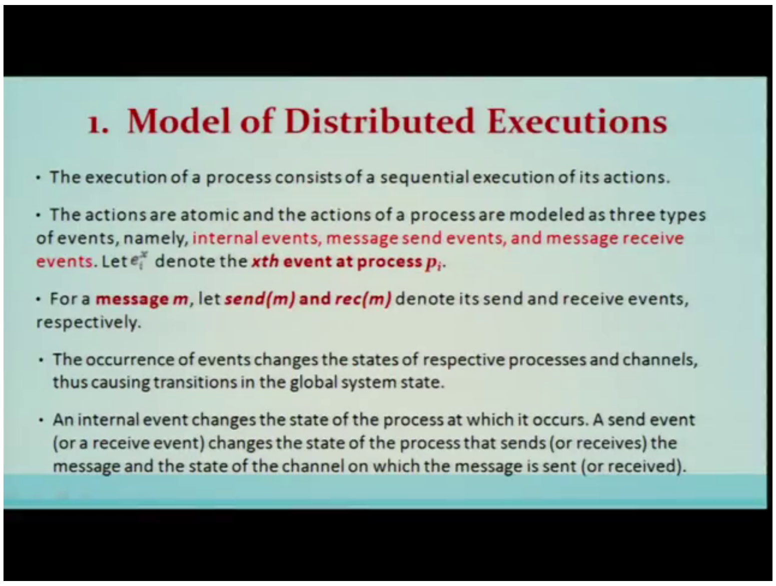
- Distributed program is composed of a set of *n* asynchronous processes, p_1, p_2, .., p_n.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let C_{ij} denote the **channel** from **process p_i** 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.

So, these are the characteristics of the distributed systems in which we are going to discuss how to write down the applications and the program.

So, about the distributed program definition distributed program is composed of a set of n asynchronous p 1 to p n process execution and the message transfers are asynchronous, without loss of generality we assume that each process is running on a different processor. So, by this way either we call it as process or a processor both are signifying the same thing here in this part of the discussion.

Now, channel let C ij denote the channel from a process p i 2 process p j and let m ij we denote the message send by the process i to process j here.

(Refer Slide Time: 03:53)



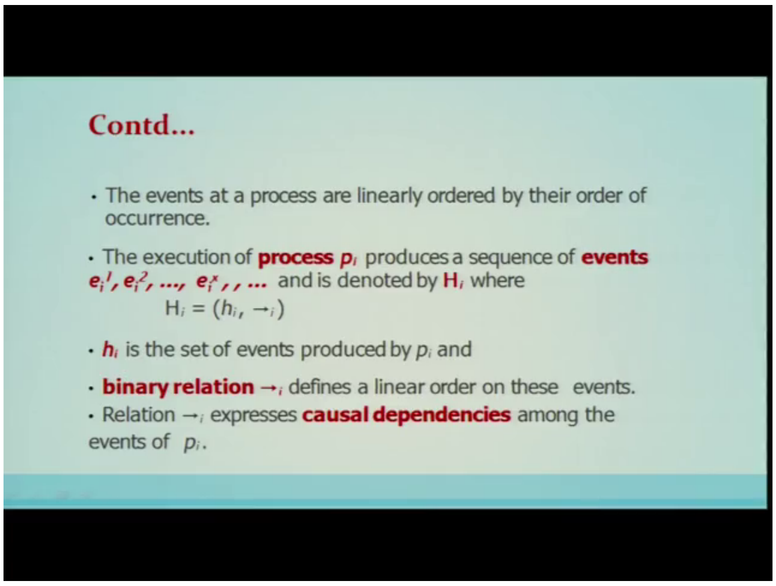
1. Model of Distributed Executions

- 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 ***xth* 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).

We assume that the message transmission delay is finite, but unpredictable models of distributed execution. The execution of a process consists of a sequential execution of it is actions. The actions are atomic and the actions of a process are modeled as 3 different type of events namely the internal events, message send event and message receive events. Let e e ix denote x-th event at a process pi for a message m let send m and receive m denote the send and receive events respectively.

The occurrences of events change the state of the respective processes and the channel thus causing the transitions in the global system state, and internally even changes the state of a process at which it occurs a send event or a receive event changes the state of a process that sends or receives the message and the state of a channel on which the message is sent.

(Refer Slide Time: 05:00)



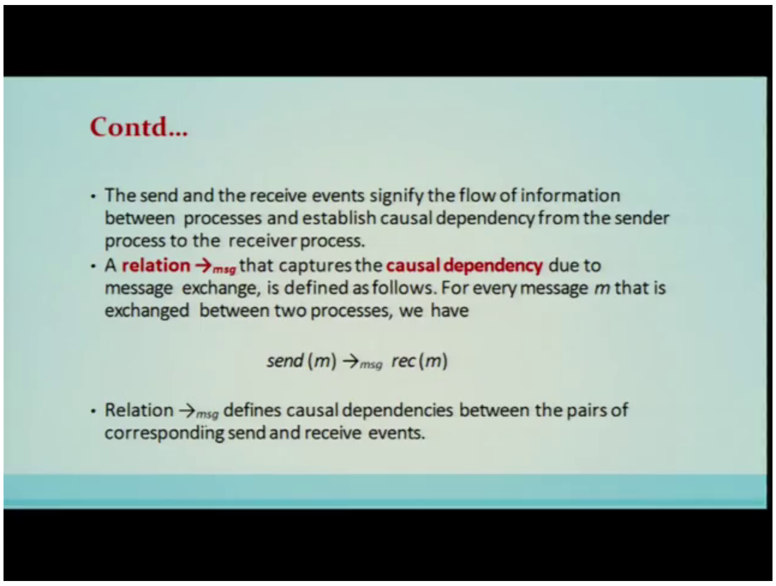
Contd...

- The events at a process are linearly ordered by their order of occurrence.
- The execution of **process** p_i produces a sequence of **events** $e_i^1, e_i^2, \ldots, e_i^x, \ldots$ 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.

The events at a process are linearly ordered by their order of occurrence. The execution of a process i produces a sequence of events that is e 1 e 2 and so on ex of a particular process i which is subscripted by i, and is denoted by a capital Hi. So, capital Hi is nothing, but small hi and the sequence in which they are these events are occurring and that is denoted by a binary relation which is shown as an arrow over here. Small hi is the set of events produced by pi and the binary relations on the set of events denote the linear order of on these events. So, the relation which basically linearly order them expresses the causal dependencies among the events of pi.

(Refer Slide Time: 05:58)



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

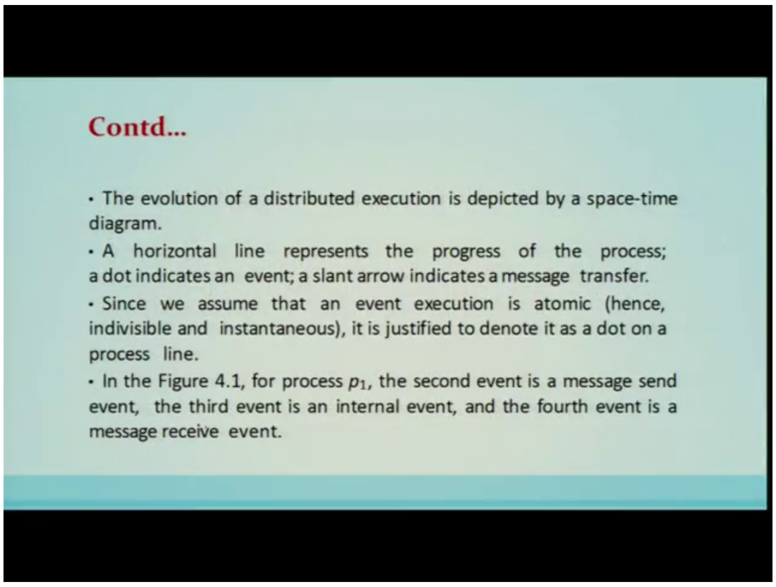
$$send(m) \rightarrow_{msg} rec(m)$$

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

The send and receive event signify the flow of flow information between processes and establish causal dependency between from sender process to the receiver process.

The relation which is denoted for the message a transmission that captures the causal dependency during the message exchange is defined as follows. For each message m that is exchanged between two processes, we have send m proceeded by the receive m message. So, the relation which establishes between send and receive event denotes the causal dependency between the pair of corresponding send and event send and receive events.

(Refer Slide Time: 06:45)



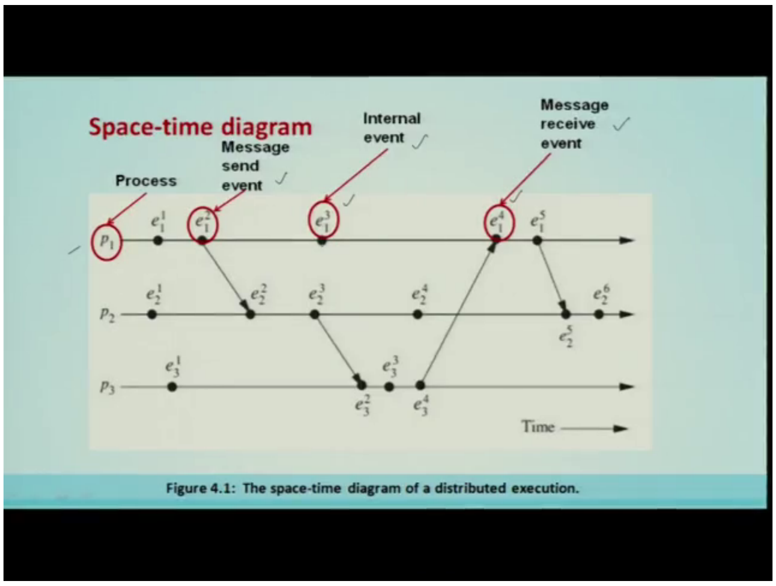
Contd...

- 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.

Now, the evolution of distributed execution is depicted by a space time diagram, the horizontal line in this space time diagram represents the progress of the process and the dot represents the event and a slant arrow indicates the message transfer between two processes.

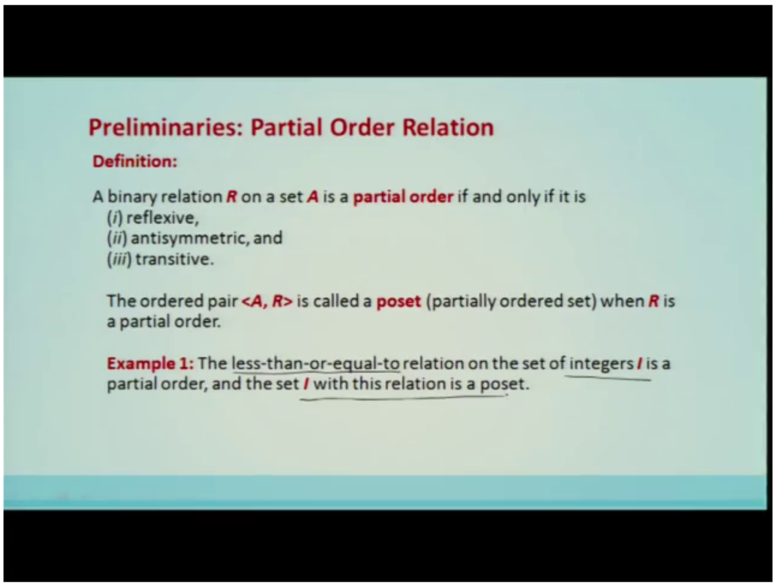
Since, we have assumed that a event execution is atomic, that is individual and instantaneous it is justified to denote it as dot on the process line.

(Refer Slide Time: 07:26)



In figure for a process p 1 the second event is the message send event third event is the internal event and forth event is the message receive event and here you can see in this particular illustrative diagram that for a process p 1 the event e 2 will be the message send event, because it is beginning with the slanted arrow over here. The message three of a process 1 indicates the internal event and the event number 4 of a process one indicates the message receive event why; because the slanted message arrow is heading towards this particular dots and dots are basically the events and they are atomic events.

(Refer Slide Time: 08:07)



Preliminaries: Partial Order Relation

Definition:

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

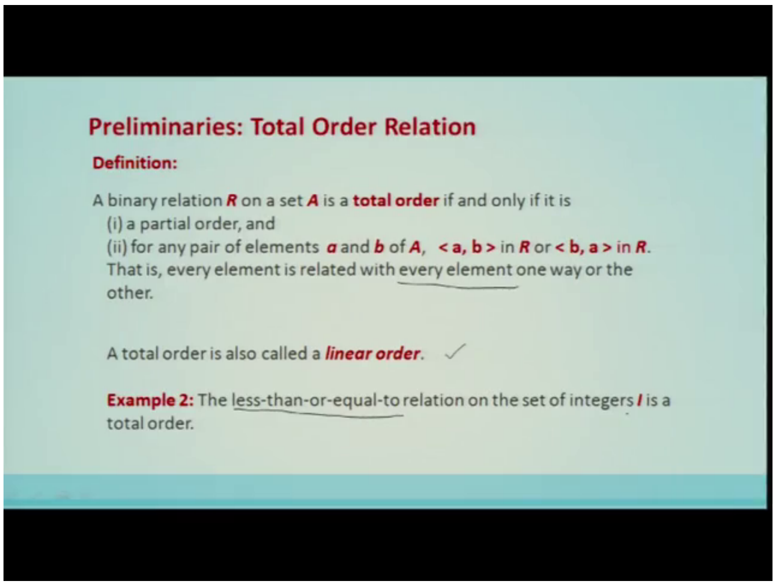
(*i*) reflexive,
(*ii*) antisymmetric, and
(*iii*) transitive.

The ordered pair **<*A*, *R*>** 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 ***I*** is a partial order, and the set ***I*** with this relation is a poset.

Now, one preliminaries which we are going to basically use in the further discussion. So, let me brief about that that is called partial order relation. So, the definition of a partial order relation goes like this, a binary relation are on a set A is a partial order if and only if it is reflexive anti symmetric and transitive. The ordered pair A and R is called poset or partially ordered set where R is a partial order. So, example here is if the relation less than or equal to on a set of integers I will form a partial order and the set I and the relation r is a poset relation.

(Refer Slide Time: 08:52)



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***, **< a, b >** in ***R*** or **< b, a >** in ***R***.

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

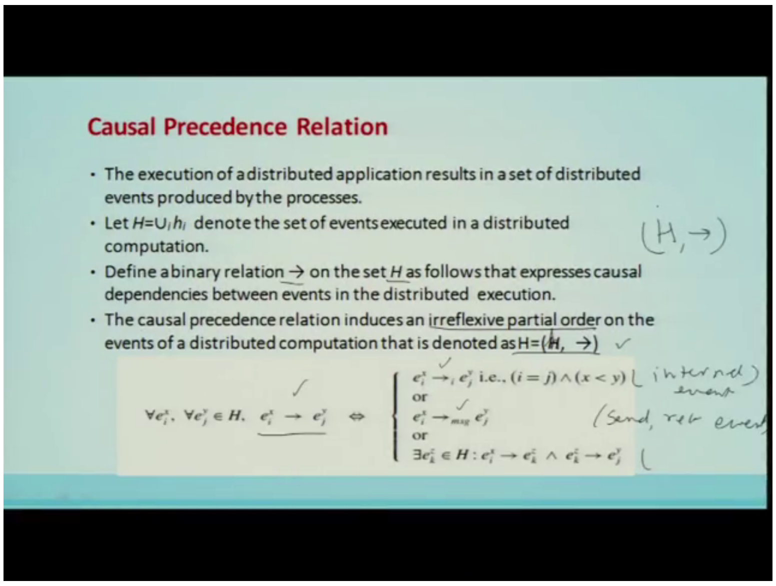
A total order is also called a ***linear order***. ✓

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

Another preliminary and definition of a total order relation a binary relation R on A set a total order if and only if it is partial order and for any pair of elements a and b of A a b pair in R R b a pair in R exist that is every element is related with every element on one way or the other then it is basically both conditions satisfied then it is called a total order.

So, total order is also called a linear order example of total order is the less than or equal to relation, on a set of integer is basically the total order.

(Refer Slide Time: 09:35)



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 → 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, →) ✓

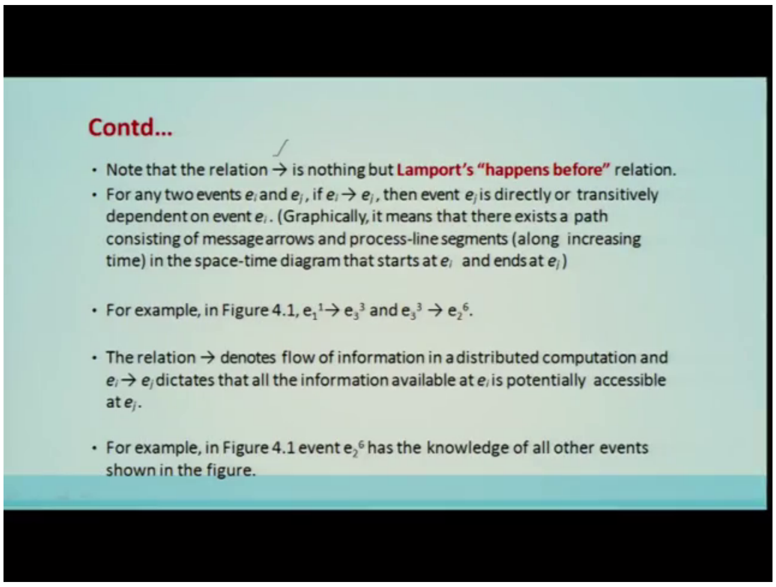
$$\forall e_i^x, \forall e_j^y \in H, \quad \overset{\checkmark}{\underline{e_i^x \rightarrow e_j^y}} \Leftrightarrow \begin{cases} e_i^x \overset{\checkmark}{\rightarrow_i} e_j^y \text{ i.e., } (i=j) \wedge (x<y) & \text{(internal event)} \\ \text{or} \\ e_i^x \overset{\checkmark}{\rightarrow_{msg}} e_j^y & \text{(Send, rev event} \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y & (\end{cases}$$

Now with these two definitions we are going to define the causal precedence relation the execution of a distributed application results in a set of distributed events produced by the processes. Let capital H be the union of all the events h is denote the set of events executed in the distributed computation.

Now, let us define a binary relation which is shown as an arrow over here on this particular set H as follows that expresses causal dependencies between the events in a distributed execution. The causal precedence relation induces and irreflexive partial order, on the events of a distributed computation that is denoted by capital H and this is nothing, but capital H is small h followed by n followed by a binary relation that is a pair. So, this will be described as. So, let e i and e j. So, e i precedes e j this by implies that e i and e j they are related with this particular in by the event that is if I is equal to j and x is less than j that is call internal event or the events which are connected or related causally using by sending the message.

Let us say e i of x preceded by e i of j using the relation which is established using the message exchange. So, with this is a message send e i is a message send and e j is message received. So, this is a send and receive event or there is a transitive relation also establishes the presidents causal precedence relation e i happened before e k and e k happened before e j. So, that indicates that e i has happened before e j. So, there are 3 kind of causal precedence relation which is which induces an irreflexive partial order on the set of events, which are happening in a distributed system which is represented by capital H and that relation together will basically denote the irreflexive partial order in a distributed system.

(Refer Slide Time: 12:20)

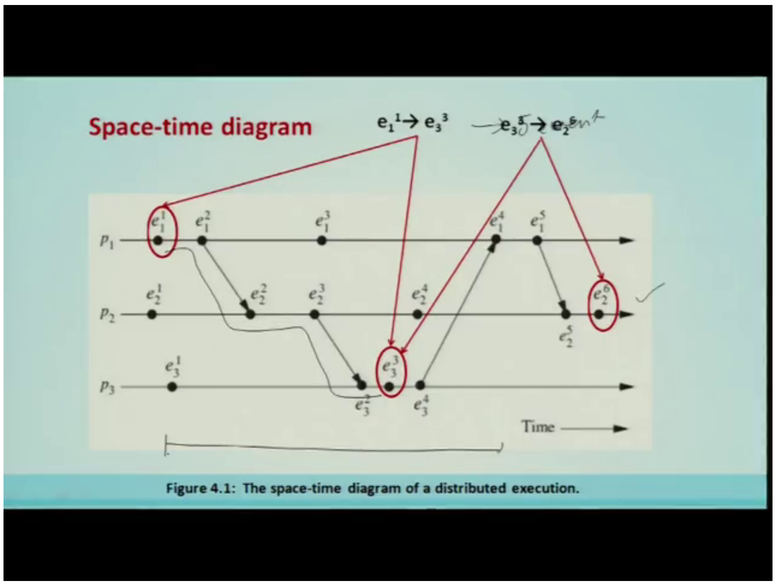


Contd...

- Note that the relation → 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 → 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.

And this basically establishes the causal precedence relation. Note that the relation arrow or the this binary relation is nothing, but a Lamport's happen before relation for any two events e i and e j if they are connected by this relation, e is happened before e j then even e j is directly or transitively dependent on e i. Graphically it means that there exist a path consisting of a message arrows and process in line segments along increasing the time on in the space time diagram that it starts at e i and ends at e j.

(Refer Slide Time: 12:55)



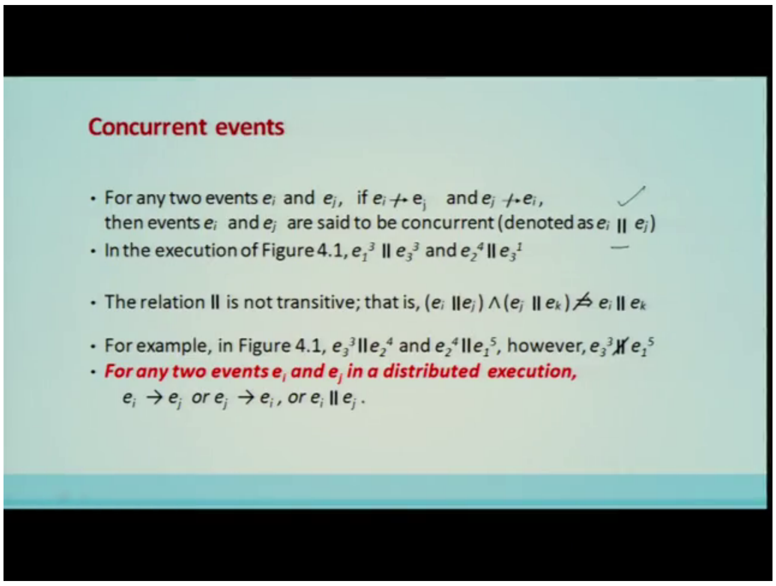
Space-time diagram

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

So, for example, in the figure e 1 and e 3; so e 1 has happened before e 3 and this basically this particular relation is basically you can see is established using a path and this particular path 1, 2, 3, 4, 5; that means, this is resulting out of the 5 different events to establish this causal residence relation of e 1 1 and e 33.

The binary relation or the causal precedence relation or a happened before relation all 3 things are same, denotes the flow of information in a distributed computation and e i has happened before e j dictates that all the information available at e i is potentially accessible at e j. So, here you can see that e 22 that e 26 has basically is preceded and has basically the information or accessing the information of all other events, which are happened before the occurrence of e 26 in this particular diagram you can see.

(Refer Slide Time: 14:17)



Concurrent events

- For any two events e_i and e_j, if $e_i \not\to e_j$ and $e_j \not\to e_i$, 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, $e_3^3 \nparallel e_1^5$
- ***For any two events e_i and e_j in a distributed execution,***
  $e_i \to e_j$ or $e_j \to e_i$, or $e_i \parallel e_j$.

Now, then concurrent events for any two events e i and e j, if e i has not happened before e j and e j also has not been happened before e i then events e i and e j are said to be a concurrent event and they are denoted by a parallel bar.

(Refer Slide Time: 14:42)

Space-time diagram

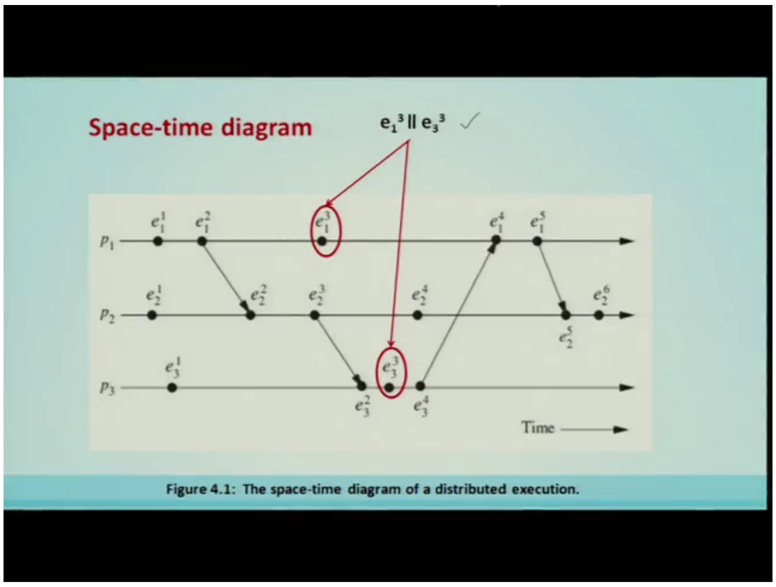
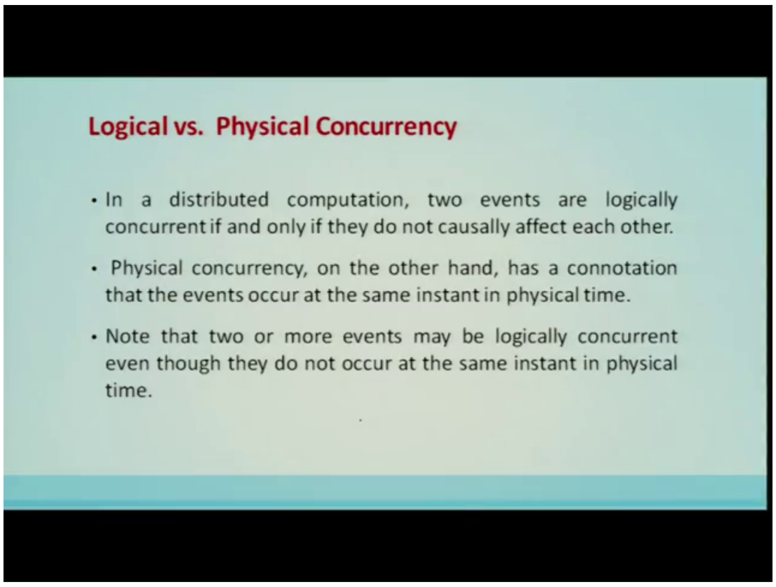


Figure 4.1: The space-time diagram of a distributed execution.

So, in the execution e 1 3 and e 3 3 they are basically the parallel y because e 1 3 has not happened before e 3 3 nor e 3 three has happened before e 1 3. So, there is no causal precedence relation hence they are called concurrent event and represented using a vertical bars. Now the relation concurrent event which is shown by the vertical bar is non transitive that is e i and e j they are related with a concurrent event and e j and e k they are related with the concurrent event that does not imply that e i and e k they are related with the concurrent event.

Now, for any two events e i and e j in a distributed system these 3 different relation, holds either e i is happened before e j or e j has happened before e i or e i and e j they are concurrent event.

(Refer Slide Time: 15:49)

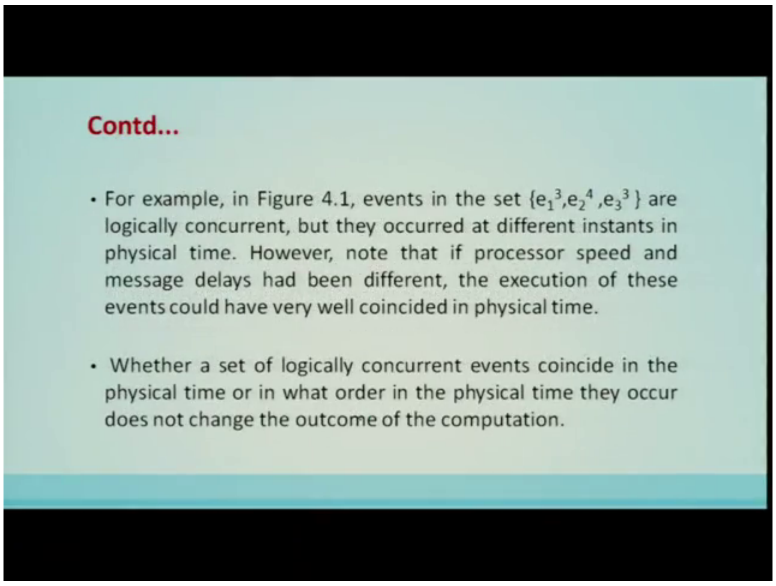


Logical vs. Physical Concurrency

- 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.

Now, logical versus physical concurrency in a distributed computation two events are logically concurrent, if they do not causally affect each other physical concurrency. On the other hand has connotation that events occur at the same instant in the physical time, note that the two or more events may be logically concurrent even though they do not occur at the same instance in the physical time.

(Refer Slide Time: 16:21)

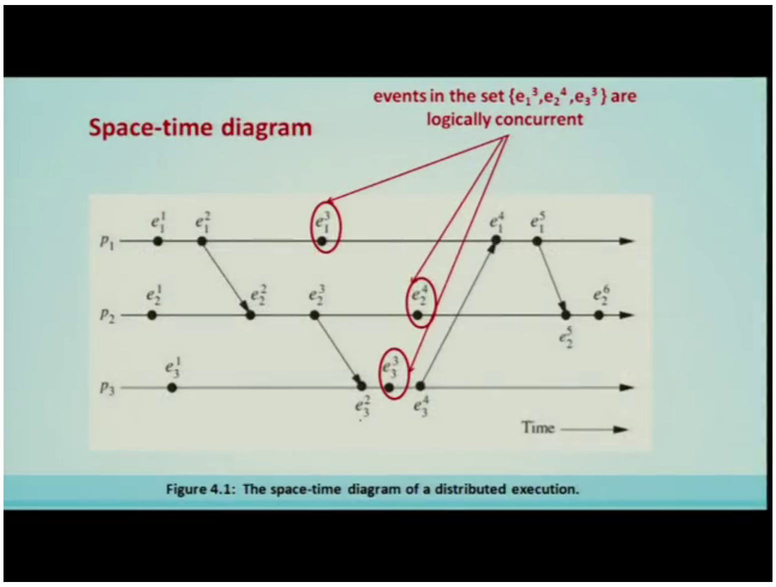


Contd...

- 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.

For example in figure 4.1 events in the set e 1, 3 and e 2 4 and e 3 3 are logically concurrent, but they occurred at different instant in the physical time; however, note that If the processor speeds and the message delays had been different the execution of these events could have very well coincided in the physical time.

(Refer Slide Time: 16:47)

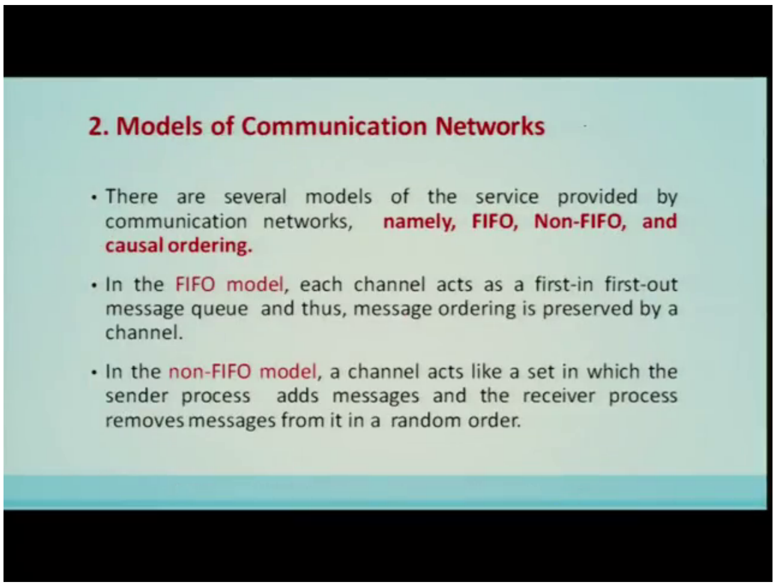


Space-time diagram

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

So, although they are logically concurrent, but physical time is a different way of expressing it. So, here even if they incidentally get the same physical time or not does not makes any difference. So, both are basically representing the logical concurrent events. So, whether I set up logically concurrent event coincide in a physical time or in what order in the physical time they occur does not change the outcome of the computation.

(Refer Slide Time: 17:35)

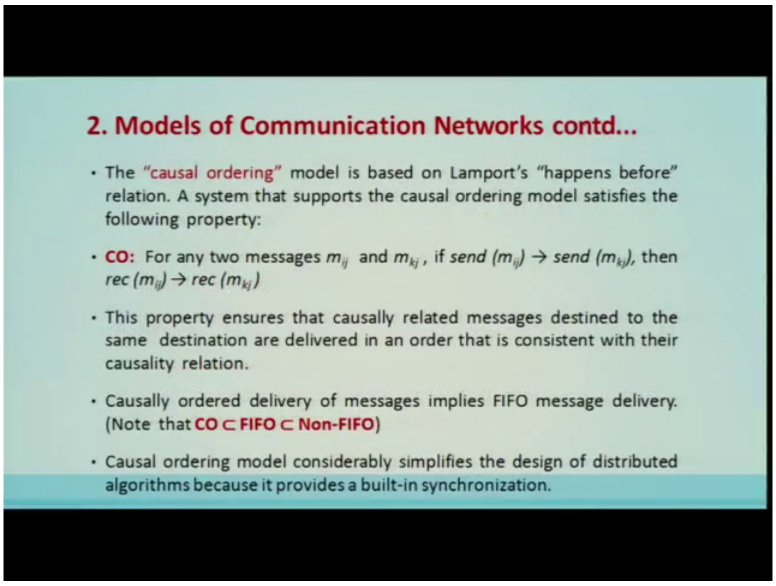


2. Models of Communication Networks

- 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.

Now another model of a distributed computation that is the model for commutation network, briefly we are going to discuss this. So, there are several models of service provided by the communication network namely FIFO model non-FIFO model and causal ordering model in FIFO model each channel acts as a first in first out message queue. Thus the message ordering is preserved by the channel itself in the non-FIFO model.

(Refer Slide Time: 18:16)



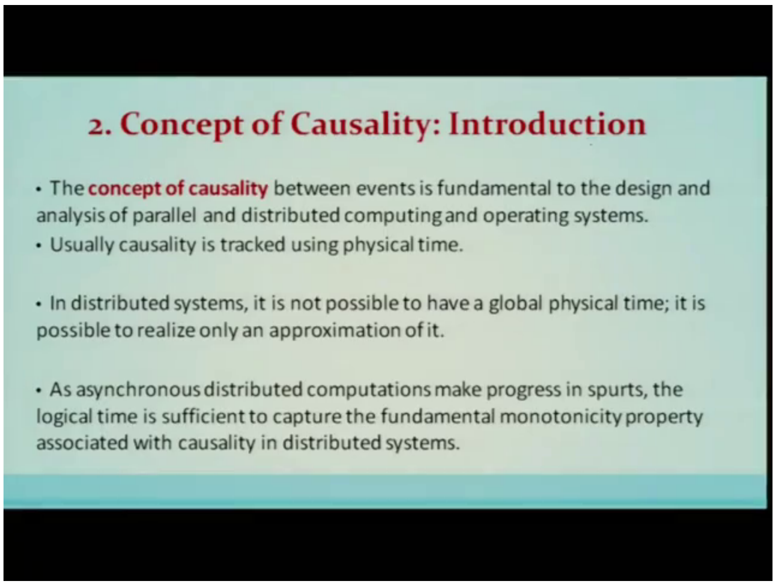
2. Models of Communication Networks contd...

- 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 ⊂ FIFO ⊂ Non-FIFO**)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

The channel acts like a set in which the sender adds the message and the receiver removes the message from it in a random order, and causal ordering model is based on Lamperts happenes before relation a system that supports the causal ordering model satisfies the following condition. Causal order for any two message m ij and m kj which are going to the same destination, if send of m ij is happened before send of m kj then receive of m ij causally related and happen before receive of m kj. This property ensures that causally related message such destiny destined to the same destination are delivered in an order that is consistent with their causality relation.

So, causally ordered delivery of the messages implies FIFO message delivery. So, causal ordering model considerably implies or signifies the design of distributed algorithm because it provides inbuilt synchronization and is used in various applications causality and logical time.

(Refer Slide Time: 19:29)



2. Concept of Causality: Introduction

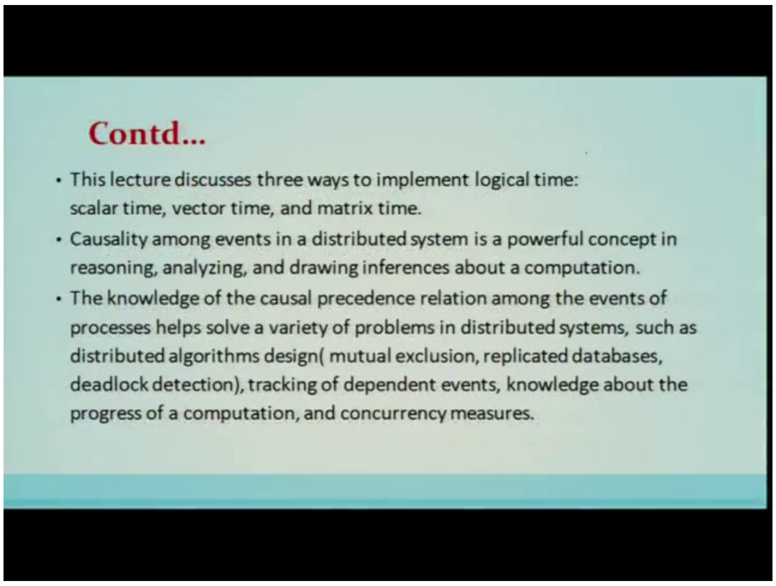
- The **concept of causality** between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have a global physical time; it is possible to realize 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.

The concept of causality the concept of causality between the events is fundamental to the design and analysis of parallel and distributed computing and operating system usually causality is tracked using physical time like we are doing in our daily life for example, a queue for purchasing a ticket are basically a line of people standing for arrival of a bus and so on and so forth.

So, they are all tracked using the physical time. So, whosoever has come first, come first means the causality of coming and joining the queue and that is basically is done or is being tracked using physical time. With this explanation we are now going to see how we are going to use it causality in the distributed system. In distributive system it is not possible to have a global physical time; it is possible to realize only approximation of it. As a synchronous distributed computation make progress in his parts the logical time is sufficient to capture the fundamental monotonicity property associated with causality in a distributed system.

So, meaning to say that though we do not have the global physical time yet we are going to use an approximation of it because that is what only is required to capture the causality in distributed systems by between the events.

(Refer Slide Time: 21:13)



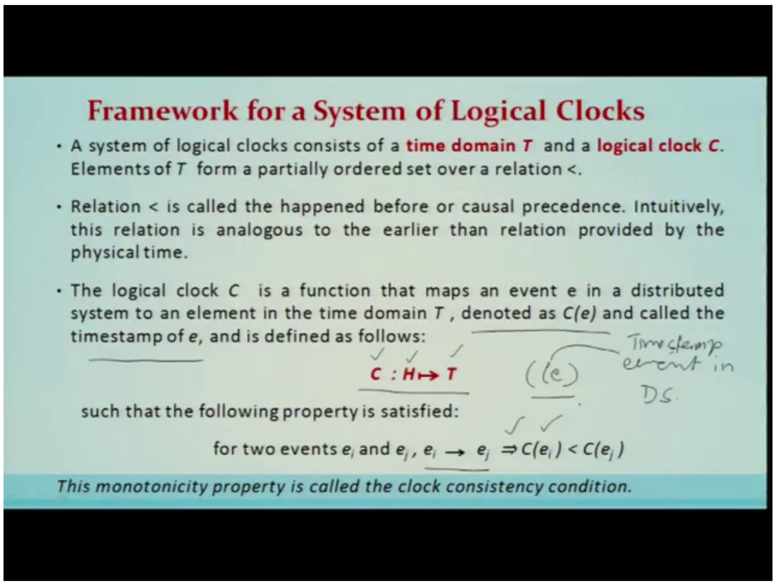
Contd...

- 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.

So, this lecture discusses 3 ways to implement the logical time, which is an approximation of a global physical time and which will capture the causality of events in the distributed system without having the common physical clock. So, there are 3 different ways to implement the logical time we are going to discuss in this part of the lecture they are the scalar time, vector time and matrix time.

Now, causality among the events in a distributed system is a powerful concept in reasoning analyzing and drawing inferences about the computation. The knowledge of causal precedents relation among the events of processes helps solve various problems in a distributed system such as when we see the distributed algorithm design for mutual exclusion, there you can use there you can see that for fairness we are going to use this particular time concept. Similarly in a replicated database databases this time concept is used to have a consistent updates, similarly in a deadlock detection correctly using for avoiding the friend terms and other problems this particular time concept or a causality concept we are going to use it; similarly, the tracking of dependent events knowledge about the progress of a computation and concurrency measures.

(Refer Slide Time: 22:58)



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:

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

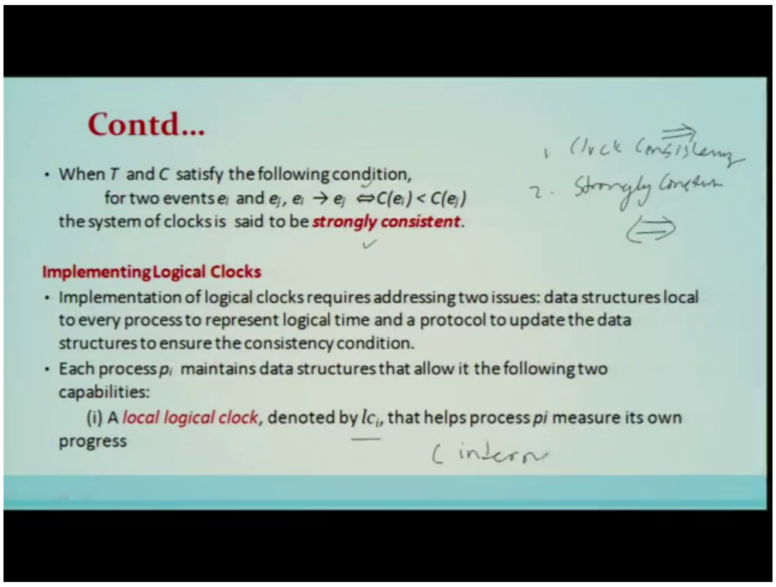
This monotonicity property is called the clock consistency condition.

So, you just see that causality is one of the important means concept in the design of distributed algorithm in the system that we are going to discuss in this course in this lecture.

Now, framework for a system of logical clocks a system of logical clocks consists of a time domain T and a logical clock C, the elements of T from a partially ordered set over a relation which is shown as less than, but it is a relation less than is called happen before or a causal precedence relation. Intuitively this relation is analogous to the earlier than relation provided by the physical time. So, now, we are correlating the properties of physical time using alternative concept that is called basically the logical time or a logical clocks. So, the logical clock C is a function that maps and even e in a distributed system to an element in a time domain T and that is denoted by C and it is also called the timestamp of an event e and is denoted by this particular function. So, C is a function which will take the events out of the distributed events which is denoted by capital H and will basically map on to a time domain and this particular function will form a time stamp that is C of e of a particular event in a distributed system.

Now, such that the following properties are satisfied; so for any two events e i and e j and if e i is happened before e j or they are related with this particular relation causal precedence relation, e i is happened before e j this will imply that the timestamp of e i is less than the timestamp of e j this monotonicity property is called the clock consistency condition.

(Refer Slide Time: 25:20)



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 lc_i, that helps process *pi* measure its own progress

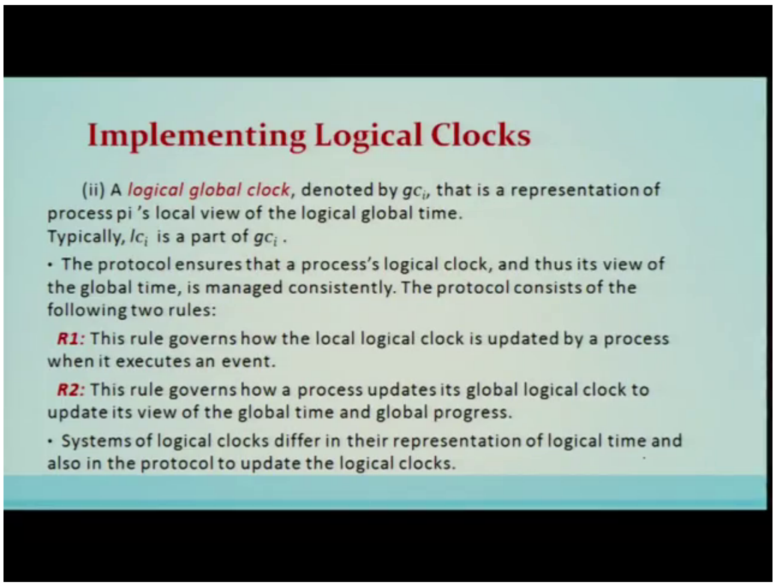
1 Clock Consistency $\Rightarrow$
2. Strongly Consistent $\Leftrightarrow$
C intern

When T and C that is the time and a clock domain satisfies the following conditions that is for any two event e i and e j, e i is happened before e j this by implies that the timestamp of e i is less than time stamp of e j or you can also say that the timestamp of e i is less than timestamp of e j this implies that e i is happened before e j, if both ways this particular relation holds then this system of clock is called strongly consistent system of clocks.

So, we have seen the two condition one is a clock consistency condition the other is called strongly consistent condition of the clocks. Here only the implication will give the timestamp if the relations are related with the happen before. Then, basically the timestamp is less than the timestamp of other events and if it is a strongly consistent then it is a by implies; that means, if the timestamps are given by that we can infer whether the two events are happened before or not that we are going to see in more detail further.

Now, implementing the logical clocks; implementation of a logical clocks requires addressing two issues first one is the data structure local to every process to represent the logical time and the protocol to update the data structure, to ensure the consistency condition. So, each process pi maintains a data structure that allows the following two capabilities the first one is called local logical clock and denoted by LCI that helps process pi measure it is own progress or you can also say that this will ensure the progress of internal events.

(Refer Slide Time: 27:27)



Implementing Logical Clocks

(ii) A *logical global clock*, denoted by gc_i, that is a representation of process pi '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.

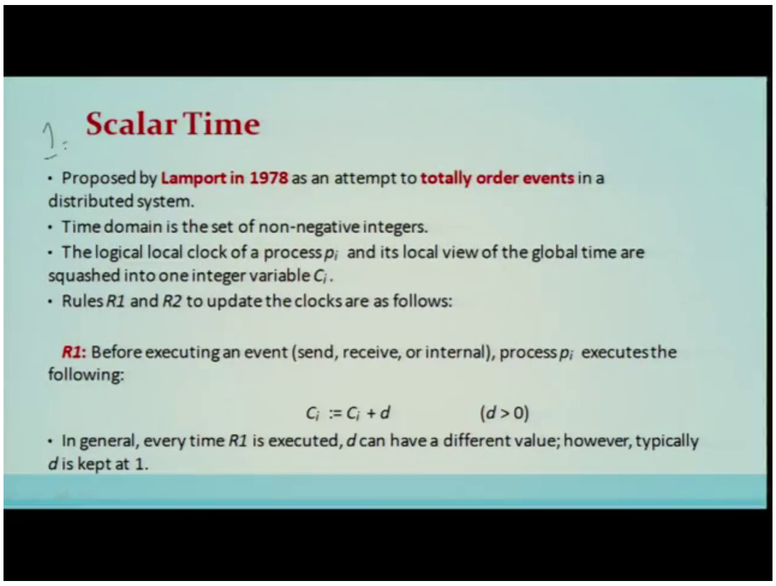
R2: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.

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

Now, second is about the protocol and it is basically called the loop the logical global clock that is gc of i that is the representation of a process pi is local view of the logical global time typically l C i is a part of gc i. So, the protocol ensures that the process is logical clock and thus it is view of the global time is managed consistently the protocol consists of the following two rules this rule governs R 1 rule governs how the local logical clock is updated by a process, when it executes an event R 2 rule this rule governs how the process updates it is global logical clock to updates it few of the global time and a global progress.

Now, the systems of logical clocks differ in their representation of the logical time and also in the protocol to update the logical clocks. So, that was the general framework.

(Refer Slide Time: 28:31)



1. **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:

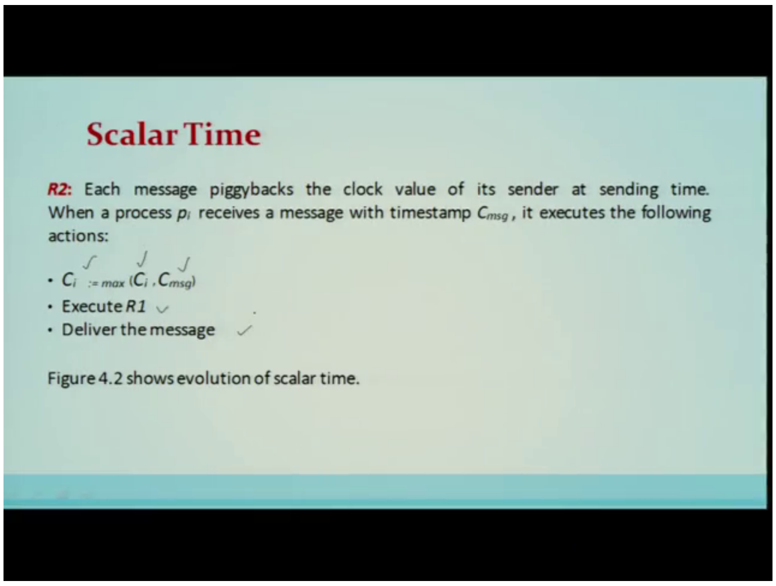
$$C_i := C_i + d \qquad (d > 0)$$

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

Now, we are going to see the first type of logical time that is called the scalar time. This particular scalar time was proposed by the Lamport in 1978 as an attempt to totally order the events in a distributed system. Here the time domain is the set of non negative integers, the logical local clock of a process p i and it is local view of a global time are squashed into one integer variable C i.

Now, the rules of implementing the protocol goes like this the rule R 1 and R 2 to update the clocks are as follows. R 1 rule before executing an event that is send receive or internal the process pi execute the following C i is equal to C i plus d, where d is greater than zero in general every time R 1 rule is executed d can have different values; however, typical d is kept as one rule R 2. So, each message piggybacks the clock value of it is sender at the sending time.

(Refer Slide Time: 29:40)



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.

Now, when a process pi receives a message with the timestamp C of message it executes the following action. So, there are 3 actions first action is, this C of message is the timestamp which basically was piggybacked and received at the particular process and C i is basically the clock value internally. So, the maximum of this will be updated in C i value then it will execute the rule one and would deliver the message. So, the figure 4.2 shows illusion of the scalar time.

(Refer Slide Time: 30:20)

Evolution of Scalar Time

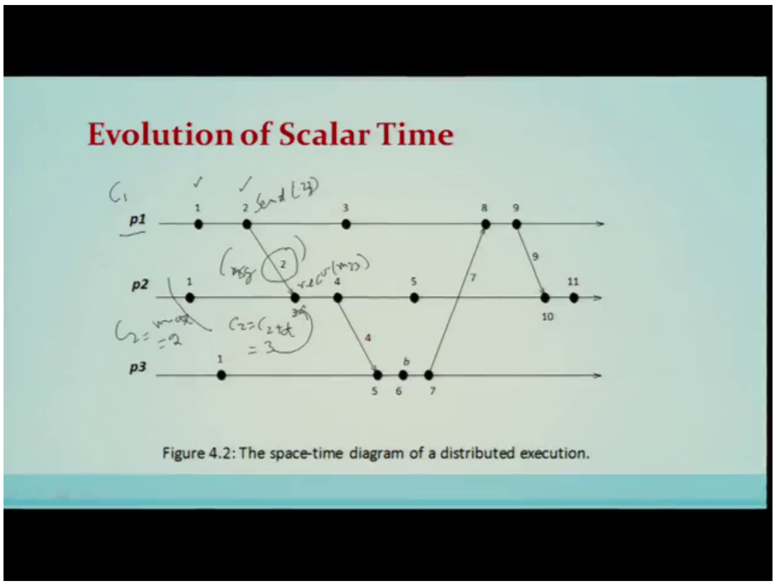
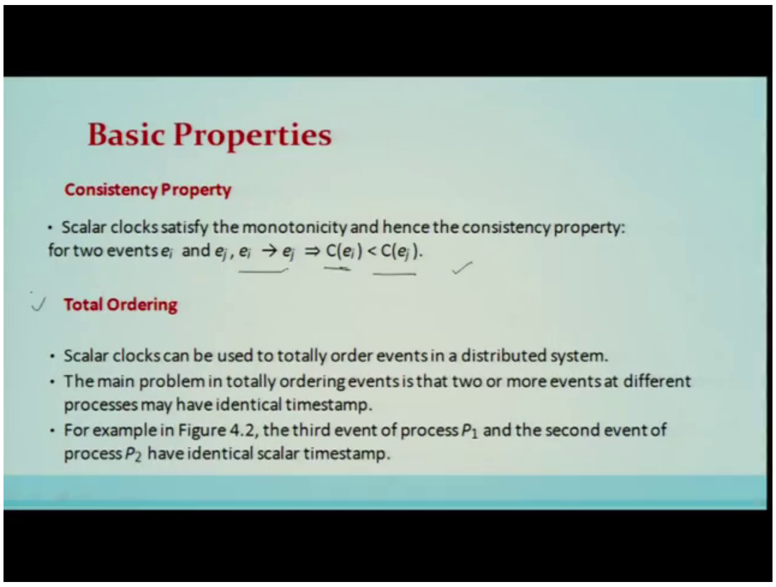


Figure 4.2: The space-time diagram of a distributed execution.

So, let me explain through this particular scalar time now p 1 the first event that is basically it has the clock C 1. So, using C 1 the event was timestamp as one now event two was time stamped as two here for the process p 1. Now event two is a send of a message. So, this particular timestamp will be piggybacked on the message and it will when it will arrive over here. So, it will take the maximum of maximum of this particular event C 2which is 1 and C message that is 2 that becomes 2 and then it will apply the rule R 1 rule R 1 says that this C 2 is equal to C 2 plus d and d is equal to 1. So, this becomes 3.

So, 3 will be time stamped on this particular event which is will be the receive of event receive of a message.

(Refer Slide Time: 31:54)



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

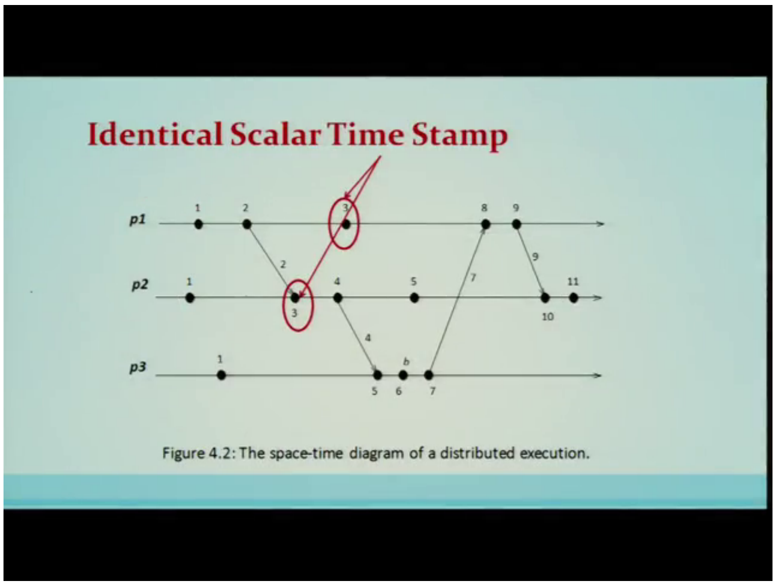
- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure 4.2, the third event of process P_1 and the second event of process P_2 have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

This particular scalar time follows following basic properties; the first property is called consistency property. Scalar clock satisfies the monotonicity property and hence the consistency property that is for any to event e i and e j if e i is happened before e j, this will imply that the timestamp of e i is less than the timestamp of e j. That means, in our normal situation also if I event has happened before the other event or if somebody is standing in the front of the queue.

The next person who comes afterwards his time will be more than the person who is standing in the front of the queue or who arrives before him. So, using this concept also here you can basically see that this particular events happening and the timestamp of the clock will hold us this particular relation that is less than relation.

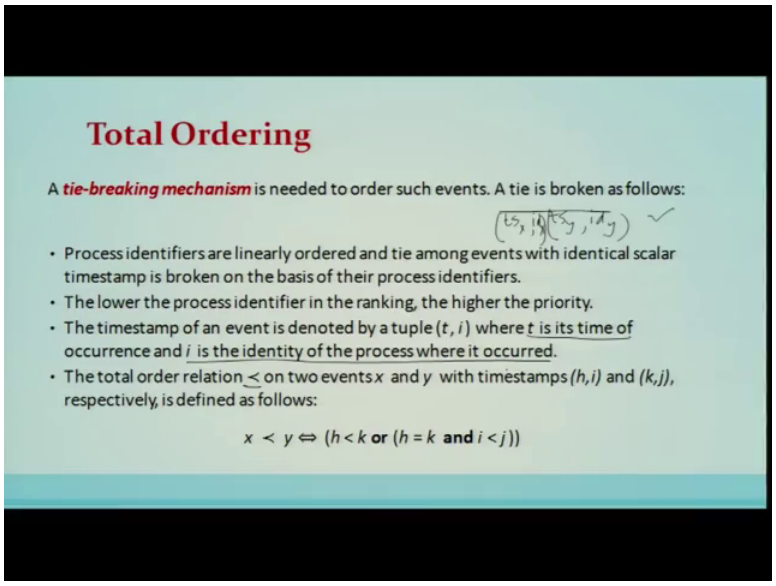
Now the next property which will be satisfied by the scalar time is called total ordering scalar clocks can be used to total order the events in a distributed system, the main problem in total ordering events is that two or more events at different processes may have identical timestamp because these clocks at different processes that is nothing, but the variable this non negative or integer variable will be incremented independently.

(Refer Slide Time: 33:39)


Identical Scalar Time Stamp
p1
p2
p3
1
2
3
8
9
1
3
4
5
10
11
1
5
6
7
2
4
7
9
Figure 4.2: The space-time diagram of a distributed execution.

So, for example, in a figure 4.2 third event of a process p 1 and the second event of process to have identical scalar timestamp here, how we are going to order these events than in that case or total order.

(Refer Slide Time: 33:48)


Total Ordering

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

$(ts, i)(ts_j, id_j)$ ✓

- 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 $\prec$ on two events x and y with timestamps (h,i) and (k,j), respectively, is defined as follows:

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

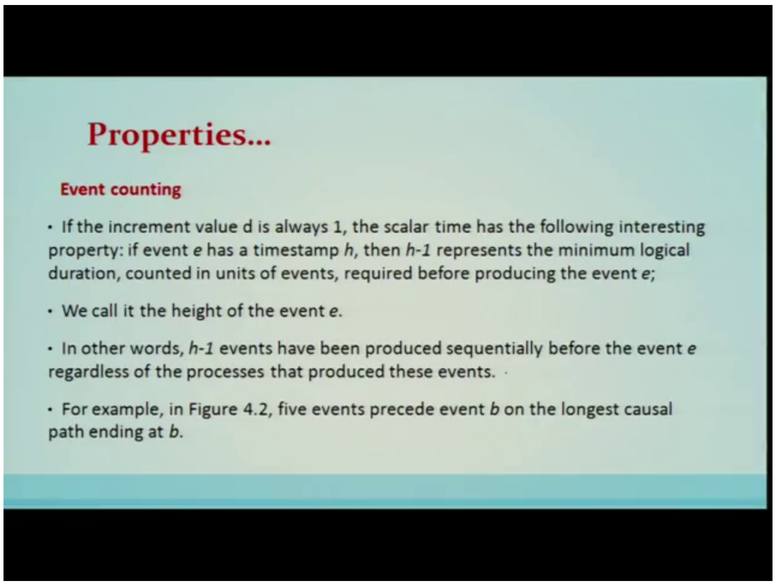
So, to do the total ordering in this particular case where by scalar time is the same a tiebreaking mechanism is needed to order such events a tie is broken as follows. The process identifiers are linearly ordered and the tie among the events with identically scalar timestamp is broken on the basis of their process ids.

So, meaning to say that if they have the same timestamp of x and timestamp of y, but their ids of x and id of y will be used to order them in case the timestamp are same. The lower the process ids in the ranking the higher the priority, the timestamp of an event is denoted by your tuple. T comma where T is the time of occurrence and i is the identity of a process, where it occurred that I have already explained.

So, the total order in relation denoted by this symbol, total order relation on two events x and y with a timestamp h i and k j respectively is defined as follows. So, x and y they are related with the total order relation this by implies that the time stamp of x that is h is less than k; that means, either x has happened before y by the time stamp or if their time stamps are same then the id of x that is nothing, but an i is less than j and if that is happening then this will total order them.

So, tiebreaking and partial order will form the total ordering of event and this is same as the definition of a total order, we have seen in the previous slides that is it follows the partial order and also a tiebreaking mechanism in this way. Further properties are that you can use the scalar time for event counting.

(Refer Slide Time: 36:08)



Properties...

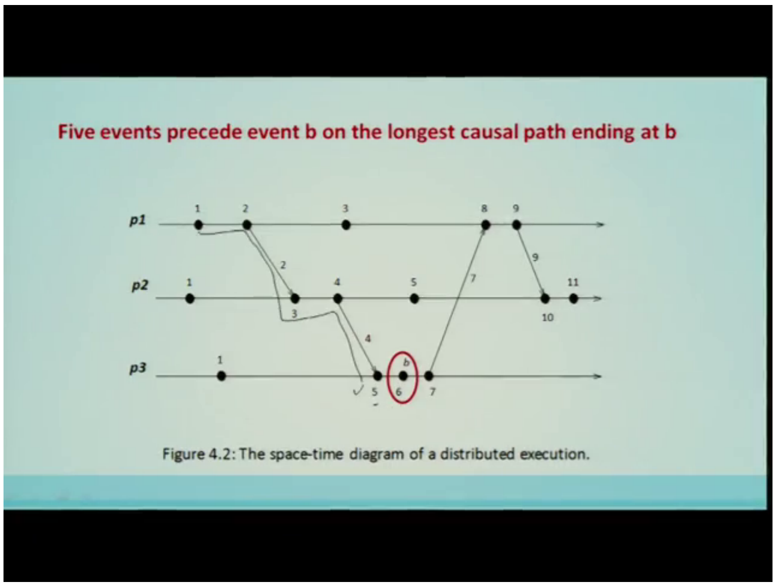
Event counting

- 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*.

If the increment value d is always 1 the scalar time has the following interesting property. That is if the event e has a timestamp h then h minus 1 represents the minimum logical duration, counted in the unit of events required before producing the event e. We call it a height of event e. In other words h minus 1 event have been produced sequentially before event e regardless of the process that produced these events.

So, we can count how many events have happened before a particular event for example: in the figure five events proceed event b on the longest causal path ending at b.

(Refer Slide Time: 36:55)

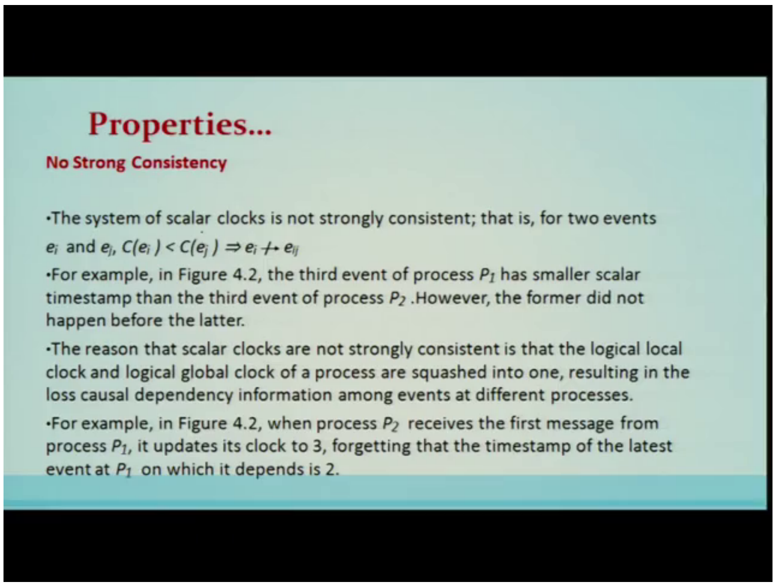


Five events precede event b on the longest causal path ending at b

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

So, in this particular figure you can see that the event b has basically to occur event b 5 different events have been preceded before it and that is why this number 5. So, here in this particular example or illustration you can see that the event b which is having the time stamp six. That means before that, five different events have happened then only this particularly event b is occurring. So, five events you can see this this this.

(Refer Slide Time: 37:34)



Properties...

No Strong Consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j, $C(e_i) < C(e_j) \Rightarrow e_i \not\rightarrow e_{ij}$
- 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.

Now, another property is no consistency property for a scalar time. So, the system of the scalar clocks is not strongly consistent, that is for any two events e and e i and d j if we compare the clock values that does not indicate that does not imply that e i is happening before e j for example, in figure 4.2 the third event of a process p 1 has a smaller still a timestamp than the third event of process 2; however, the former did not happen before the latter.

(Refer Slide Time: 38:12)

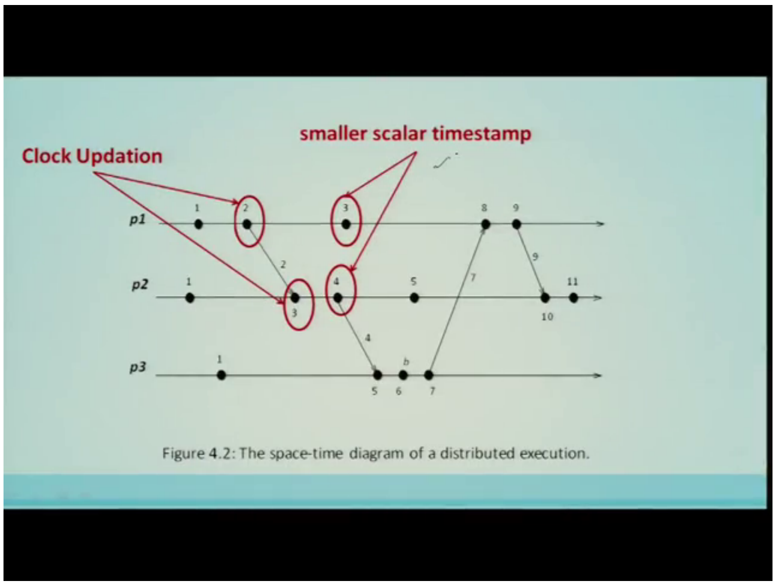
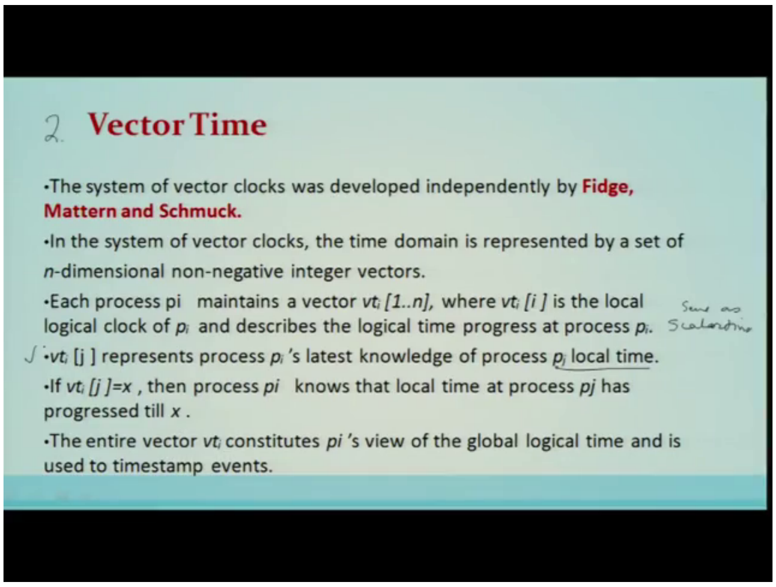


Figure 4.2: The space-time diagram of a distributed execution.

So, here in this particular example you can see that by smaller a scalar timestamp does not mean that they basically hold the relation that is happened before relation.

(Refer Slide Time: 38:24)



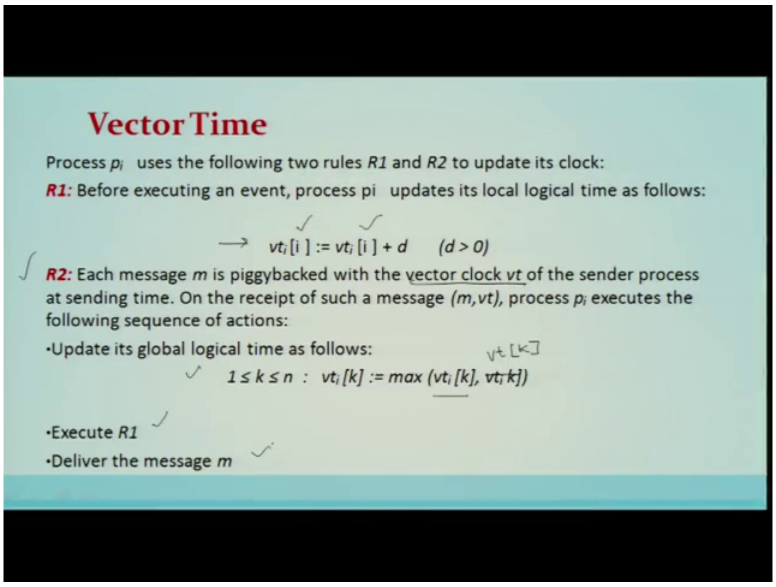
2. Vector Time

- 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 pi 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. Same as Scalar time
- ✓ vt_i [j] represents process p_i 's latest knowledge of process p_j local time.
- If vt_i *[j]=x* , then process *pi* knows that local time at process *pj* has progressed till *x* .
- The entire vector vt_i constitutes *pi* 's view of the global logical time and is used to timestamp events.

Now, with this particular discussion of a scalar time they does not have the strong consistency property, we have a motivation to how another system of clocks or a time we should have the strong consistency property because by looking at the time stamp if we can infer that the events have happened before or not. So, for those kinds of applications we require another clock or another time. So, this is the motivation to study another time which is called the vector time. So, 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.

So, each process pi maintains a vector of size n, where vt of I is the local logical clock of pi n denotes the loop the logical time progresses at pi or that is equivalent to the scalar time of I of pi vt i of j represent the process p is latest knowledge of the process pjs local time. So, pis latest knowledge of pjs local time is basically stored in the i-th indexed vector of a process i. Now if vt i of j-th index is equal to x then process p i knows that the local time at process pj has progressed till x the entire vector vi vt i constitutes pi is view of the global logical time and is used to this timestamp the events.

(Refer Slide Time: 40:38)



Vector Time

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

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

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

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:

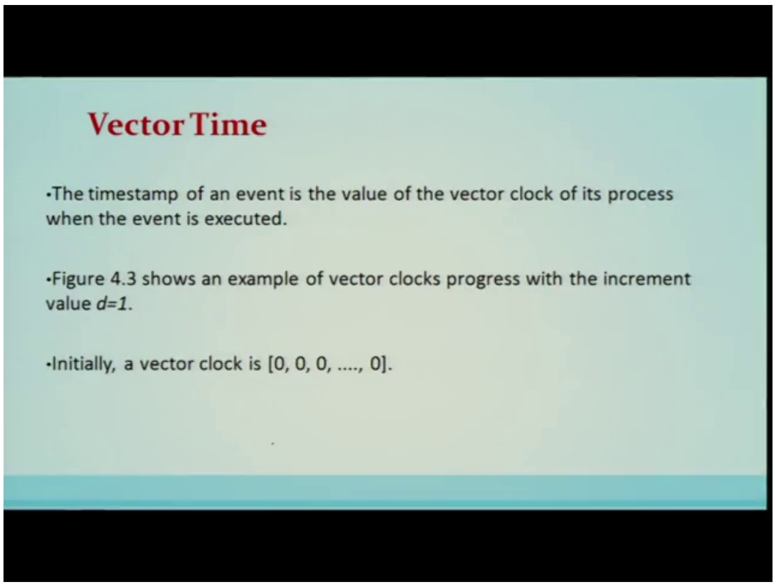
$$1 \le k \le n \;:\; vt_i[k] := \max(\underline{vt_i[k]}, \overset{vt[k]}{\cancel{vt_i\,k]}})$$

- Execute *R1*
- Deliver the message *m*

Now, the process pi used the following two rules to update it is vector clock. The rule one for the vector clock says that before executing an event process I update this local logical time just like a scalar clock that is the vector of i is assigned to vector i plus d.

So, rule 2 of a vector clock says that each message m is piggybacked with the vector clock vt of a sender process at the sending time. On the received of such message m comma vt the process pi execute the following actions, the first action says that update is global logical time as follows. So, it will take it is vector time and also the vector time which is basically piggyback here in the message and the maximum of that will be updated so; that means, it will update the global logical time according to this particular formula and once having done that then it will execute R 1 and it will deliver the message.

(Refer Slide Time: 42:22)

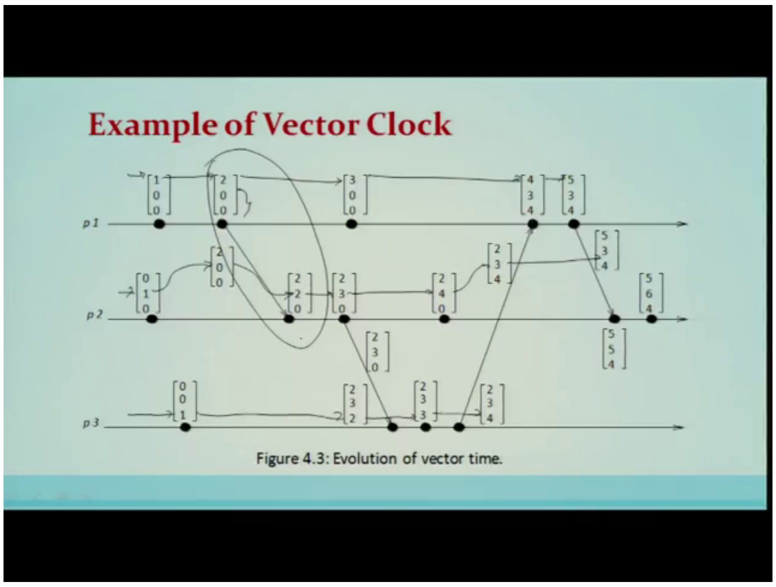


Vector Time

- 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, ..., 0].

So, the timestamp of an event is the value of the vector clock of it is process when the event is executed.

(Refer Slide Time: 42:46)



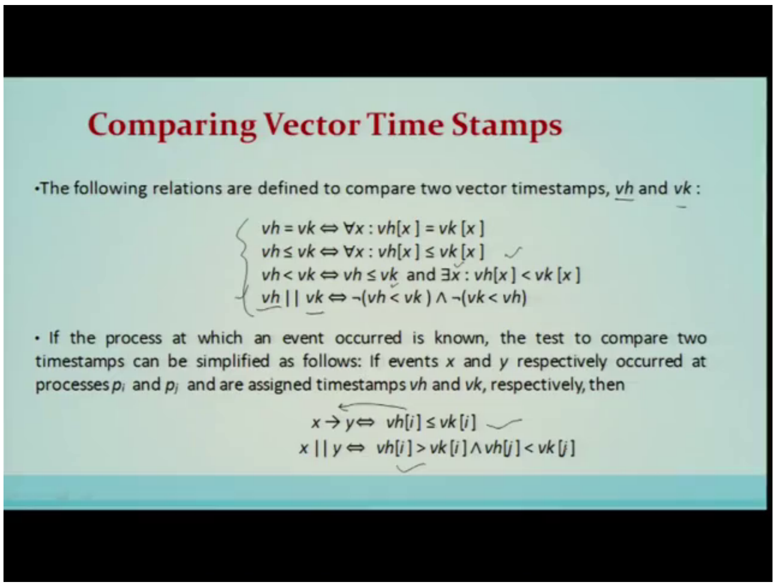
Example of Vector Clock

Figure 4.3: Evolution of vector time.

Now, figure 4.3 shows an example of vector clock progress with the increment value d is equal to 1, initially a vector clock is all zeroes initialized. So, the example of a vector clock you can see here in this is space time diagram. So, in the vector indexed 1 it is same as by scalar clock you see that it is growing in a linear manner.

Similarly as far as for p 2 the second indexed of a vector time is growing linearly. And similarly the third vector for p 3 is also growing linearly in the vector timing. And whenever there is a send of a message or whenever there is a message exchange these global views will be updated at the other end of a process.

(Refer Slide Time: 43:54)



Comparing Vector Time Stamps

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

$$\begin{aligned} vh = vk &\Leftrightarrow \forall x : vh[x] = vk[x] \\ vh \leq vk &\Leftrightarrow \forall x : vh[x] \leq vk[x] \\ vh < vk &\Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x] \\ vh \,||\, vk &\Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh) \end{aligned}$$

- 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

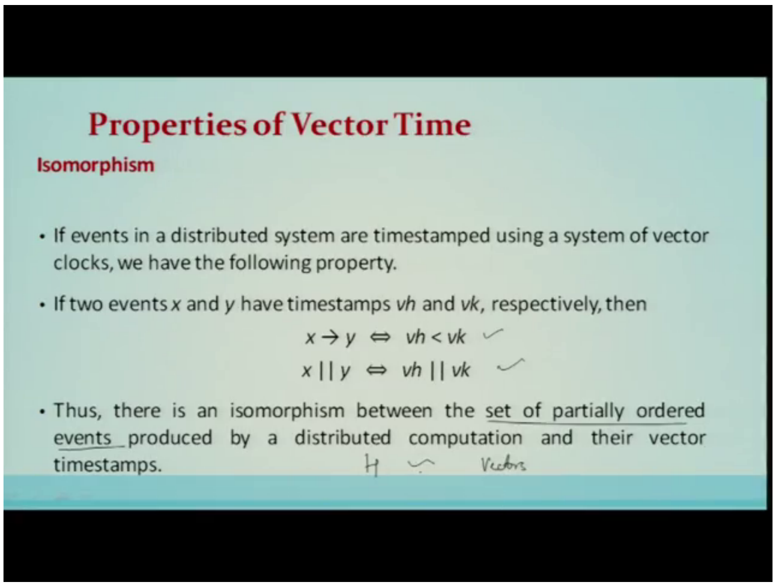
$$\begin{aligned} x \rightarrow y &\Leftrightarrow vh[i] \leq vk[i] \\ x \,||\, y &\Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j] \end{aligned}$$

Now, comparing the vector timestamps the following relations are defined to compute to compare the two vector timestamps vh and vk. So, these are the operations by which we can compare the timestamps and infer the causal precedence relation between these events. Now if the time stamps vh and vk they are same this by implies that for all values of x v h of x is equal to v k of x, vh if it is less than or equal to vk this by implies that for all values of x. Vh of x is less than or equal to v k of x v h of v h is strictly less than vk this by implies that vh is less than or equal to vk and there exists an x where v h of x is strictly less than x vk of x. Vh and vk that is the vector timestamps they are concurrent this by implies that v h is not less than vk and also not vk is less than vh then they are concurrent events.

So, these set of comparative relations are used to compare the timestamps of between two events and we can infer the causal precedence relation between these two events. So, if the process at which the event occurred is known the test to compare the two timestamp can be simplified as follows. So, if events x and y respectively occurred at processes p i n pj and are assigned time stamp vh and vk respectively then x has happened before y this by implies that v h of i is less than or equal to vk of i. So, this is a very important understanding or if let us say v h of i is less than or equal to v k of I then it will also indicate that x has happened before y.

Similarly, x and y they are concurrent events if their vector time stamps are basically if vh of I is greater than v k of I and v h of I is to strictly less than the vk of I then they are concurrent events.

(Refer Slide Time: 46:37)



Properties of Vector Time

Isomorphism

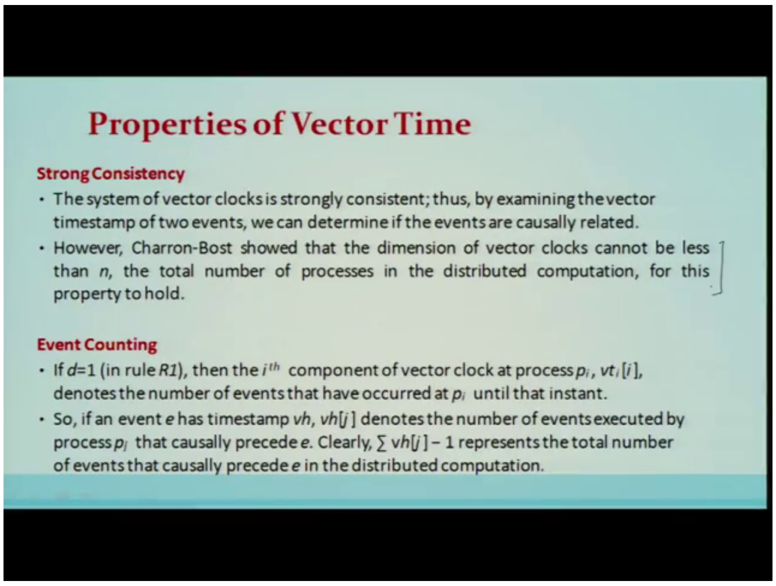
- If events in a distributed system are timestamped using a system of vector clocks, we have the following property.
- If two events *x* and *y* have timestamps *vh* and *vk*, respectively, then

$$x \rightarrow y \Leftrightarrow vh < vk \quad ✓$$
$$x \parallel y \Leftrightarrow vh \parallel vk \quad ✓$$

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

So, another property of a vector clock is isomorphism. So, if the events in a distributed system are time stamp using a system of vector clock we have the following properties. So, if the two events x and y have time stamp vh and vk respectively, then x is happened before y this will be indicated using v h is less than vk if x and y they are parallel; that means, their vector timestamp also are basically un comparable thus there is isomorphism between the set of partially ordered events produced by the distributed computation that is h and their vector timestamp. So, they are basically holding the isomorphism property.

(Refer Slide Time: 47:28)



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

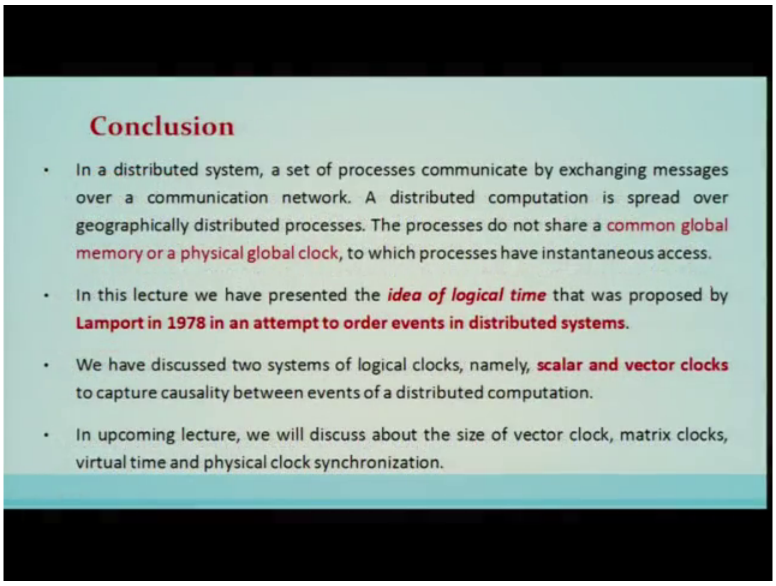
- If d=1 (in rule $R1$), then the i^{th} component of vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant.
- So, if an event e has timestamp vh, $vh[j]$ denotes the number of events executed by process p_j that causally precede e. Clearly, $\sum vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation.

The vector time follows by strong consistency property the system of vector clocks is strongly consistent that is by examining the vector timestamp of the two events we can determine if the events are causally related. However, Charron and Bost showed that the dimension of a vector clocks cannot be less than n, n is the total number of processes in the distributed computation for this particular property to hold this particular discussion we will see in the next class in more details.

Now another property of a vector clock is the event counting when d is equal to one in rule one then i-th component of the vector clock at a process i that is v T of i of i denotes the number of event that have been that have occurred at pi until that particular time. So, we can do the event counting just like we have seen in the scalar clock.

So, if an event e has a timestamp v h. So, vhf i indicates the number of events executed by pj that causally precede e clearly.

(Refer Slide Time: 48:48)



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.

If we take the summation of vh of j minus one represent the total number of events that causally precede e in the distributed computation. Conclusion: in the distributed system a set of processes they communicate by exchanging messages over the communication network the distributed computation is spread geographically over the distributed processes. The processes do not share a common global memory or a common physical clock to which the process have instantaneous access.

Thus, we have seen in this particular lecture how to overcome from these particular difficulties of this particular model which is called a distributed system. So, instead of having the physical clock we have seen that the logical clock which will give will capture the causal relations between different distributed events. So, in this lecture we have presented the idea of logical clock, which is going which captures the causal relation between the event that was proposed by the Lamport in 1978 in an attempt to order the events in a distributed system. We have discussed two system of logical clock in this lecture namely the scalar and vector clocks to capture the causality between the events in a distributed computation, holding the strong consistency property and clock consistency property.

So, in the upcoming lectures we will discuss about the size of the vector clock, matrix clock and the virtual time and a physical clock synchronization; all these important aspects which is basically giving you the concept of causality in a distributed system, which is the most fundamental in the design of the distributed system.

Thank you.