Distributed Systems

Dr. Y. V. Lokeswari
ASP / CSE
SSN College of Engineering

Agenda

- Introduction to Distributed Systems
- Characteristics of Distributed Systems
- Distributed System Model
- Motivation for Distributed Systems

Introduction to Distributed Systems

- A distributed system is a collection of **independent entities that cooperate** to **solve a problem** that **cannot be individually solved**.
- **Definition**: Distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.
- Computers are **semi-autonomous and are loosely coupled** while they cooperate to address a problem collectively.

Introduction to Distributed Systems

- Autonomous processors communicating over a communication network.
- **Some characteristics**
  - ❖ No common physical clock
  - ❖ No shared memory
  - ❖ Communicate by a messages passing over a communication network.
  - ❖ Each computer has its own memory and runs its own operating system.
  - ❖ Geographical separation
  - ❖ Autonomy and heterogeneity
  - ❖ Independent Failure is natural in Distributed Systems
    - o Faults in the network result in the isolation of the computers.
    - o Failure of a computer is not immediately made known to the other components with which it communicates.
  - ❖ Concurrent program execution

Distributed System Model

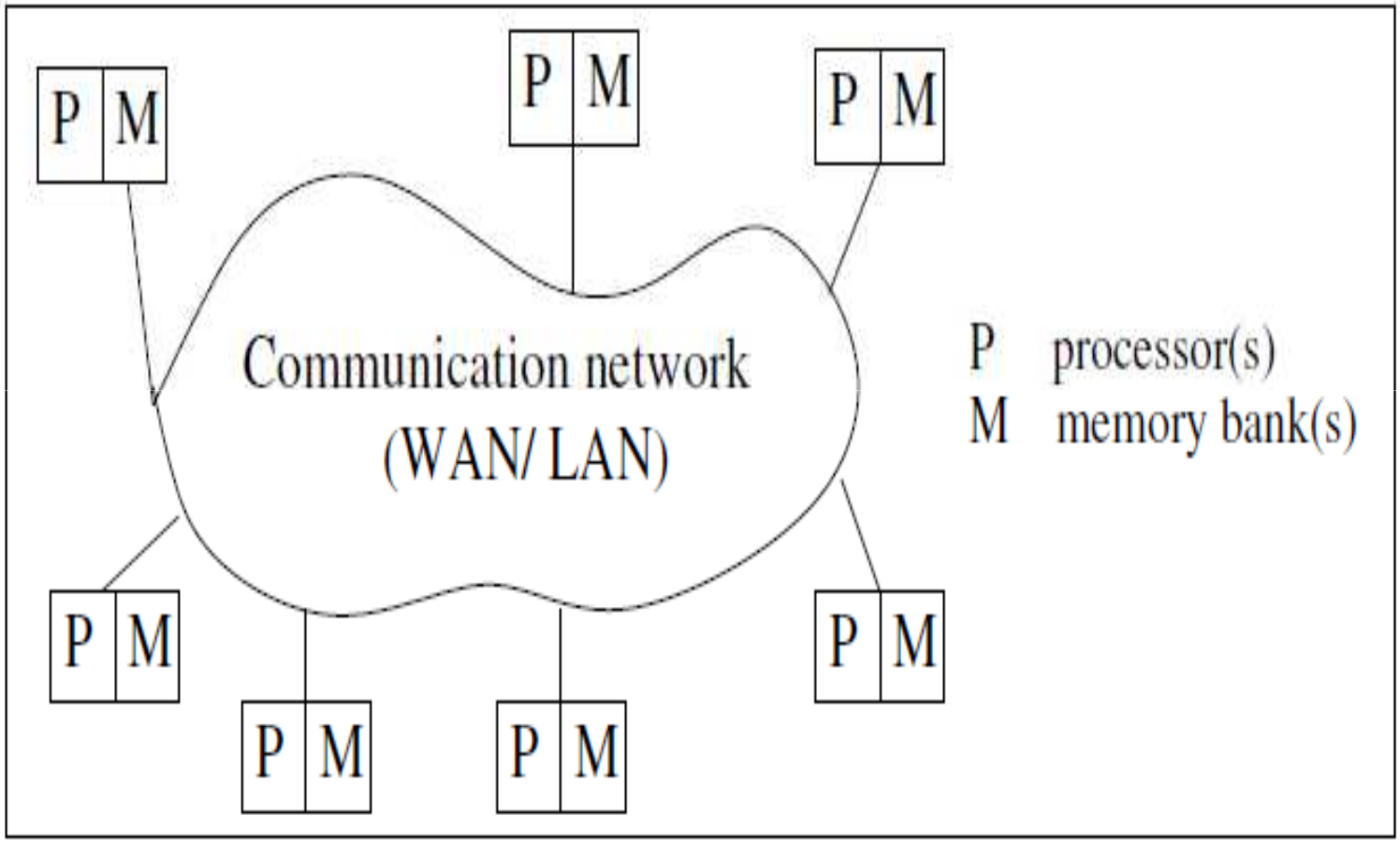


Figure 1.1: A distributed system connects processors by a communication network.

Relation Between Software Components

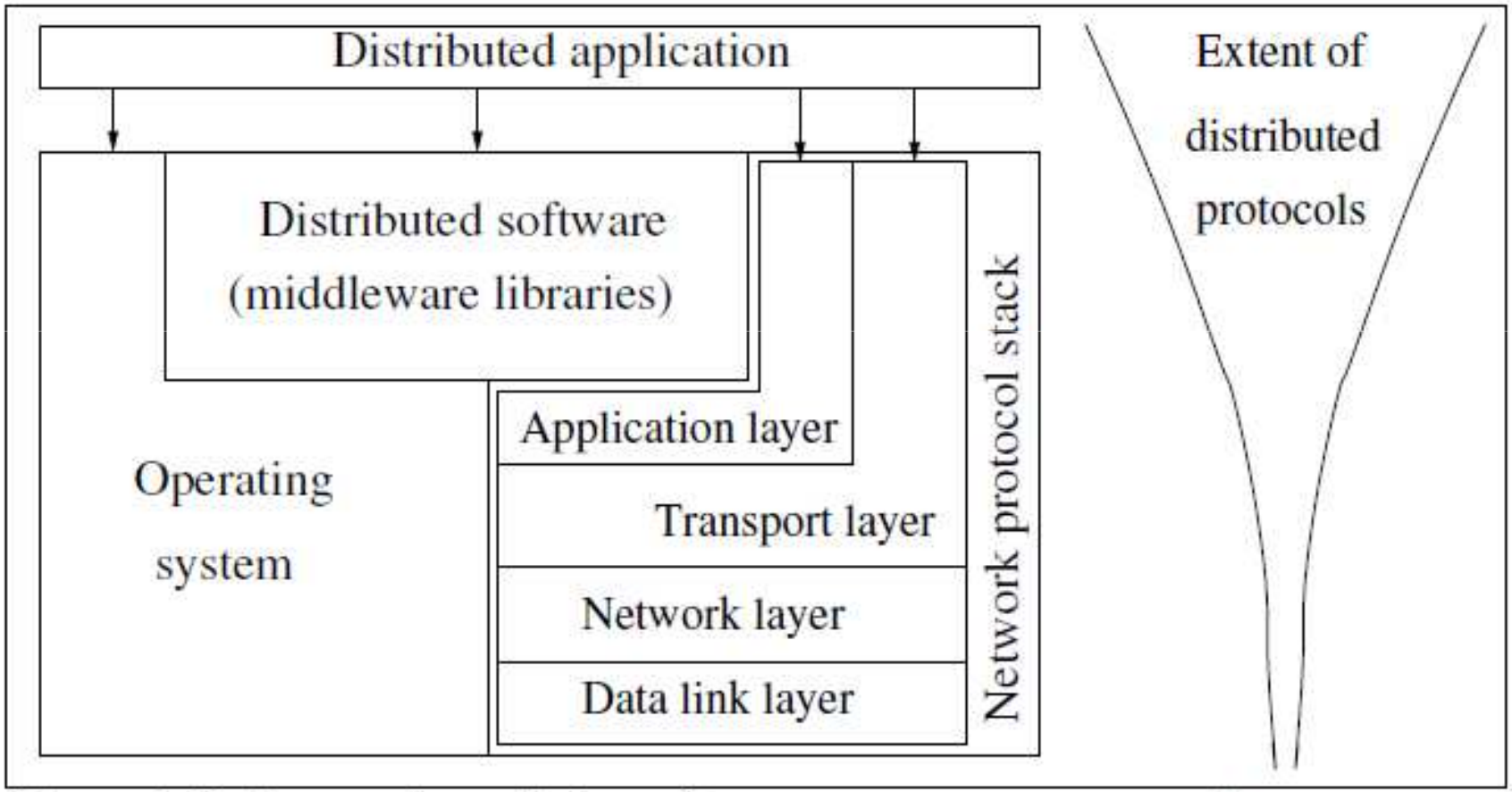


Figure 1.2: Interaction of the software components at each process.

Motivation for Distributed Systems

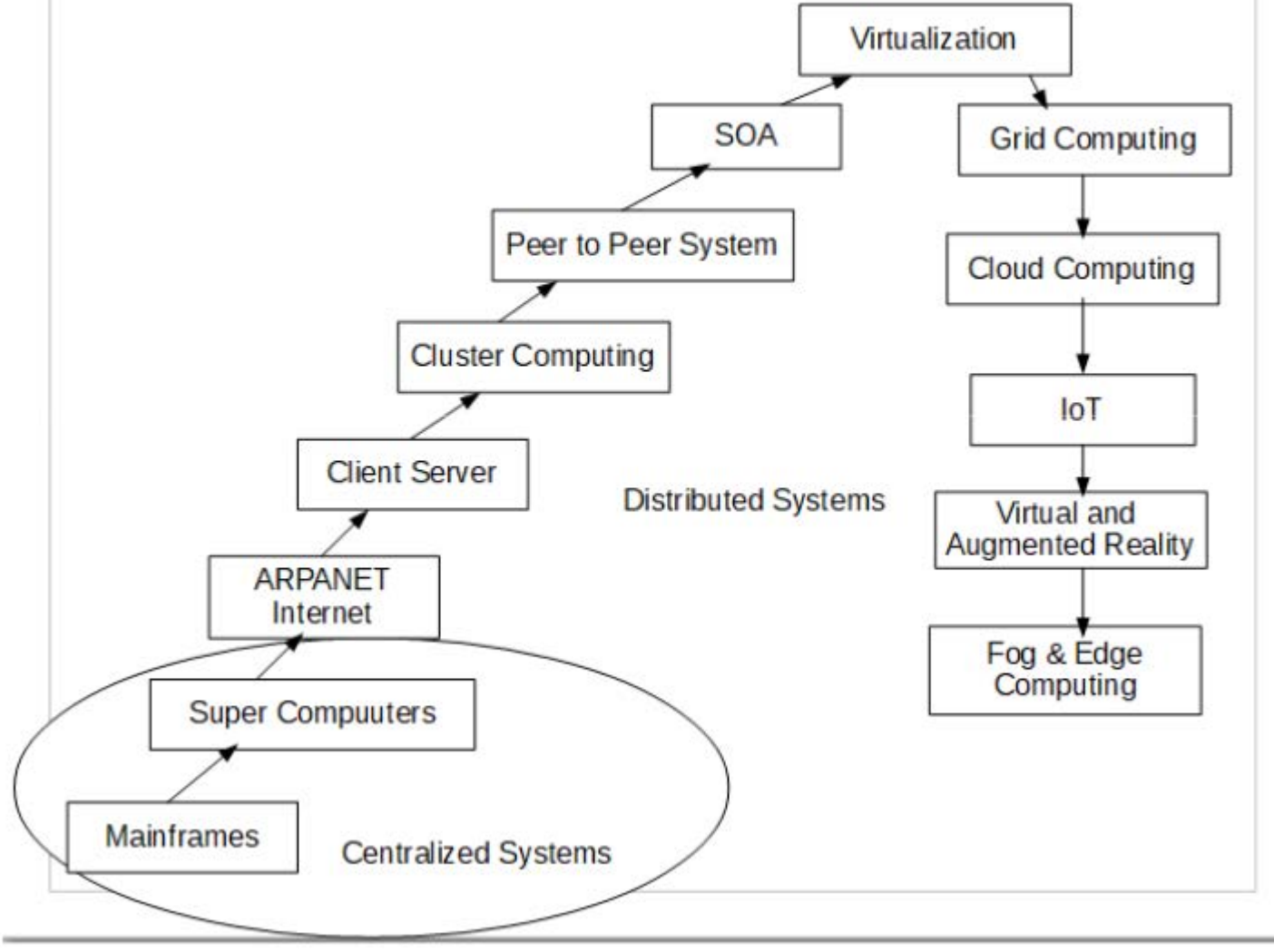
- **Inherently distributed computation**
  - Money transfer in banking.
- **Resource sharing**
  - Data in DBs, special libraries, data, files cannot be replicated.
  - Distributed DB
- **Access to geographically remote data & resources**
  - Replication of data is not always possible (data is too large and sensitive)
  - Eg: Payroll Data is too large and sensitive to replicate to every branch.
- **Increased performance/cost ratio**
  - Due to resource sharing, partitioning task across various computers.
- **Reliability**
  - Availability, integrity, fault-tolerance
- **Scalability**
  - Adding more processors to the WAN is not difficult
- **Modularity and incremental expandability**
  - Heterogeneous processors can be easily added without affecting performance.
  - Existing processors can also be easily replaced by others.

Summary

- Introduction
- Characteristics
- Motivation

Thank You

Evolution of Distributed Systems



Challenges: System Perspective (1)

- Communication mechanisms: E.g., Remote Procedure Call (RPC), remote object invocation (ROI), message-oriented vs. stream-oriented communication
- Processes: Code migration, process/thread management at clients and servers, design of software and mobile agents
- Naming: Easy to use identifiers needed to locate resources and processes transparently and scalably
- Synchronization
- Data storage and access
  - Schemes for data storage, search, and lookup should be fast and scalable across network
  - Revisit file system design
- Consistency and replication
  - Replication for fast access, scalability, avoid bottlenecks
  - Require consistency management among replicas

Challenges: System Perspective (2)

- Fault-tolerance: correct and efficient operation despite link, node, process failures
- Distributed systems security
  - Secure channels, access control, key management (key generation and key distribution), authorization, secure group management
- Scalability and modularity of algorithms, data, services
- Some experimental systems: Globe, Globus, Grid

Challenges: System Perspective (3)

- API for communications, services: ease of use
- Transparency: hiding implementation policies from user
  - Access: hide differences in data rep across systems, provide uniform operations to access resources
  - Location: locations of resources are transparent
  - Migration: relocate resources without renaming
  - Relocation: relocate resources as they are being accessed
  - Replication: hide replication from the users
  - Concurrency: mask the use of shared resources
  - Failure: reliable and fault-tolerant operation

Challenges: Algorithm/Design (1)

- Useful execution models and frameworks: to reason with and design correct distributed programs
  - Interleaving model
  - Partial order model
  - Input/Output automata
  - Temporal Logic of Actions
- Dynamic distributed graph algorithms and routing algorithms
  - System topology: distributed graph, with only local neighborhood knowledge
  - Graph algorithms: building blocks for group communication, data dissemination, object location
  - Algorithms need to deal with dynamically changing graphs
  - Algorithm efficiency: also impacts resource consumption, latency, traffic, congestion

Challenges: Algorithm/Design (2)

- Time and global state
  - 3D space, 1D time
  - Physical time (clock) accuracy
  - Logical time captures inter-process dependencies and tracks relative time progression
  - Global state observation: inherent distributed nature of system
  - Concurrency measures: concurrency depends on program logic, execution speeds within logical threads, communication speeds

Challenges: Algorithm/Design (3)

- Synchronization/coordination mechanisms
  - Physical clock synchronization: hardware drift needs correction
  - Leader election: select a distinguished process, due to inherent symmetry
  - Mutual exclusion: coordinate access to critical resources
  - Distributed deadlock detection and resolution: need to observe global state; avoid duplicate detection, unnecessary aborts
  - Termination detection: global state of quiescence; no CPU processing and no in-transit messages
  - Garbage collection: Reclaim objects no longer pointed to by any process

Challenges: Algorithm/Design (4)

- Group communication, multicast, and ordered message delivery
  - Group: processes sharing a context, collaborating
  - Multiple joins, leaves, fails
  - Concurrent sends: semantics of delivery order
- Monitoring distributed events and predicates
  - Predicate: condition on global system state
  - Debugging, environmental sensing, industrial process control, analyzing event streams
- Distributed program design and verification tools
- Debugging distributed programs

Challenges: Algorithm/Design (5)

- Data replication, consistency models, and caching
  - Fast, scalable access;
  - coordinate replica updates;
  - optimize replica placement
- World Wide Web design: caching, searching, scheduling
  - Global scale distributed system; end-users
  - Read-intensive; prefetching over caching
  - Object search and navigation are resource-intensive
  - User-perceived latency

Challenges: Algorithm/Design (6)

- Distributed shared memory abstraction
  - Wait-free algorithm design: process completes execution, irrespective of actions of other processes, i.e., $n-1$ fault-resilience
  - Mutual exclusion
    - ⋆ Bakery algorithm, semaphores, based on atomic hardware primitives, fast algorithms when contention-free access
  - Register constructions
    - ⋆ Revisit assumptions about memory access
    - ⋆ What behavior under concurrent unrestricted access to memory? Foundation for future architectures, decoupled with technology (semiconductor, biocomputing, quantum ...)
  - Consistency models:
    - ⋆ coherence versus access cost trade-off
    - ⋆ Weaker models than strict consistency of uniprocessors

Challenges: Algorithm/Design (7)

- Reliable and fault-tolerant distributed systems
  - Consensus algorithms: processes reach agreement in spite of faults (under various fault models)
  - Replication and replica management
  - Voting and quorum systems
  - Distributed databases, commit: ACID properties
  - Self-stabilizing systems: "illegal" system state changes to "legal" state; requires built-in redundancy
  - Checkpointing and recovery algorithms: roll back and restart from earlier "saved" state
  - Failure detectors:
    - ⋆ Difficult to distinguish a "slow" process/message from a failed process/ never sent message
    - ⋆ algorithms that "suspect" a process as having failed and converge on a determination of its up/down status

Challenges: Algorithm/Design (8)

- Load balancing: to reduce latency, increase throughput, dynamically. E.g., server farms
  - Computation migration: relocate processes to redistribute workload
  - Data migration: move data, based on access patterns
  - Distributed scheduling: across processors
- Real-time scheduling: difficult without global view, network delays make task harder
- Performance modeling and analysis: Network latency to access resources must be reduced
  - Metrics: theoretical measures for algorithms, practical measures for systems
  - Measurement methodologies and tools

Applications and Emerging Challenges (1)

- Mobile systems
  - Wireless communication: unit disk model; broadcast medium (MAC), power management etc.
  - CS perspective: routing, location management, channel allocation, localization and position estimation, mobility management
  - Base station model (cellular model)
  - Ad-hoc network model (rich in distributed graph theory problems)
- Sensor networks: Processor with electro-mechanical interface
- Ubiquitous or pervasive computing
  - Processors embedded in and seamlessly pervading environment
  - Wireless sensor and actuator mechanisms; self-organizing; network-centric, resource-constrained
  - E.g., intelligent home, smart workplace

Applications and Emerging Challenges (2)

- Peer-to-peer computing
  - No hierarchy; symmetric role; self-organizing; efficient object storage and lookup;scalable; dynamic reconfig
- Publish/subscribe, content distribution
  - Filtering information to extract that of interest
- Distributed agents
  - Processes that move and cooperate to perform specific tasks; coordination, controlling mobility, software design and interfaces
- Distributed data mining
  - Extract patterns/trends of interest
  - Data not available in a single repository

Applications and Emerging Challenges (3)

- Grid computing
  - Grid of shared computing resources; use idle CPU cycles
  - Issues: scheduling, QOS guarantees, security of machines and jobs
- Security
  - Confidentiality, authentication, availability in a distributed setting
  - Manage wireless, peer-to-peer, grid environments
    - ★ Issues: e.g., Lack of trust, broadcast media, resource-constrained, lack of structure

Chapter 2: A Model of Distributed Computations

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

A Distributed Program

- A distributed program is composed of a set of n asynchronous processes, p_1, p_2, ..., p_i, ..., p_n.
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- 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.

A 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.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

A Model of Distributed Executions

- 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, ..., e_i^x, e_i^{x+1}, ... and is denoted by $\mathcal{H}_i$ where

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

  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.

A Model of Distributed Executions

- 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.

A Model of Distributed Executions

- 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 2.1, 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.

A Model of Distributed Executions

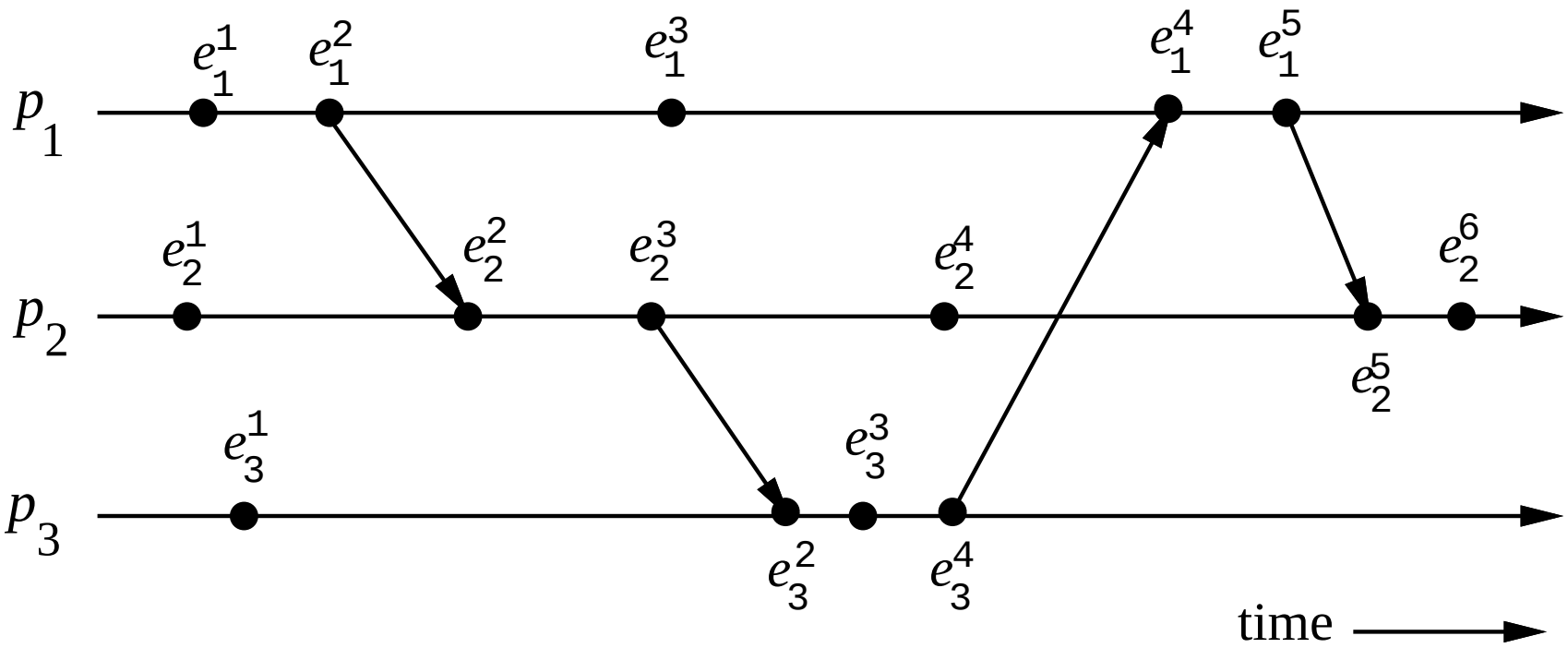


Figure 2.1: The space-time diagram of a distributed execution.

A Model of Distributed Executions

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.

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

- The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as $\mathcal{H} = (H, \rightarrow)$.

A Model of Distributed Executions

. . . Causal Precedence Relation

- 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 2.1, $e_1^1 \rightarrow e_3^3$ 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 2.1, event e_2^6 has the knowledge of all other events shown in the figure.

A Model of Distributed Executions

. . . Causal Precedence Relation

- For any two events e_i and e_j, $e_i \not\rightarrow e_j$ denotes the fact that event e_j does not directly or transitively dependent on event e_i. That is, event e_i does not causally affect event e_j.
- In this case, event e_j is not aware of the execution of e_i or any event executed after e_i on the same process.
- For example, in Figure 2.1, $e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$.

Note the following two rules:

- For any two events e_i and e_j, $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$.
- For any two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.

A Model of Distributed Executions

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 2.1, $e_1^3 \parallel e_3^3$ 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 2.1, $e_3^3 \parallel e_2^4$ 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 $e_i \parallel e_j$.

A Model of Distributed Executions

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.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time.
- However, if processor speed and message delays would have 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 not, does not change the outcome of the computation.
- Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, we can assume that these events occured at the same instant in physical time.

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.

Models of Communication Networks

- 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 $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

- 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.

Global State of a Distributed System

"A collection of the local states of its components, namely, the processes and the communication channels."

- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that or receives the message and the state of the channel on which the message is received.

. . . Global State of a Distributed System

Notations

- LS_i^x denotes the state of process p_i after the occurrence of event e_i^x and before the event e_i^{x+1}.
- LS_i^0 denotes the initial state of process p_i.
- LS_i^x is a result of the execution of all the events executed by process p_i till e_i^x.
- Let $send(m) \leq LS_i^x$ denote the fact that $\exists y{:}1 \leq y \leq x :: e_i^y = send(m)$.
- Let $rec(m) \not\leq LS_i^x$ denote the fact that $\forall y{:}1 \leq y \leq x :: e_i^y \neq rec(m)$.

. . . Global State of a Distributed System

A Channel State

- The state of a channel depends upon the states of the processes it connects.
- Let $SC_{ij}^{x,y}$ denote the state of a channel C_{ij}.

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} | \ send(m_{ij}) \leq e_i^x \bigwedge rec(m_{ij}) \not\leq e_j^y\}$$

Thus, channel state $SC_{ij}^{x,y}$ denotes all messages that p_i sent upto event e_i^x and which process p_j had not received until event e_j^y.

. . . Global State of a Distributed System

Global State

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state GS is defined as,

$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$$

- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)

. . . Global State of a Distributed System

A Consistent Global State

- Even if the state of all the components is not recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent.
- Basic idea is that a state should not violate causality – an effect should not be present without its cause. A message cannot be received if it was not sent.
- Such states are called *consistent global states* and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.
- A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

$$\forall m_{ij}: \; send(m_{ij}) \not\leq LS_i^{x_i} \;\Leftrightarrow\; m_{ij} \notin SC_{ij}^{x_i,y_j} \bigwedge rec(m_{ij}) \not\leq LS_j^{y_j}$$

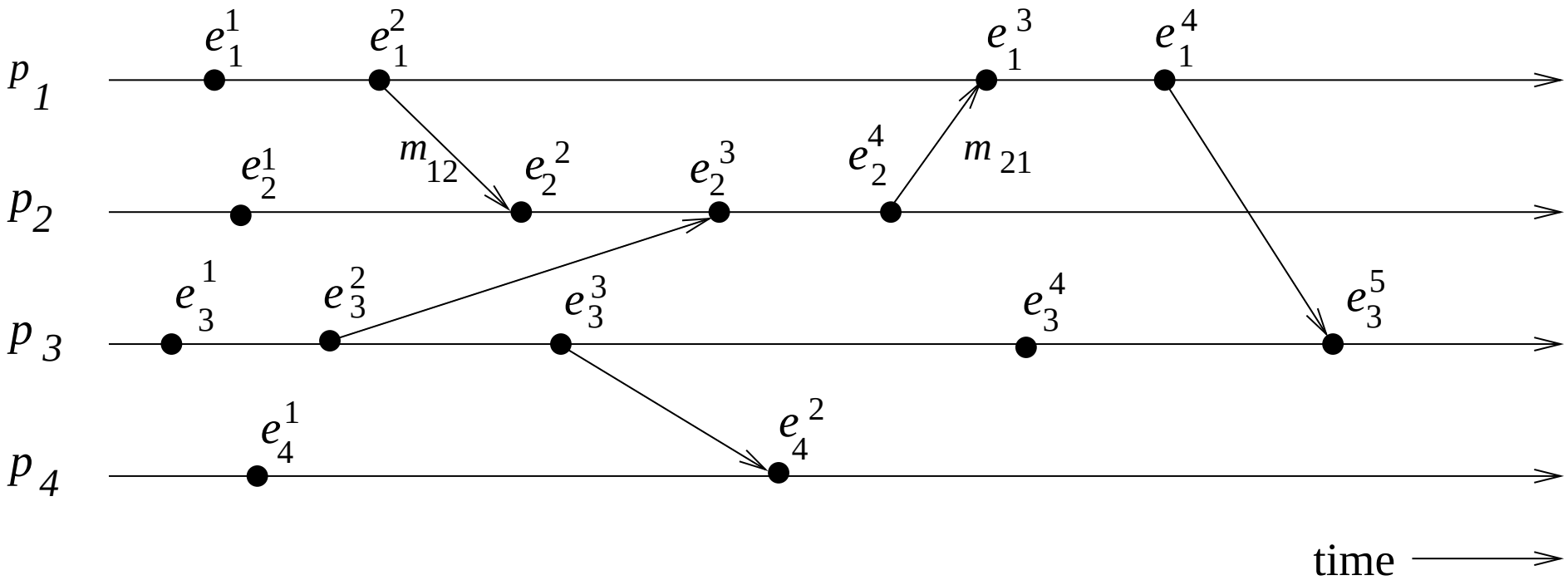
- That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.

. . . Global State of a Distributed System

An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



. . . Global State of a Distributed System

In Figure 2.2:

- A global state $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent because the state of p_2 has recorded the receipt of message m_{12}, however, the state of p_1 has not recorded its send.
- A global state GS_2 consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except C_{21} that contains message m_{21}.

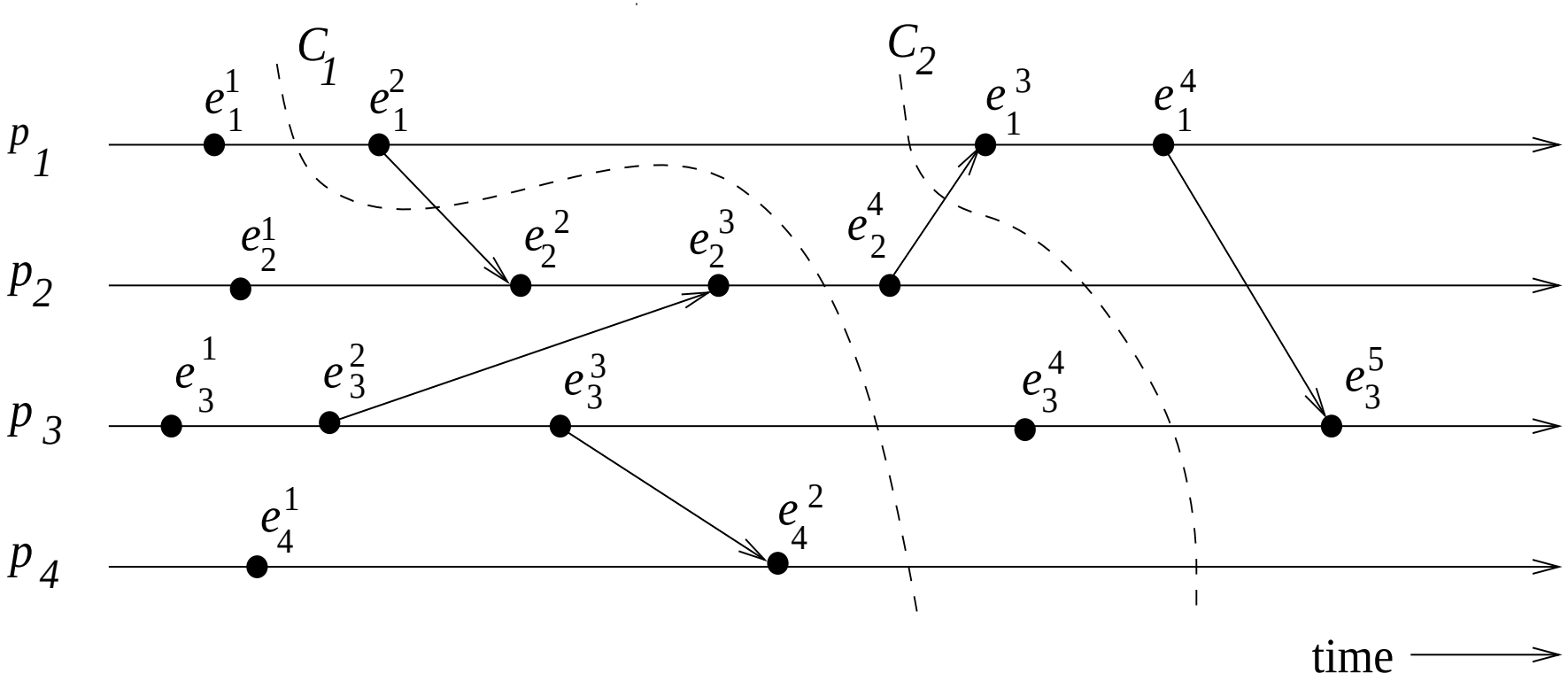
Cuts of a Distributed Computation

"In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line."

- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut C, let PAST(C) and FUTURE(C) denote the set of events in the PAST and FUTURE of C, respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space-time diagram.
- Cuts in a space-time diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

. . . Cuts of a Distributed Computation

Figure 2.3: Illustration of cuts in a distributed execution.



. . . Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure 2.3, cut C_2 is a consistent cut.)
- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is *inconsistent* if a message crosses the cut from the FUTURE to the PAST. (In Figure 2.3, cut C_1 is an inconsistent cut.)

Past and Future Cones of an Event

Past Cone of an Event

- An event e_j could have been affected only by all events e_i such that $e_i \rightarrow e_j$.
- In this situtaion, all the information available at e_i could be made accessible at e_j.
- All such events e_i belong to the past of e_j.

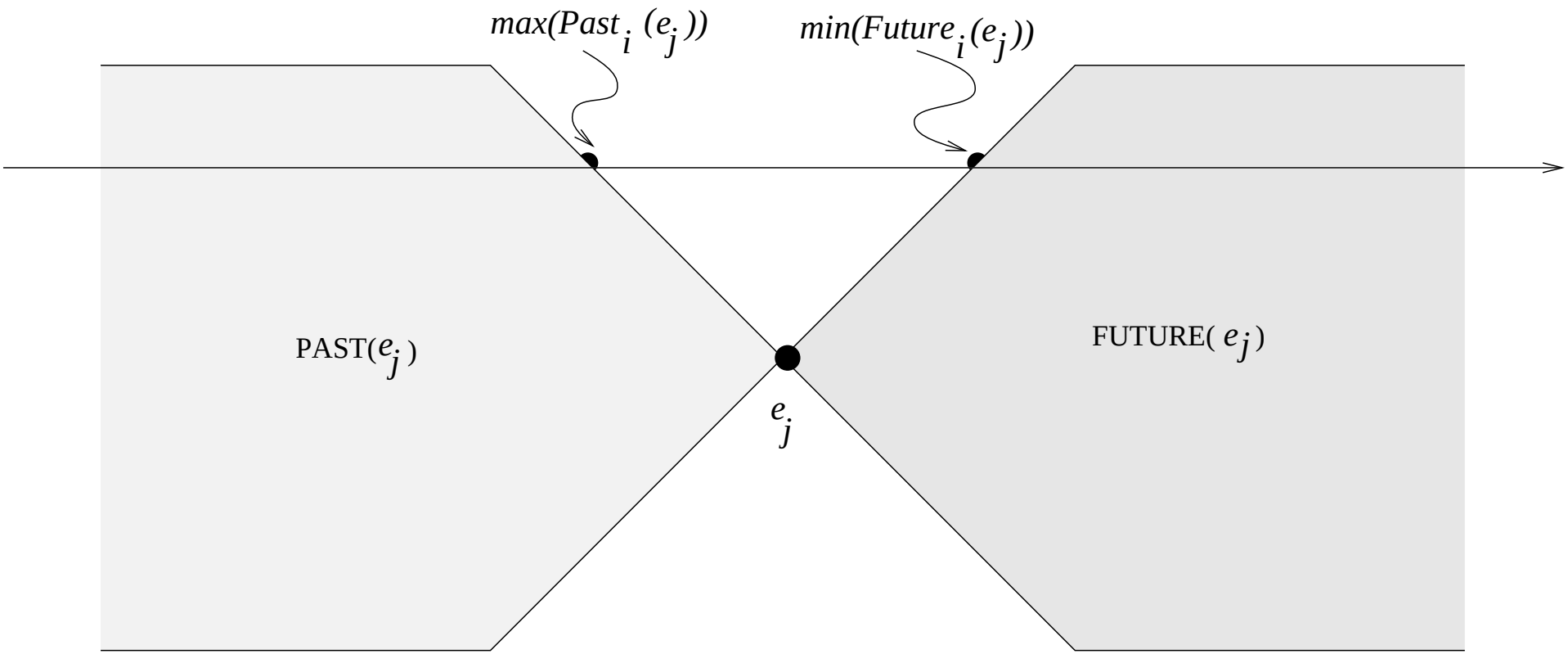
Let $Past(e_j)$ denote all events in the past of e_j in a computation $(H, \rightarrow)$. Then,

$$Past(e_j) = \{e_i | \forall e_i \in H,\ e_i \rightarrow e_j\ \}.$$

- Figure 2.4 (next slide) shows the past of an event e_j.

. . . Past and Future Cones of an Event

Figure 2.4: Illustration of past and future cones.



. . . Past and Future Cones of an Event

- Let $Past_i(e_j)$ be the set of all those events of $Past(e_j)$ that are on process p_i.
- $Past_i(e_j)$ is a totally ordered set, ordered by the relation $\rightarrow_i$, whose maximal element is denoted by $max(Past_i(e_j))$.
- $max(Past_i(e_j))$ is the latest event at process p_i that affected event e_j (Figure 2.4).

. . . Past and Future Cones of an Event

- Let $Max\_Past(e_j) = \bigcup_{(\forall i)}\{max(Past_i(e_j))\}$.
- $Max\_Past(e_j)$ consists of the latest event at every process that affected event e_j and is referred to as the *surface of the past cone* of e_j.
- $Past(e_j)$ represents all events on the past light cone that affect e_j.

Future Cone of an Event

- The future of an event e_j, denoted by $Future(e_j)$, contains all events e_i that are causally affected by e_j (see Figure 2.4).
- In a computation $(H, \rightarrow)$, $Future(e_j)$ is defined as:

$$Future(e_j) = \{e_i | \forall e_i \in H,\ e_j \rightarrow e_i\}.$$

. . . Past and Future Cones of an Event

- Define $Future_i(e_j)$ as the set of those events of $Future(e_j)$ that are on process p_i.
- define $min(Future_i(e_j))$ as the first event on process p_i that is affected by e_j.
- Define $Min\_Future(e_j)$ as $\bigcup_{(\forall i)}\{min(Future_i(e_j))\}$, which consists of the first event at every process that is causally affected by event e_j.
- $Min\_Future(e_j)$ is referred to as the *surface of the future cone* of e_j.
- All events at a process p_i that occurred after $max(Past_i(e_j))$ but before $min(Future_i(e_j))$ are concurrent with e_j.
- Therefore, all and only those events of computation H that belong to the set "$H - Past(e_j) - Future(e_j)$" are concurrent with event e_j.

Models of Process Communications

- There are two basic models of process communications – synchronous and asynchronous.
- The *synchronous* communication model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process.
- The sender process resumes execution only after it learns that the receiver process has accepted the message.
- Thus, the sender and the receiver processes must synchronize to exchange a message. On the other hand,
- *asynchronous* communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
- After having sent a message, the sender process does not wait for the message to be delivered to the receiver process.
- The message is bufferred by the system and is delivered to the receiver process when it is ready to accept the message.

. . . Models of Process Communications

- Neither of the communication models is superior to the other.
- Asynchronous communication provides higher parallelism because the sender process can execute while the message is in transit to the receiver.
- However, A buffer overflow may occur if a process sends a large number of messages in a burst to another process.
- Thus, an implementation of asynchronous communication requires more complex buffer management.
- In addition, due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications.
- Synchronous communication is simpler to handle and implement.
- However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks.

Synchronizing Physical Clock Logical Clock & Vector Clock

Y. V. Lokeswari

ASP/CSE

Overview

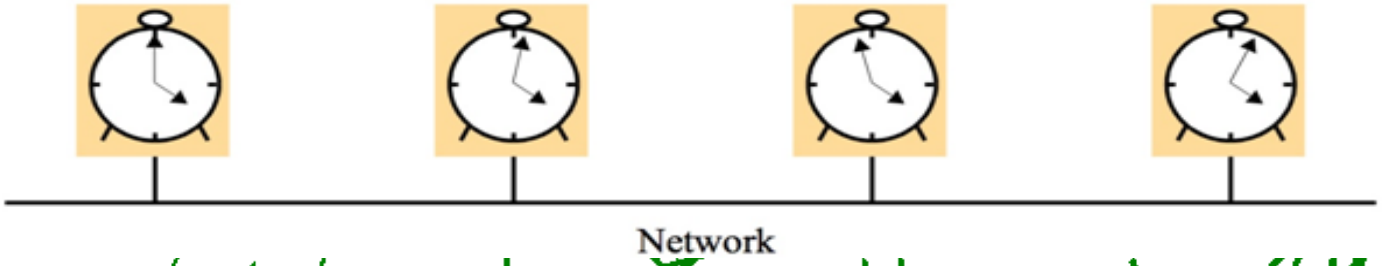
- Physical Clocks
- Synchronizing Physical Clock (Algorithms)
- Problems with Physical Clock
- Lamport's Logical Clock
- Problems with Logical Clock
- Vector Clock
- Drawbacks of Vector Clocks
- Applications of Vector Clocks

Clock Synchronization

- Temporal ordering of events produced by concurrent processes
- Synchronization between senders and receivers of messages
- Serialization of concurrent access for shared objects
- **Physical Clock**: It is the internal clock present in a computer. (Time of a day)
- **Logical Clock**: keeps track of event ordering among related (causal) events.

Clock Synchronization

- Getting two systems to agree on time
  - Two clocks hardly ever agree
  - **Quartz oscillators oscillate at slightly different frequencies**
- **Clock Drift :** Clocks tick at different rates. Create ever-widening gap in perceived time
  - Clock drift in ordinary clocks based on quartz crystal is 10^-6 seconds.
  - This creates a difference of 1 sec for every 11.6 days (1,000,000 sec)
  - Clock drift of high precision clock is 10^-7 to 10^-8
- **Clock Skew :** Difference between two clocks at one point in time.



Clock Synchronization

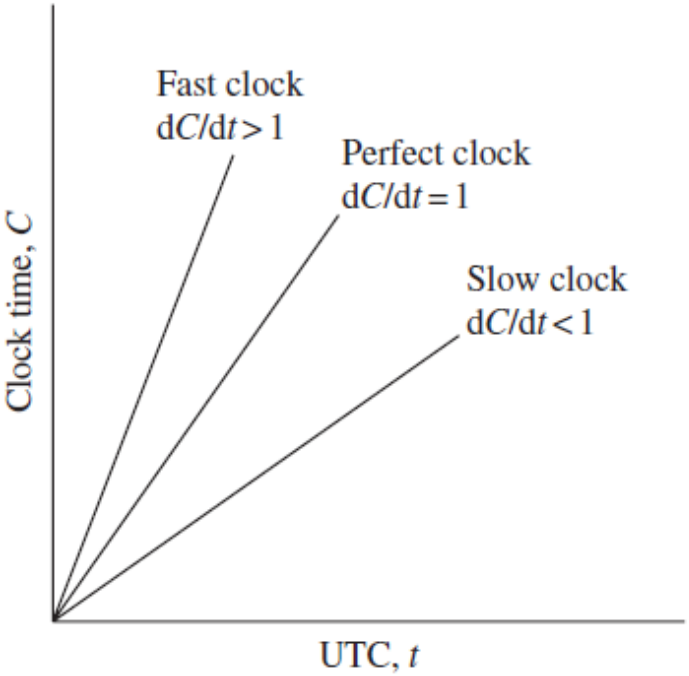
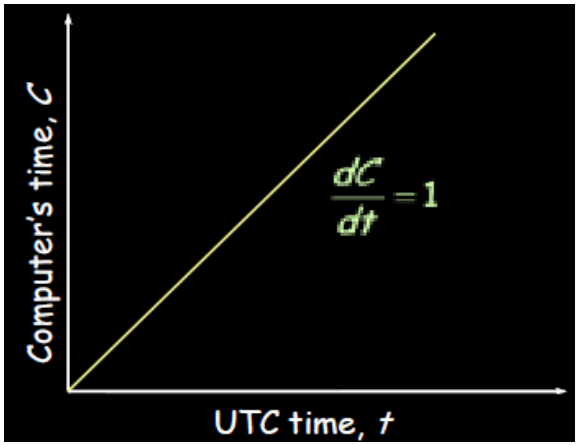


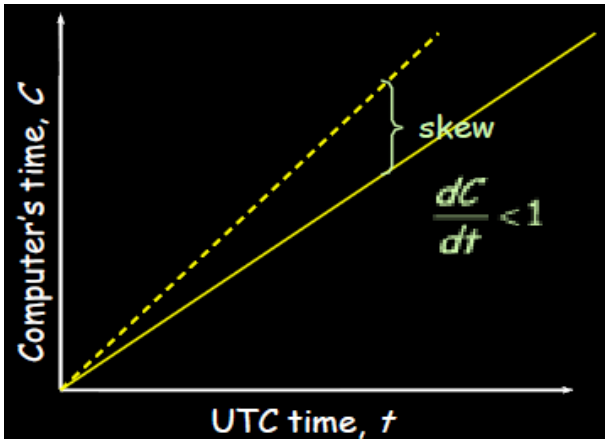
Figure 3.8 The behavior of fast, slow, and perfect clocks with respect to UTC.

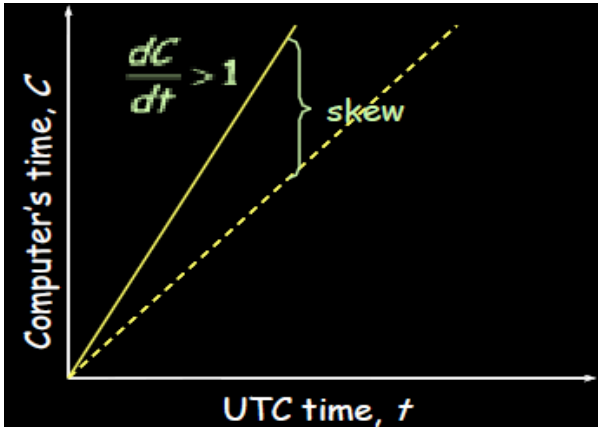
Clock Synchronization

- Dealing with Clock Drift: Go for gradual clock correction

UTC : Universal Coordinated Time







If slow:

Make clock run faster until it Synchronizes

If fast:

Make clock run slower until it Synchronizes

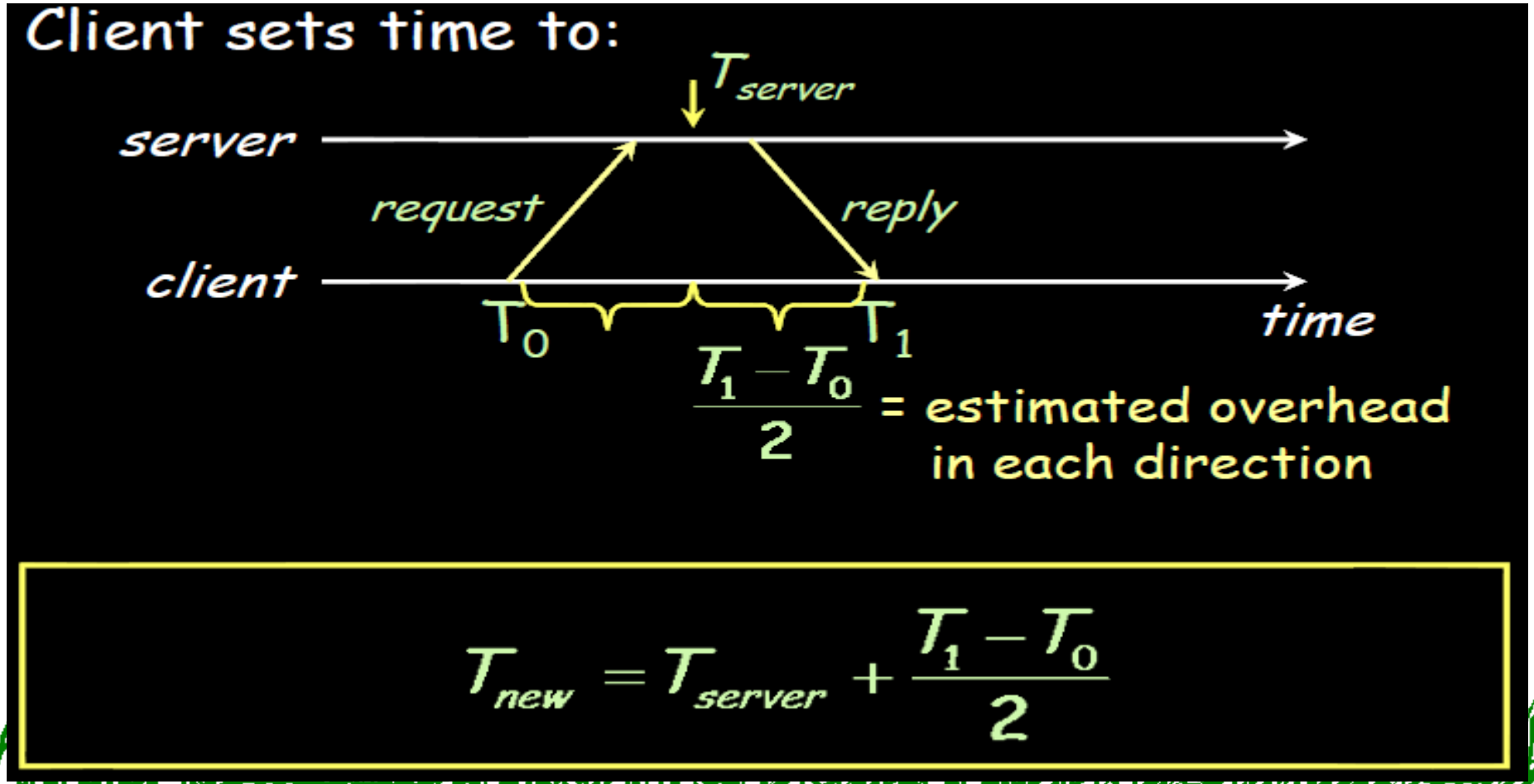
Clock Synchronization

- **External Synchronization** : Clock synchronizes to correct time from external timing elements like Radio / Satellite time.
- **Internal Synchronization** : Clock synchronizes to correct time by getting timings from other computers.
- 3 - Algorithms for Synchronizing Physical Clock
  1. **Cristian's Algorithm**
  2. **Berkeley Algorithm**
  3. **Network Time Protocol (NTP)**

Clock Synchronization

1. Cristian's Algorithm

(T1-T0)/2 is round-trip time



Clock Synchronization

1. Cristian's Algorithm – Example

- Send request at 5:08:15.100 (T0)
- Receive response at 5:08:15.900 (T1)
- Response contains 5:09:25.300 (Tserver)
- Elapsed time is T1 -T0

  5:08:15.900 - 5:08:15.100 = 800 msec
- Best guess: timestamp was generated

  400 msec ago (800/2)
- Set time to Tserver+ elapsed time

  5:09:25.300 + 400 = 5:09.25.700

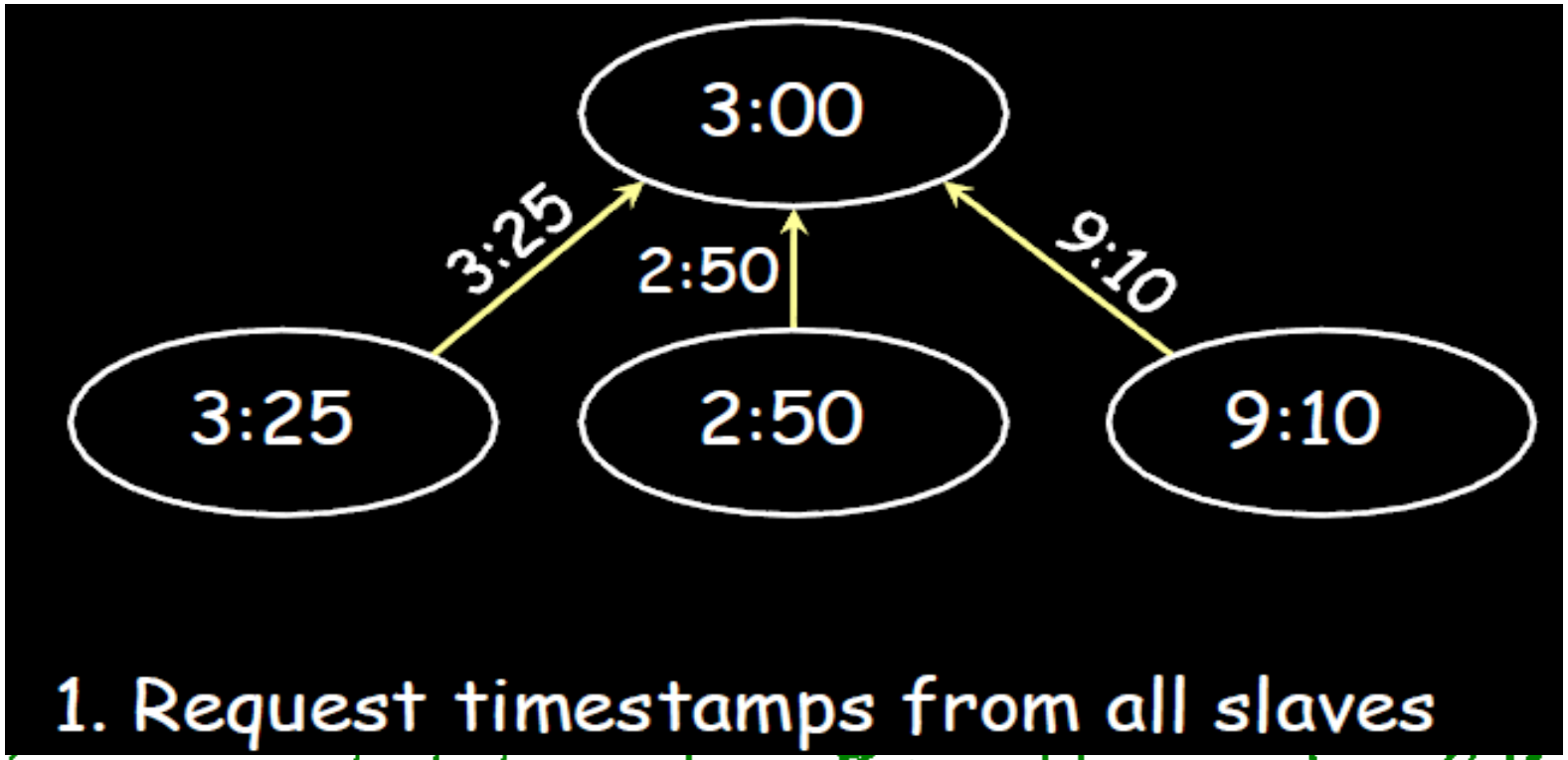
Clock Synchronization

2. Berkeley Algorithm

- Machines run **time dæmon** Process that implements protocol
- One machine is elected (or designated) as the server (**master)** Others are **slaves**
- Master polls each machine periodically and ask each machine for time
- Can use Cristian's algorithm to compensate for network latency
- When results are received, master computes average of times - Including master's time
- **Hope**: Average cancels out individual clock's tendencies to run fast or slow (clock drift)
- Send offset by which each clock needs adjustment to each slave.
- Algorithm has provisions for ignoring readings from clocks whose skew is too great
- Compute a **fault-tolerant average**
- If master fails - any slave can take over

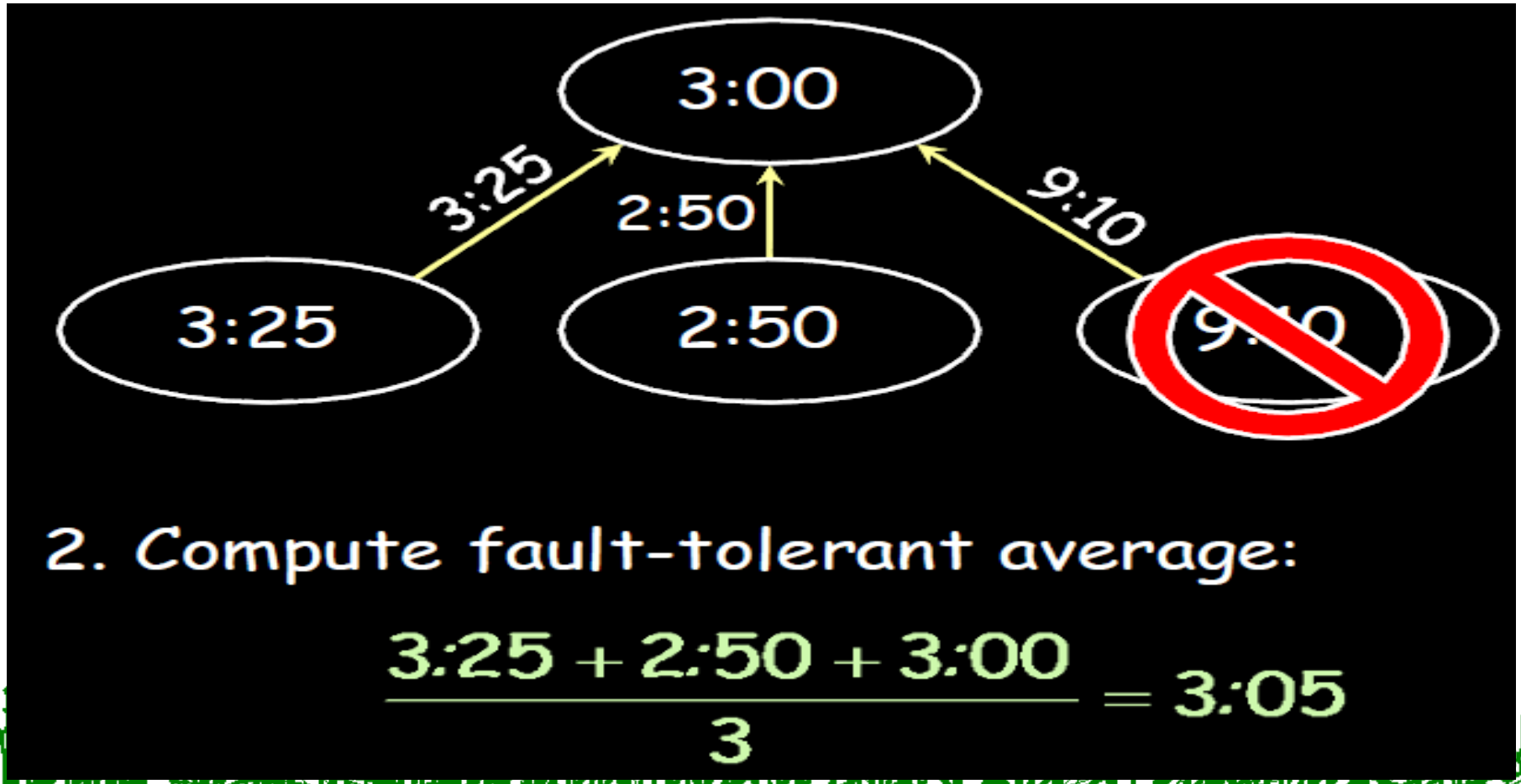
Clock Synchronization

2. Berkeley Algorithm Example



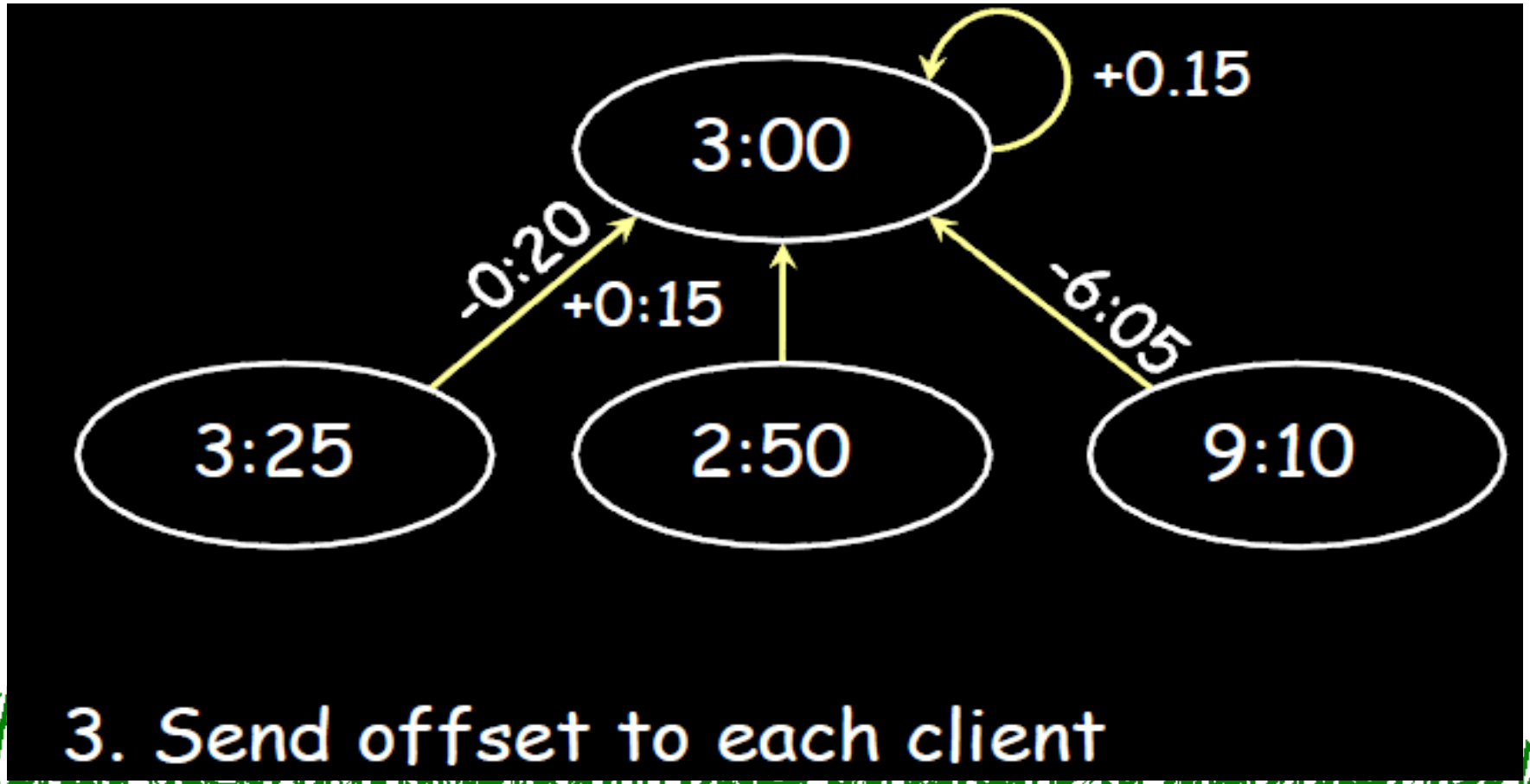
Clock Synchronization

2. Berkeley Algorithm Example



Clock Synchronization

2. Berkeley Algorithm Example



Clock Synchronization

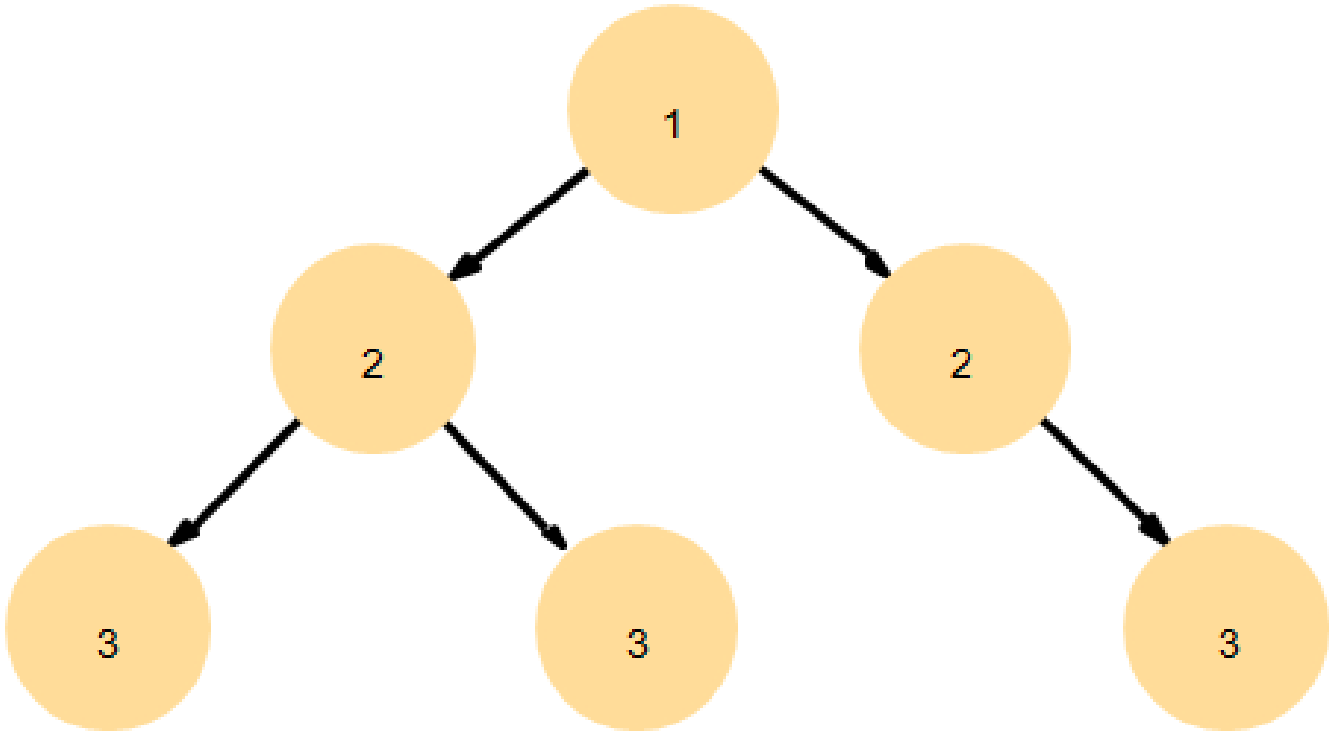
3. Network Time Protocol

- Enable clients across Internet to be accurately synchronized to UTC despite message delays
- Primary servers are connected directly to a time source such as a radio clock receiving UTC; secondary servers are synchronized with primary servers.
- The servers are connected in a logical hierarchy called a **synchronization subnet** whose levels are called **strata**.
- Primary servers occupy stratum 1: they are at the root.
- Stratum 2 servers are secondary servers that are synchronized directly with the primary servers;
- Stratum 3 servers are synchronized with stratum 2 servers, and so on.

Clock Synchronization

3. Network Time Protocol

Arrows denote synchronization control, numbers denote strata.



Clock Synchronization

Network Time Protocol

Let a = $T_1 - T_3$ and b = $T_2 - T_4$.

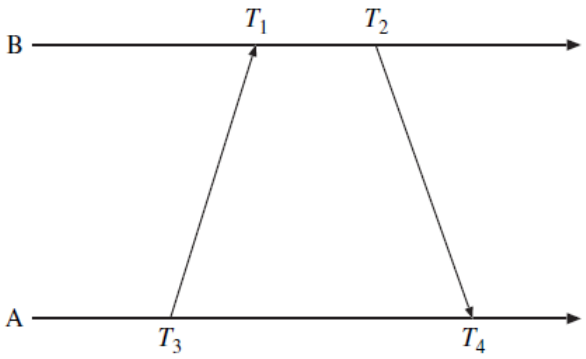


Figure 3.9 Offset and delay estimation [15].

$$\theta = \frac{a+b}{2}, \qquad \delta = a - b.$$

the offset O_i can be estimated as

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2.$$

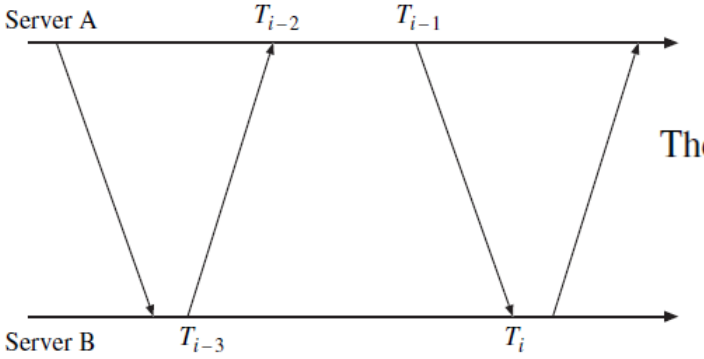


Figure 3.10 Timing diagram for the two servers [15].

The round-trip delay is estimated as

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}).$$

Clock Synchronization

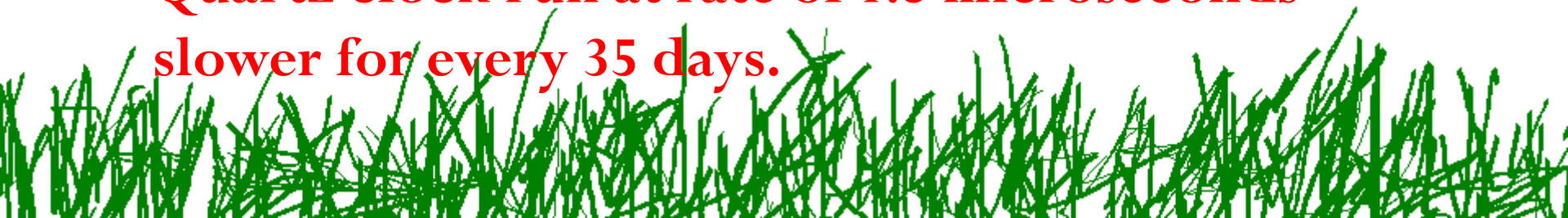
3. Network Time Protocol (Synchronization Modes)

- **Multicast mode**
  - Every node sends its time to all the other nodes in the LAN
  - For high speed LANs
  - Lower accuracy but efficient
- **Procedure call mode**
  - Similar to Cristian's algorithm
- **Symmetric mode**
  - Intended for master servers
  - Pair of servers exchange messages and retain data to improve synchronization over time

All messages delivered unreliably with UDP

Problems with Physical Clocks

- **Clock Drift :** Clocks tick at different rates. Create ever-widening gap in perceived time
- **Clock Skew :** Difference between two clocks at one point in time.
- For quartz crystal clocks, typical drift rate is about one second every 106 seconds =11.6 days
- Best atomic clocks have drift rate of one second in 1013 seconds = 300,000 years.
- **Quartz clock run at rate of 1.5 microseconds slower for every 35 days.**



Problems with Physical Clocks

Logical Clocks

Need for Logical Clock

- For many purposes, it is sufficient to know the order in which events occurred.
- Lamport (1978) — introduce logical *(virtual) time*, synchronize *logical clocks*.
- An event may be an instruction execution, may be a function execution, etc.
- Events include message send / receive

Within a single process, or between two processes on the same computer

- The order in which two **events** occur **can** be determined using the **physical clock**

Between two different computers in a distributed system

- The order in which two events occur **cannot** be determined **using local physical clocks**, since those clocks cannot be synchronized perfectly

Happened Before Relation

- Lamport defined the happened before relation (denoted as "→"), which describes a **causal ordering of events:**

1. if a and b are events in the same process, and a occurred before b, then a → b
2. if a is the event of sending a message m in one process, and b is the event of receiving that message m in another process, then a → b
3. if a → b, and b → c, then a → c (i.e., the relation "→" is transitive

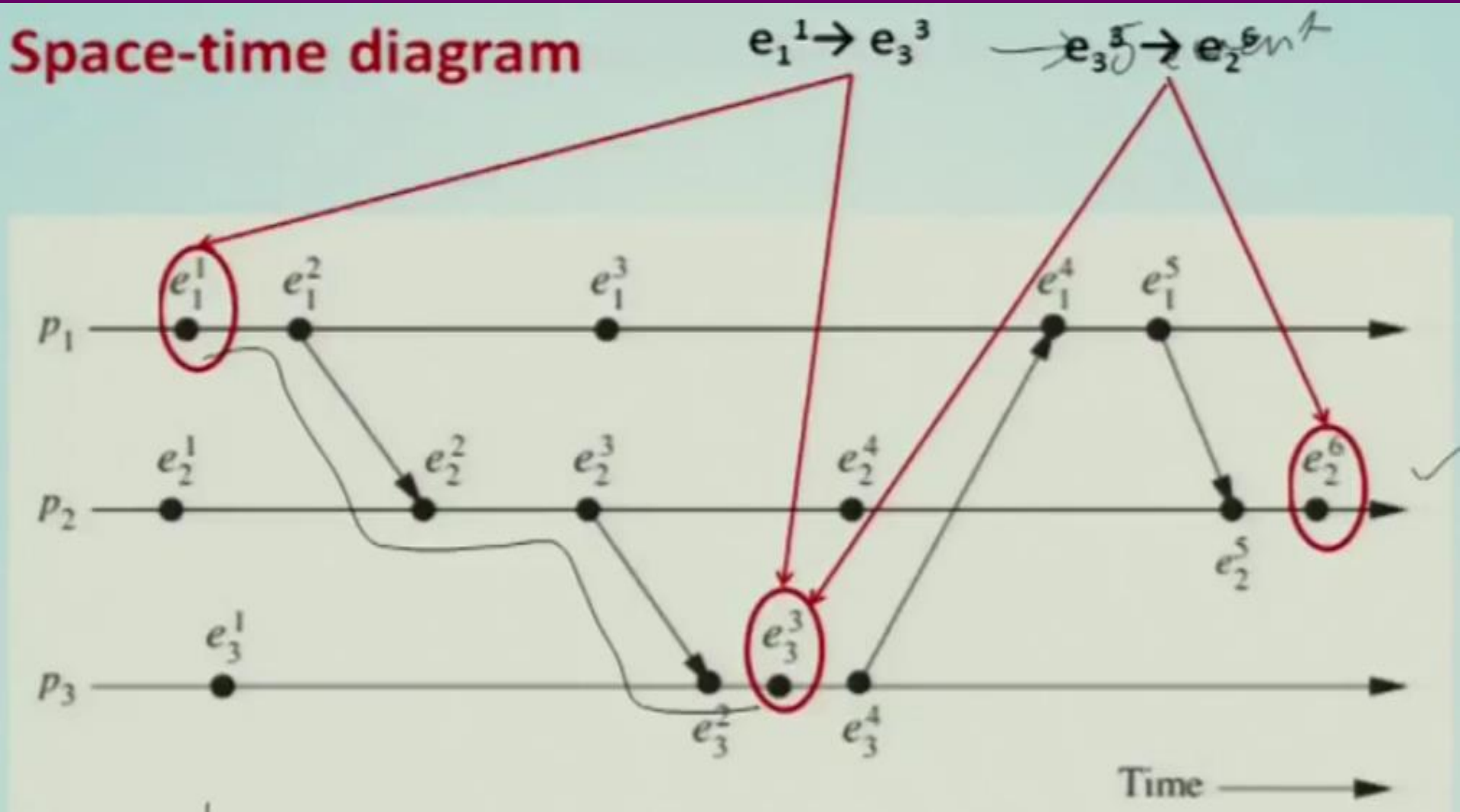
Causality

- Past events influence future events
- This influence among causally related events (those that can be ordered by " → ") is referred to **causally affects.**
- If a → b, event **a** causally affects event **b**

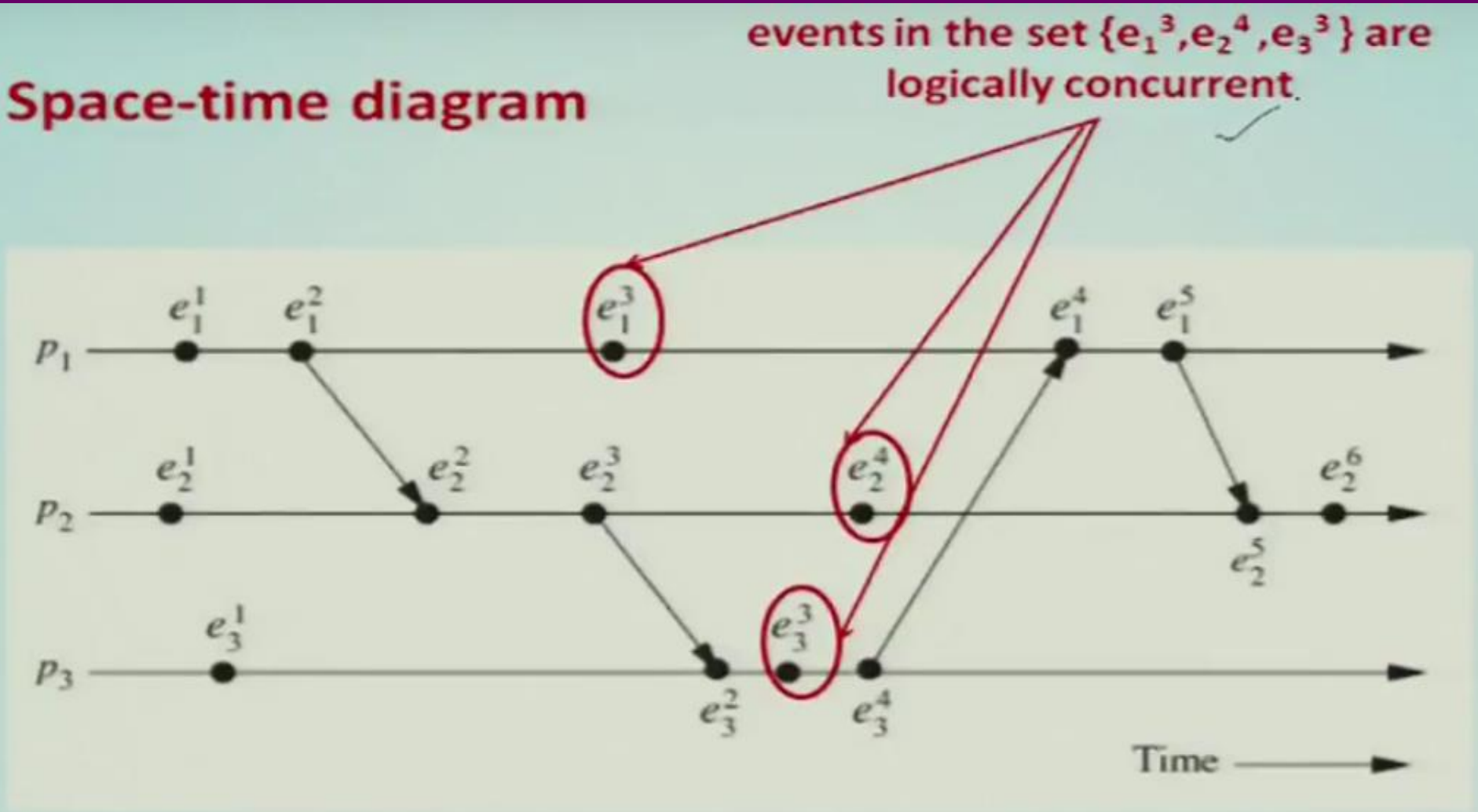
Concurrent events

- Two distinct events **a** and **b** are said to be concurrent (denoted "a || b"), if neither a→ b nor b → a
- In other words, concurrent events do not causally affect each other
- For any two events a and b in a system, either: a → b or b → a or a || b

Causal Events



Concurrent Events



Lamport's Logical Clock

- To implement " → " in a distributed system, Lamport (1978) introduced the concept of logical clocks, which captures " → " numerically
- Each process Pi has a logical clock Ci
- Clock Ci can assign a value Ci (a) to any event **a** in process Pi
- The value Ci (a) is called the timestamp of event **a** in process Pi
- The value C(a) is called the timestamp of event **a** in whatever process it occurred.
- The timestamps have no relation to physical time, which leads to the term logical clock.
- The logical clocks assign monotonically increasing timestamps, and can be implemented by simple counters

Lamport's Logical Clock

- **Clock condition: if a → b, then C(a)<C(b)**
- If event a happens before event b, then the clock value (timestamp) of a should be less than the clock value of b
- Note that we can **not say: if C(a)<C(b), then a → b**
- **Correctness conditions (must be satisfied by the logical clocks to meet the clock condition above):**
- [C1] For any two events a and b in the same process Pi ,

  if a happens before b, then Ci (a) < Ci (b)
- [C2] If event a is the event of sending a message m in process Pi , and event b is the event of receiving that same message m in a different process Pk, then Ci (a) < Ck (b)

Lamport's Logical Clock

- **[IR1]** Clock Ci must be incremented between any two successive events in process Pi

  Ci := Ci + d , (d>0) (usually d=1)

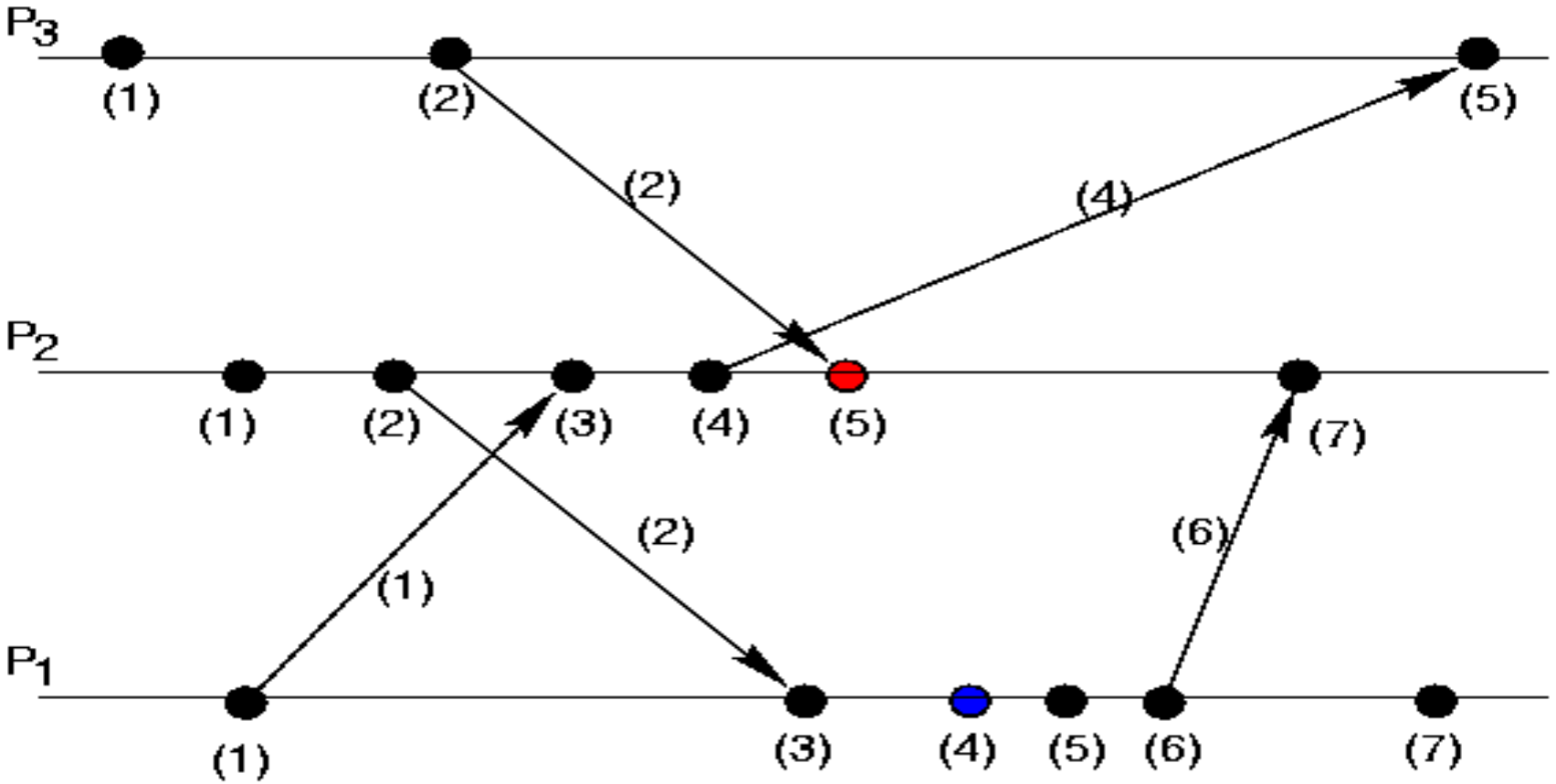
- **[IR2]** If event a is the event of sending a message m in process Pi , then message m is assigned a timestamp Cmsg =Ci (a) When that same message m is received by a different process Pk, Ck is set to a value greater than or equal to its present value, and greater than Cmsg.

  Ck := max(Ck , Cmsg + d) , (d>0) (usually d=1)

Lamport's Logical Clock

IR 1: Ci := Ci + d , (d>0), (usually d=1)

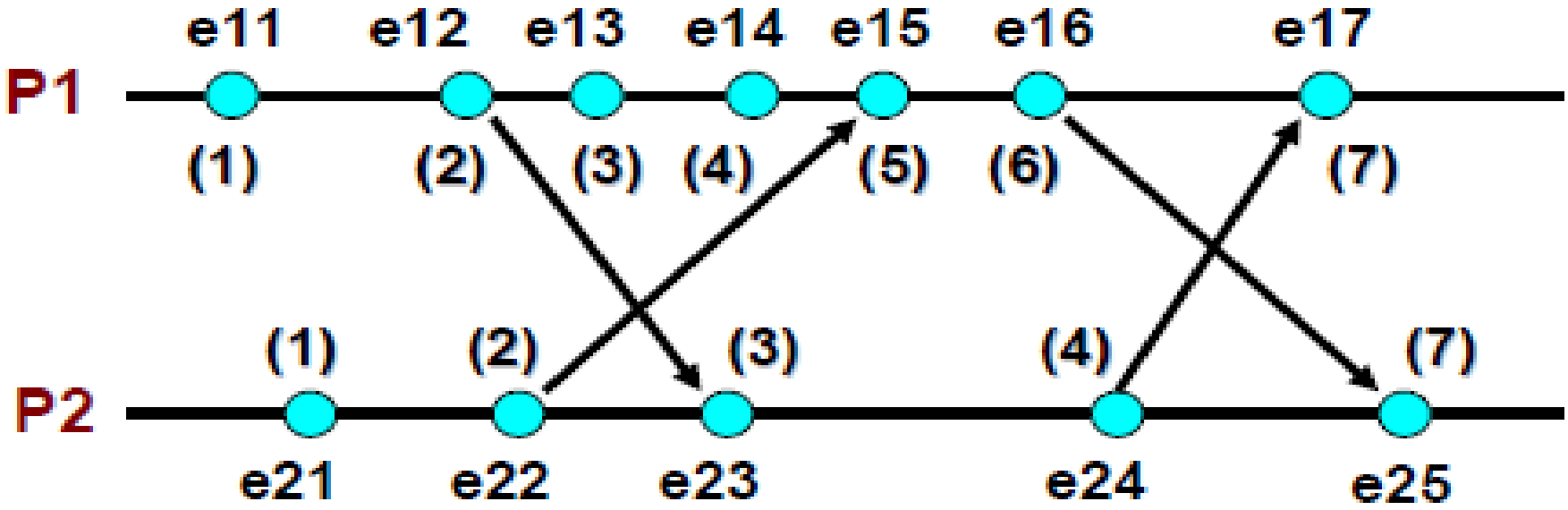
IR2: Ck := max(Ck , Cmsg + d) , (d>0) (Usually d=1)



Lamport's Logical Clock

IR 1: Ci := Ci + d , (d>0) , (usually d=1)

IR2: Ck := max(Ck , Cmsg + d) , (d>0) (Usually d=1)



Lamport's Logical Clock

- A **total order of events (" => ") can be obtained as follows:**
- If a is any event in process Pi , and b is any event in process Pk , then **a => b** if and only if either:
- **Ci (a) < Ck (b) or**
- **Ci (a) = Ck (b) and Pi << Pk (if scalar timestamps of a and b are equal, ordering of events need to done through some mechanism).**

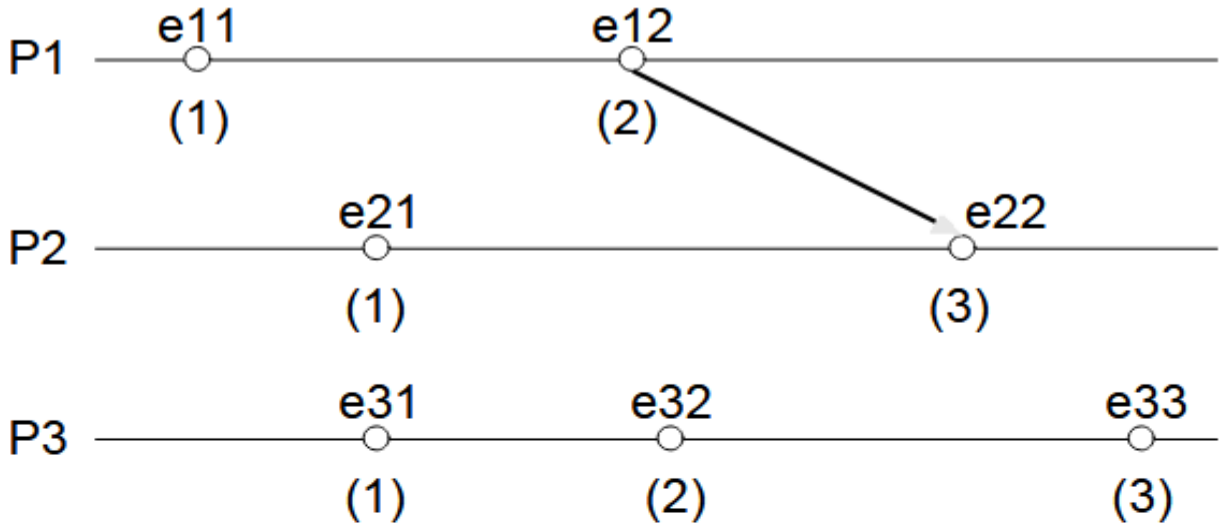
where "<<" denotes a relation that totally orders the processes. Process identifier is used to break ties.

Lower the process identifier in ranking higher the priority.

Limitations of Lamport's Logical Clock

- With Lamport's logical clocks,
- if a $\rightarrow$b, then C(a) < C(b)
- The following is **not necessarily true if events a and b occur in** different processes:
- if C(a) < C(b), then a $\rightarrow$ b **is NOT TRUE, if a and b events occur in two different processes.**
- **Scalar time is not strongly consistent.**

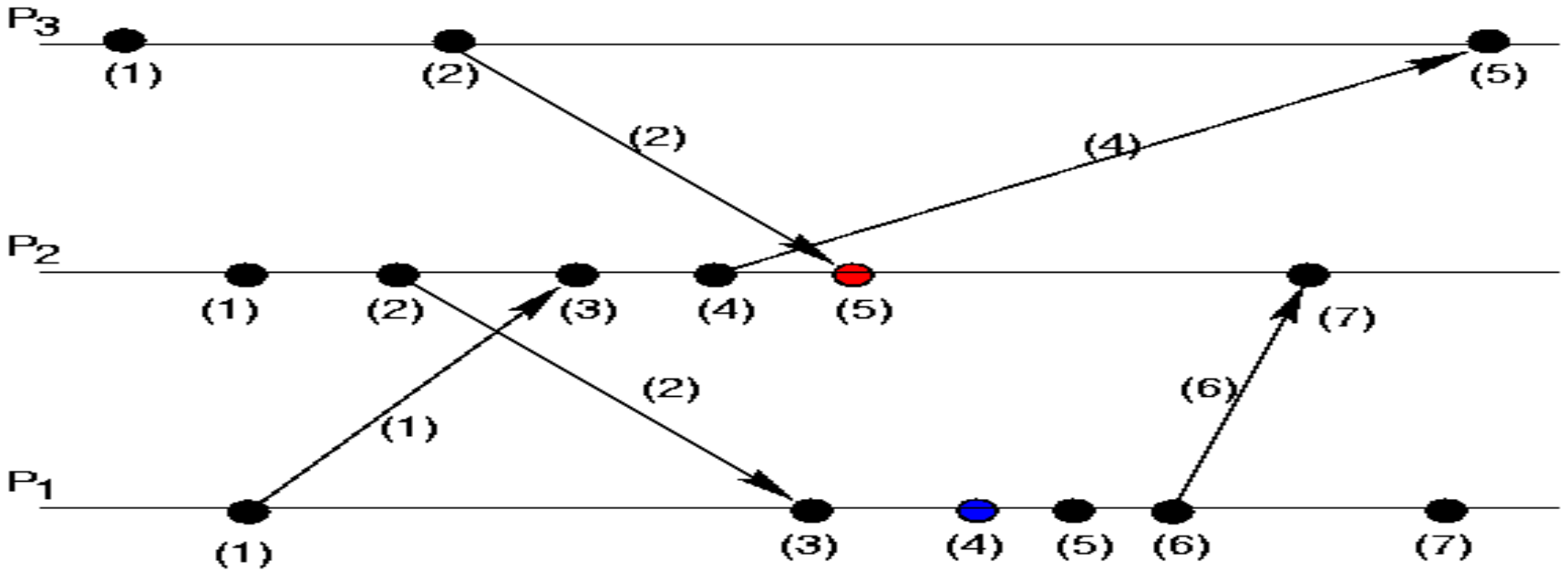
Limitations of Lamport's Logical Clock



- With Lamport's logical clocks, if $a \rightarrow b$, then $C(a) < C(b)$
  - The following is **not** necessarily true if events *a* and *b* occur in different processes: if $C(a) < C(b)$, then $a \rightarrow b$
  - $C(e11) < C(e22)$, and $e11 \rightarrow e22$ is true
  - $C(e11) < C(e32)$, but $e11 \rightarrow e32$ is false
- Cannot determine whether two events are causally related from timestamps

Limitations of Lamport's Logical Clock

There is drastic increase in clock values due to events that occur in different processes and that causally affects the other events. This is not captured in logical clocks.



Vector Clocks

Vector Clock

- Maintain a vector of values for every event that happens in all processes.
- Update happens for group of values in every event.

Vector Clock

- [IR1] Clock Ci must be incremented between any two successive events in process Pi :

  Ci [i]:= Ci [i] + d , (d>0, usually d=1)

- [IR2] If event a is the event of sending a message m in process Pi , then message m is assigned a vector timestamp tm =Ci (a) When that same message m is received by a different process Pk , Ck is updated as follows:

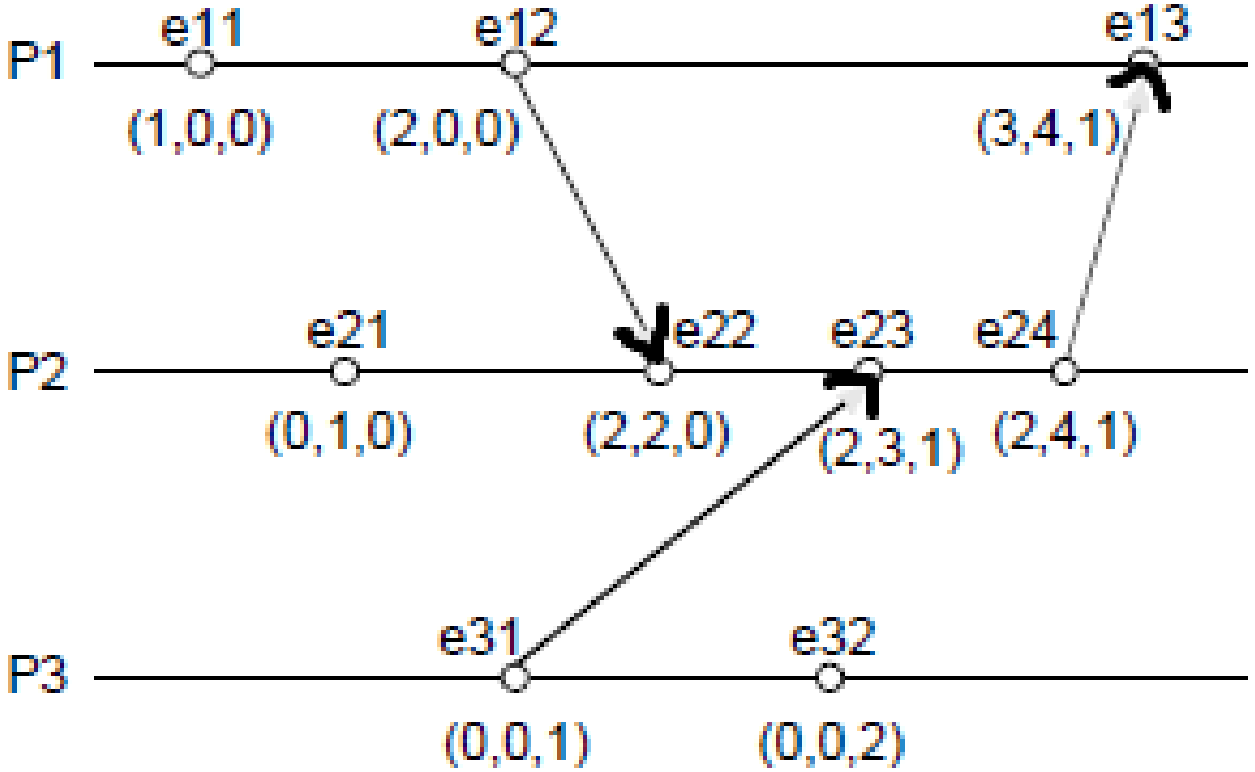
  ∀p, Ck [p]:= max(Ck [p], tm [p] + d) , (usually d=0 unless needed to model network delay)

- It can be shown that **∀ i,∀ k : Ci [i] > = Ck [i]**
- Rules for comparing timestamps can also be established so that
- if ta < tb , then a → b. Solves the problem with Lamport's clocks

Vector Clock

IR1: Ci [i]:= Ci [i] + d . (d=1)

IR2: ∀p, Ck [p]:= max(Ck [p], tm [p] + d) . (d=0)



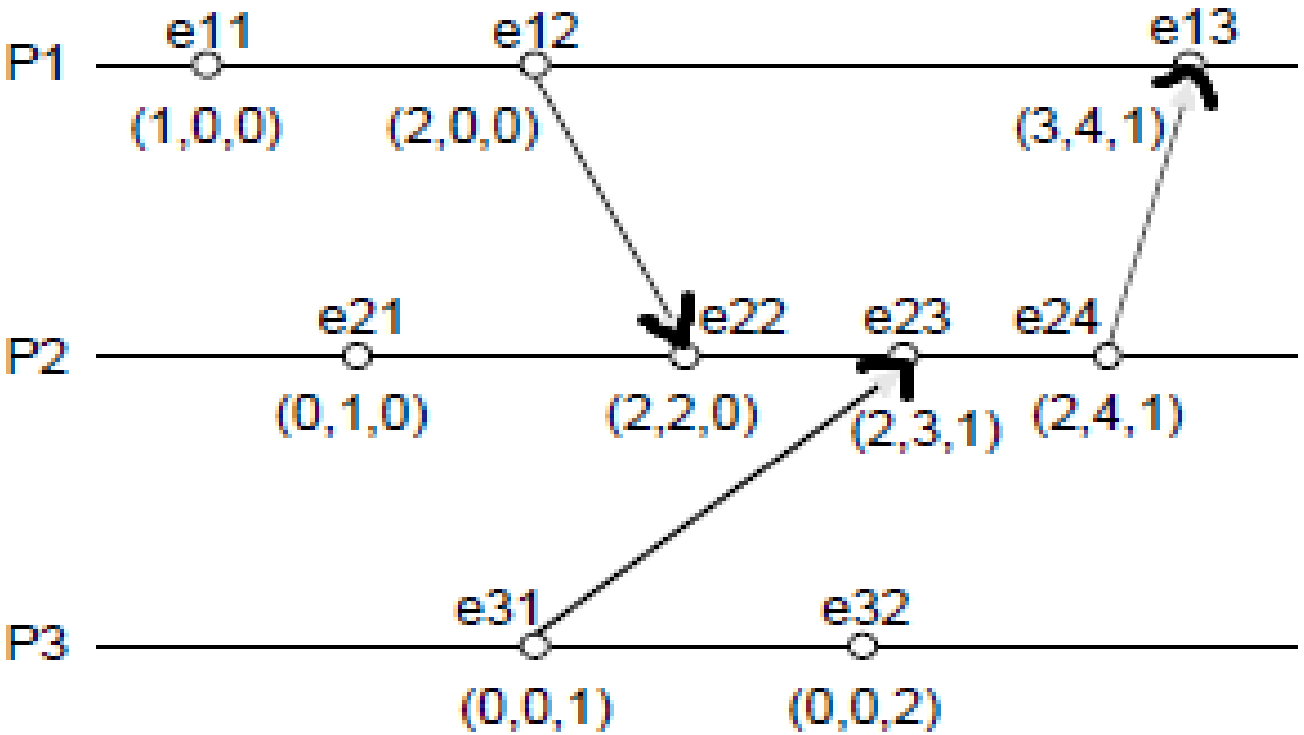
"e*nn*" is event;
"(*n*,*n*,*n*)" is clock value

Vector Clock – Solving Logical Clock Problem

if t(e31) < t(e23) then e31 → e23 is true.

if t(e12) < t(e24) then e12 → e24 is also true.

Concurrent events e32 | | e24; e32 ↛ e24 and e24 **↛** e32



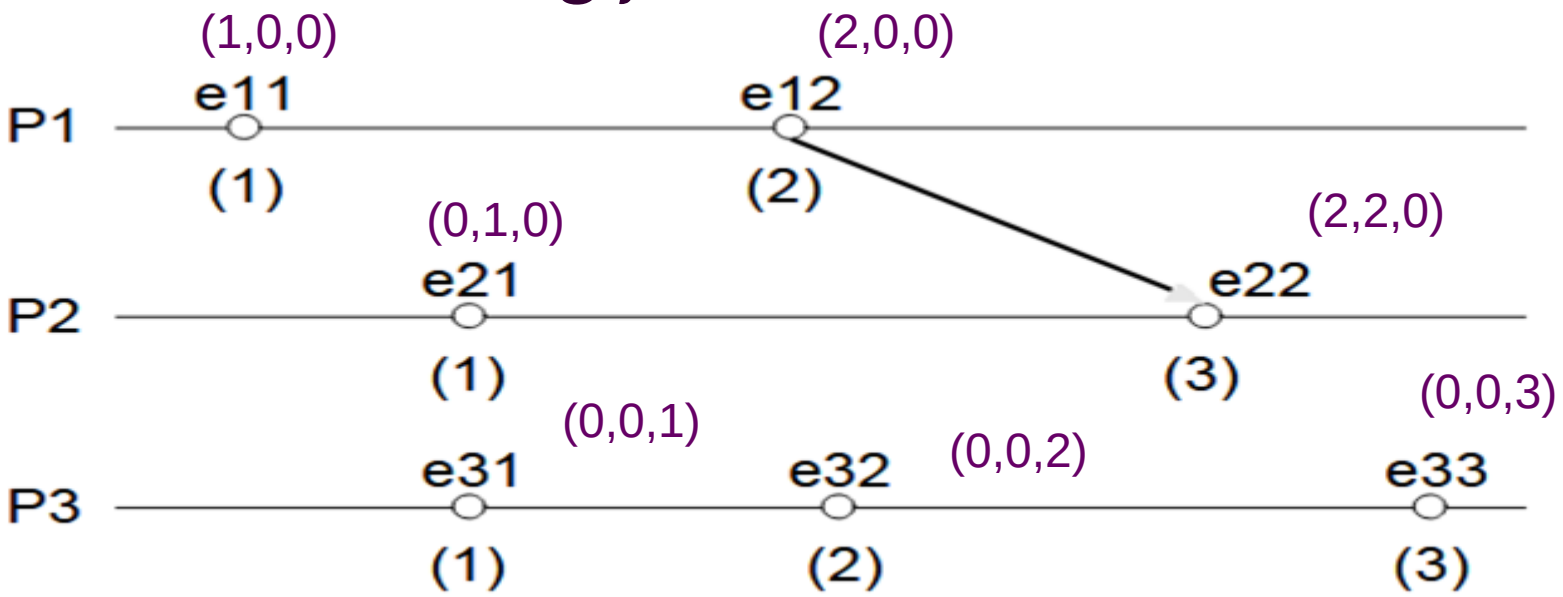
"e*nn*" is event; "(*n*,*n*,*n*)" is clock value

Vector Clock – Solving Logical Clock Problem

if C(e11) < C(e22) then e11 → e22 is true.

if C(e11) < C(e32) then e11 → e32 is also true.

Vector time is strongly consistent.



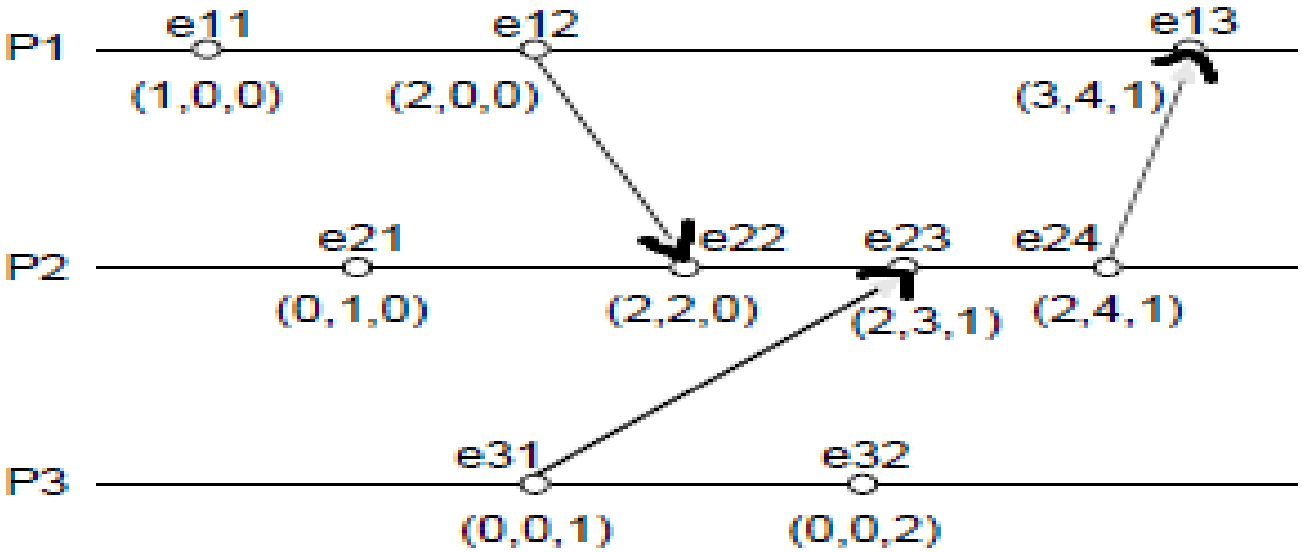
Vector Clock – Concurrent Events

Concurrent events e32 ||e24; e32 $\not\rightarrow$ e24 and e24 $\not\rightarrow$ e32

C(e32) = (0,0,2) and C(e24) = (2,4,1)

C(e32) is neither less nor grater than C(e24)

e11 || e21, e11||e31, e11||e32, e21 ||e32, e12 ||e32, e22 || e32, e32 ||e13.



"e*nn*" is event;
"(*n*,*n*,*n*)" is clock value

Vector Clock

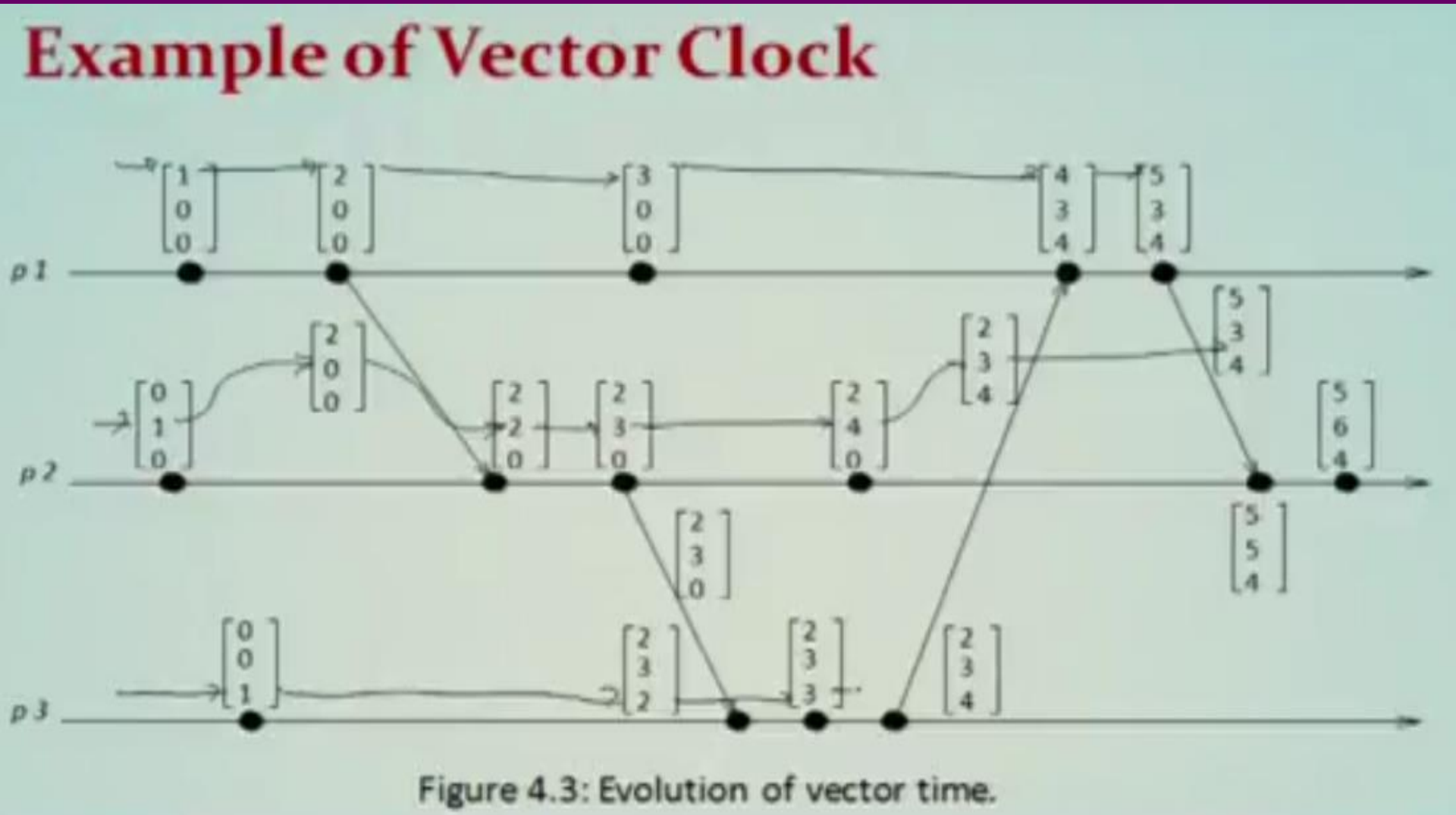


Figure 4.3: Evolution of vector time.

Drawbacks of Vector Clock

- Need to maintain memory units for tracking all events in all processes as a vector.
- The total numbers of processes may not be know in advance. Number of processes may be created or terminated at any time.
- Unnecessary wastage of maximum allocation of memory units.

Applications of Vector Clock

- **Distributed debugging**.
- Implementations of **causal ordering** communication and **causal distributed shared memory**.
- Establishment of **global breakpoints**.
- Determining the **consistency of checkpoints** in optimistic recovery.

Summary

- Physical Clocks
- Synchronizing Physical Clock (Algorithms)
- Problems with Physical Clock
- Lamport's Logical Clock
- Problems with Logical Clock
- Vector Clock
- Drawbacks of Vector Clocks
- Applications of Vector Clocks

References

1. Kshemkalyani, Ajay D., and Mukesh Singhal. Distributed computing: principles, algorithms, and systems. Cambridge University Press, 2011
2. Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems
3. George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012

Thank You

2M
Message-passing systems versus shared memory systems
Synchronous versus asynchronous executions

6M
Flynns taxonomy

10M
Design challenges
Algorithmic challenges

Youtube:

Cristian's Algorithm Physical clock synchronization in Distributed Systems
https://www.youtube.com/watch?v=9RUmRzAI3jA

Berkeley's Algorithm Physical clock synchronization
https://www.youtube.com/watch?v=8NW7-6Ea59Y

Network Time Protocol Physical Clock Synchronization Distributed Systems
https://www.youtube.com/watch?v=af9hZMiEe5U

Lamport logical clock | Distributed systems
https://www.youtube.com/watch?v=i1Z66AxbJrw&t=912s

Vector clock | Distributed Operating systems
https://www.youtube.com/watch?v=Ne4gdadd7gw

Causual ordering of messages | BSS Algorithm | Distributed OS
https://www.youtube.com/watch?v=XRkv77EhKUo