KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

CST 402 - DISTRIBUTED COMPUTING

Course Outcomes: After the completion of the course the student will be able to

CO1	Summarize various aspects of distributed computation model and logical time. **(Cognitive Knowledge Level: Understand)**
CO2	Illustrate election algorithm, global snapshot algorithm and termination detection algorithm. **(Cognitive Knowledge Level: Apply)**
CO3	Compare token based, non-token based and quorum based mutual exclusion algorithms. **(Cognitive Knowledge Level: Understand)**
CO4	Recognize the significance of deadlock detection and shared memory in distributed systems. **(Cognitive Knowledge Level: Understand)**
CO5	Explain the concepts of failure recovery and consensus. **(Cognitive Knowledge Level: Understand)**
CO6	Illustrate distributed file system architectures. **(Cognitive Knowledge Level: Understand)**

Text Books

1. Ajay D. Kshemkalyani and Mukesh Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, 2011.

Reference Books

1. George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. Distributed Systems: Concepts and Design, Addison Wesley, Fifth edition.
2. Kai Hwang, Geoffrey C Fox, Jack J Dongarra, Distributed and Cloud Computing – From Parallel Processing to the Internet of Things, Morgan Kaufmann Publishers, 2012.
3. Sukumar Ghosh, Distributed Systems: An Algorithmic Approach, CRC Press, Second edition, 2015.
4. Maarten Van Steen, Andrew S. Tanenbaum, Distributed Systems, Prentice Hall of India,Third edition, 2017.
5. Randy Chow and Theodore Johnson, Distributed Operating Systems and Algorithm Analysis, Pearson Education India, First edition, 2009.
6. Valmir C. Barbosa, An Introduction to Distributed Algorithms, MIT Press, 2003.

- Module – 1 (Distributed systems basics and Computation model)
- Module – 2 (Election algorithm, Global state and Termination detection)
- Module – 3 (Mutual exclusion and Deadlock detection)
- Module – 4 (Distributed shared memory and Failure recovery)
- Module – 5 (Consensus and Distributed file system)

Module1- Lesson Plan

- **L1:** Distributed System – Definition, Relation to computer system components
- **L2:** Primitives for distributed communication
- **L3:** Design issues, challenges and applications.
- **L4:** Design issues, challenges and applications.
- **L5:** A model of distributed computations – Distributed program, Model of distributed executions
- **L6:** Models of communication networks, Global state of a distributed
- **L7:** Cuts of a distributed computation, Past and future cones of an event, Models of process communications.

Distributed Systems

- A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.
- Collection of independent systems that work together to solve a problem or to accomplish a task.
- **Distributed system has been characterized in one of several ways:**
- Crash of a single machine never prevents from doing work
- A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system
- A collection of independent computers that appears to the users of the system as a single coherent computer
- A term that describes a wide range of computers, from weakly coupled systems such as wide-area networks, to strongly coupled systems such as local area networks.

Features of Distributed Systems

- **No common physical clock:**
- **No shared memory**
- **Geographical separation**
- **Autonomy and heterogeneity:** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system, cooperate with one another by offering services or solving a problem jointly.

Relation to computer system components

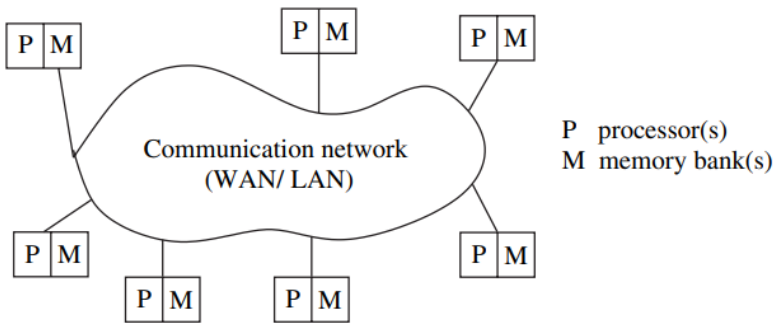


Figure 1.1 A distributed system connects processors by a communication network.

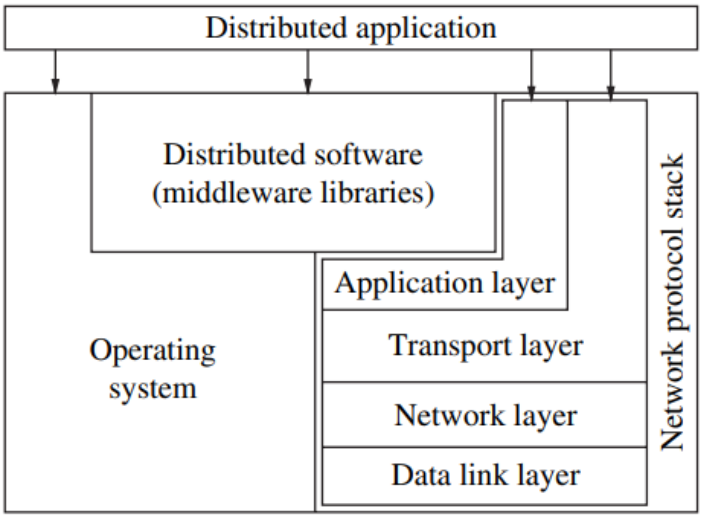


Figure 1.2 Interaction of the software components at each processor.

Relation to computer system components

- Each computer has a memory-processing unit and the computers are connected by a communication network.
- The distributed software is also termed as middleware. A distributed execution is the execution of processes across the distributed system to collaboratively achieve a common goal
- The distributed system uses a layered architecture to break down the complexity of system design.
- The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity

Motivation For Distributed Systems (Advantages)

- **Inherently distributed computations :** In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed
- **Resource sharing**
- **Access to geographically remote data and resources**
- **Enhanced reliability**

Reliability entails several aspects: availability, integrity, fault-tolerance,

- **Increased performance/cost ratio :** By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased

- **Scalability**
- **Modularity and incremental expandability**

Primitives for distributed communication

Communication Systems: 1. Message Passing , 2. Shared Memory

Blocking/non-blocking, synchronous/asynchronous primitives

- Send() and Receive() primitives are used to send and receive messages
- A Send primitive has at least two parameters – the destination, and the buffer in the user space, containing the data to be sent
- Receive primitive has at least two parameters –source from which the data is to be received, and the user buffer into which the data is to be received.
- There are two ways of sending data when the Send primitive is invoked –

1. the buffered option
2. the unbuffered option.

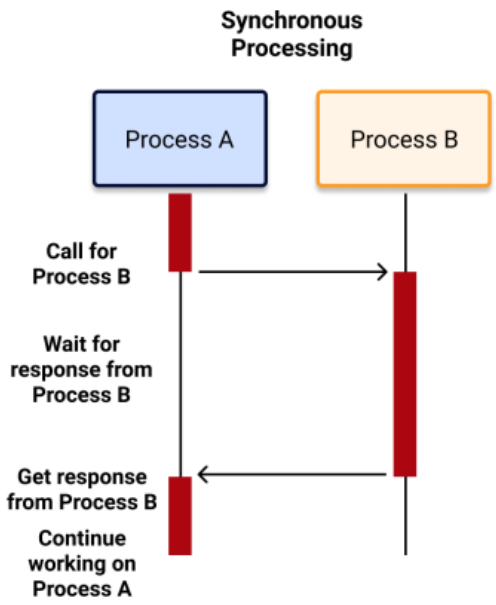
Primitives for distributed communication

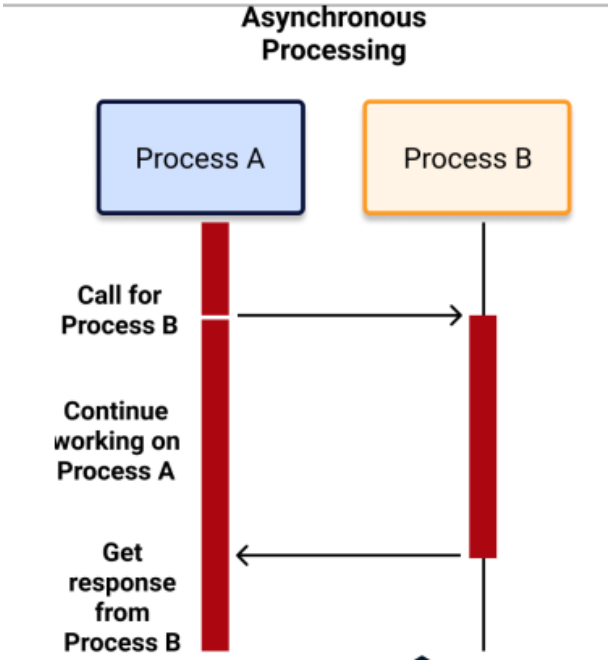
Synchronous primitives

A Send or a Receive primitive is synchronous if both the Send() and Receive() handshake with each other. The processing for the Send primitive completes only after the invoking processor learns that the other corresponding Receive primitive has also been invoked and that the receive operation has been completed

Asynchronous primitives

A Send primitive is said to be asynchronous if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer





Primitives for distributed communication

Blocking primitives : A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.

- **Blocking Send :** waits after sending message until its received by the receiver and acknowledged
- **Blocking Receiver :** Blocked until a message is received

Non-blocking primitives

A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.

Non Blocking Send : Continues after sending message

Non Blocking Receiver : Continues even if message is received or not

Primitives for distributed communication

Processor synchrony :

- Processor synchrony indicates that all the processors execute in lockstep with their clocks synchronized.
- As this synchrony is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a step, the processors are synchronized.
- This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have

completed executing the previous steps of code assigned to each of the processors.

Design issues and challenges

The following functions must be addressed when designing and building a distributed system:

1. **Communication**
2. **Processes**
3. **Naming**
4. **Synchronization**
5. **Data storage and access**
6. **Consistency and replication**
7. **Fault tolerance**
8. **Security**
9. **Applications Programming Interface (API) and transparency**
10. **Scalability and modularity**

Design issues and challenges

1. **Communication**

This task involves designing appropriate mechanisms for communication among the processes in the network. Some example mechanisms are:

remote procedure call (RPC), remote object invocation (ROI), message-oriented communication versus stream-oriented communication.

2. Processes

Some of the issues involved are: management of processes and threads at clients/servers; code migration; and the design of software and mobile agents.

3. Naming

Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a transparent and scalable manner.

Design issues and challenges

4. **Synchronization Mechanisms**

synchronization or coordination among the processes are essential.

Mutual exclusion is the classical example of synchronization,

In addition, synchronizing physical clocks, and devising logical clocks that capture the essence of the passage of time,

5. Data storage and access

Schemes for data storage, and implicitly for accessing the data in a fast and scalable manner across the network are important for efficiency.

Traditional issues such as file system design have to be reconsidered in the setting of a distributed system.

6. **Consistency and replication**

To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable.

7. Fault tolerance

Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes.

Process resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization are some of the mechanisms to provide fault-tolerance.

8. Security

Distributed systems security involves various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.

9. Applications Programming Interface (API) and transparency

Transparency deals with hiding the implementation policies from the user, and can be classified as follows

- **Access transparency** hides differences in data representation on different systems and provides uniform operations to access system resources.
- **Location transparency** makes the locations of resources transparent to the users.
- **Migration transparency** allows relocating resources without changing names.
- **Relocation transparency**: The ability to relocate the resources as they are being accessed is.
- **Replication transparency** does not let the user become aware of any replication.

- **Concurrency transparency** deals with masking the concurrent use of shared resources for the user.
- **Failure transparency** refers to the system being reliable and fault-tolerant.

10. **Scalability and modularity**

- The algorithms, data (objects), and services must be as distributed as possible.
- Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.
- **Design issues and challenges**
- **Algorithmic challenges in distributed computing**
- Designing useful execution models and frameworks
- Dynamic distributed graph algorithms and distributed routing algorithms
- Time and global state in a distributed system
- Synchronization/coordination mechanisms
- Group communication, multicast, and ordered message delivery
- Monitoring distributed events and predicates
- Distributed program design and verification tools
- Debugging distributed programs
- Data replication, consistency models, and caching.

Algorithmic challenges in distributed computing

1. **Designing useful execution models and frameworks**

The interleaving model and partial order model are two widely adopted models of distributed system executions. They have proved to be particularly useful for operational reasoning and the design of distributed algorithms.

2. Dynamic distributed graph algorithms and distributed routing algorithms

The distributed system is modeled as a distributed graph, and the graph algorithms form the building blocks for a large number of higher level communication, data dissemination, object location, and object search functions.

The algorithms need to deal with dynamically changing graph characteristics, such as to model varying link loads in a routing algorithm.

Algorithmic challenges in distributed computing

3. Time and global state in a distributed system

The challenges pertain to providing accurate physical time, and to providing a variant of time, called logical time Logical time is relative time, and eliminates the overheads of providing physical time for applications where physical time is not required.

4. Synchronization/coordination mechanisms

Synchronization is essential for the distributed processes to overcome the limited observation of the system state from the viewpoint of any one process.

Overcoming this limited observation is necessary for taking any actions that would impact other processes.

Algorithmic challenges in distributed computing

Problems Requiring Synchronization

- Physical clock synchronization
- Leader election - All the processes need to agree on which process will play the role of a distinguished process – called a leader process.
- Mutual exclusion
- Deadlock detection and resolution
- Termination detection
- **Garbage collection -** refers to objects that are no longer in use and that are not pointed to by any other process. Detecting garbage requires coordination among the processes

5. Group communication, multicast, and ordered message delivery

A group is a collection of processes that share a common context and collaborate on a common task within an application domain.

Algorithmic challenges in distributed computing

Specific algorithms need to be designed to enable efficient group communication and group management wherein processes can join and leave groups dynamically, or even fail.

When multiple processes send messages concurrently, different recipients may receive the messages in different orders, possibly violating the semantics of the distributed program.

Hence, formal specifications of the semantics of ordered delivery need to be formulated, and then implemented.

6. Monitoring distributed events and predicates

Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state, and are useful for applications such as debugging, sensing the environment, and in industrial process control.

On-line algorithms for monitoring such predicates are hence important.

8. Distributed program design and verification tools

Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering.

Designing mechanisms to achieve these design and verification goals is a challenge.

9. Debugging distributed programs

Debugging sequential programs is hard; debugging distributed programs is that much harder because of the concurrency in actions and the ensuing uncertainty due to the large number of possible executions defined by the interleaved concurrent actions.

Adequate debugging mechanisms and tools need to be designed to meet this challenge.

10. Data replication, consistency models, and caching

Fast access to data and other resources requires them to be replicated in the distributed system.

Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies.

Additionally, placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.

Applications of distributed computing and newer challenges

1. Mobile systems
2. Sensor networks
3. Ubiquitous or pervasive computing
4. Peer-to-peer computing
5. Publish-subscribe, content distribution, and multimedia
6. Distributed agents
7. Distributed data mining

8. Grid computing
9. Security in distributed system

1. **Mobile systems**

   Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium.

   the characteristics of communication are different;

   set of problems such as

i. routing,

ii. location management,

iii. channel allocation,

iv. localization and position estimation,

v. the overall management of mobility

vi. There are two popular architectures for a mobile network.

   1. **base-station approach,**

   also known as the cellular approach, wherein a cell which is the geographical region within range of a static but powerful base transmission station is associated with that base station

   2. **ad-hoc network approach** where there is no base station

   All responsibility for communication is distributed among the mobile nodes, wherein mobile nodes have to participate in routing by forwarding packets of other pairs of communicating nodes

2. Sensor networks

A sensor is a processor with an electro-mechanical interface that is capable of sensing physical parameters, such as temperature, velocity, pressure, humidity, and chemicals Sensors may be mobile or static;

sensors may communicate wirelessly, although they may also communicate across a wire when they are statically installed.

3. Ubiquitous or pervasive computing

The intelligent home, and the smart workplace are some example of ubiquitous environments Ubiquitous systems are essentially distributed systems;

recent advances in technology allow them to leverage wireless communication and sensor and actuator mechanisms .

4. Peer-to-peer computing

- Peer-to-peer (P2P) computing represents computing over an application layer network wherein all interactions among the processors are at a "peer" level, without any hierarchy among the processors.
- P2P computing arose as a paradigm shift from client–server computing where the roles among the processors are essentially asymmetrical.
- P2P networks are typically self-organizing, and may or may not have a regular structure to the network.

5. Publish-subscribe, content distribution, and multimedia

In a dynamic environment where the information constantly fluctuates

there needs to be:

i. an efficient mechanism for distributing this information (publish),

ii. an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information (subscribe),

iii. an efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter

6. Distributed agents

Agents collect and process information, and can exchange such information with other agents

Challenges in distributed agent systems include coordination mechanisms among the agents, controlling the mobility of the agents, and their software design and interfaces.

7. Distributed data mining

The data is necessarily distributed and cannot be collected in a single repository, massive to collect and process at a single repository in real-time.

8. Grid computing

Grid Computing is a subset of distributed computing, where a virtual supercomputer comprises machines on a network connected by some bus, mostly Ethernet or sometimes the Internet.

idle CPU cycles of machines connected to the network will be available to others

9. Security in distributed systems

The traditional challenges of security in a distributed setting include:

confidentiality (ensuring that only authorized processes can access certain information),

authentication (ensuring the source of received information and the identity of the sending process),

availability (maintaining allowed access to services despite malicious actions).

A model of distributed computations

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.

A distributed program :

- A distributed program is composed of a set of n asynchronous processes p1, p2, , pi, , pn that communicate by message passing over the communication network.
- Without loss of generality, we assume that each process is running on a different processor.
- The processes do not share a global memory and communicate solely by passing messages.
- The global state of a distributed computation is composed of the states of the processes and the communication channels
- Process execution and message transfer are asynchronous.
- 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,

1. internal events

2.message send events

3.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.

- 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

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

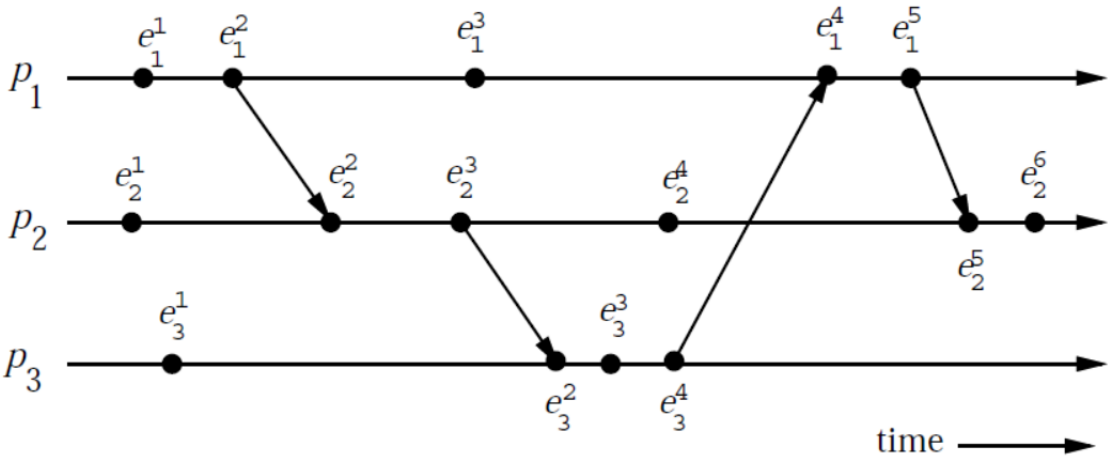
h_i is the set of events produced by p_i and
binary relation $\rightarrow_i$ defines a linear order on these events.

- Relation $\rightarrow_i$ expresses causal dependencies among the events of p_i.

- 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 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.

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 $\mathcal{H} = (H, \rightarrow)$.

The Causal Precedence relation is **related to the happened-before relation in message-based communication**.

. . . 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.)
- 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.

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.

- 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.

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

- 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,

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

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.

Figure 2.3: Illustration of cuts in a distributed execution.

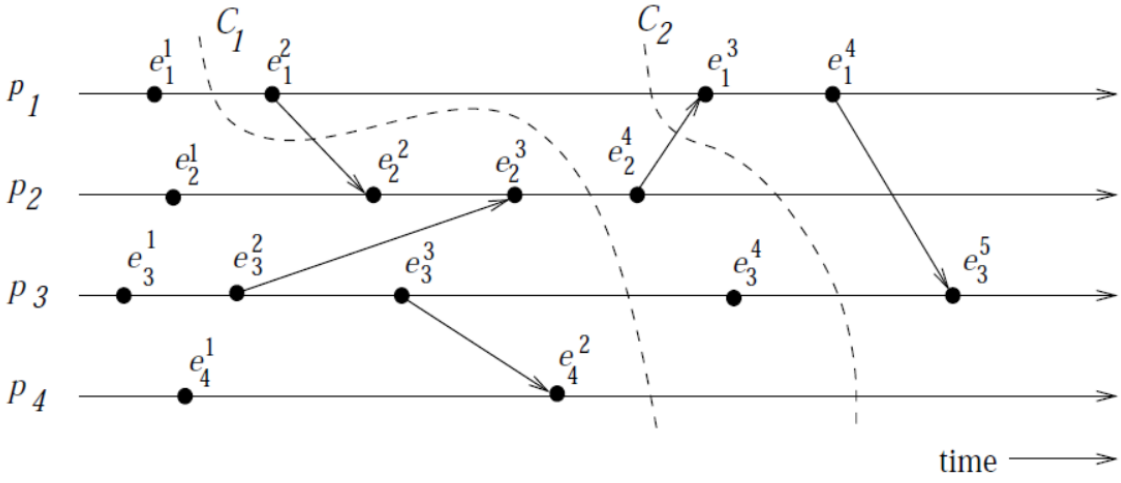
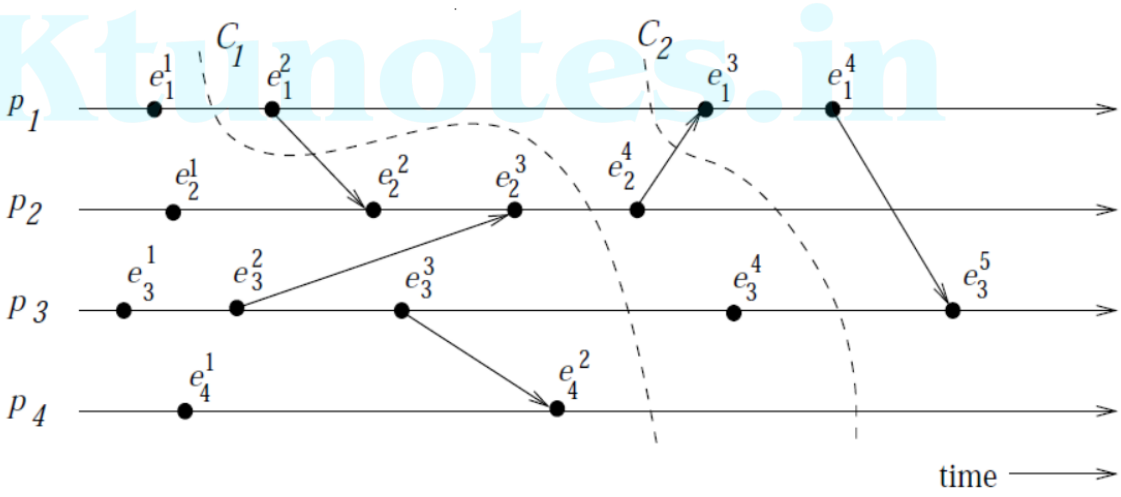


Figure 2.3: Illustration of cuts in a distributed execution.

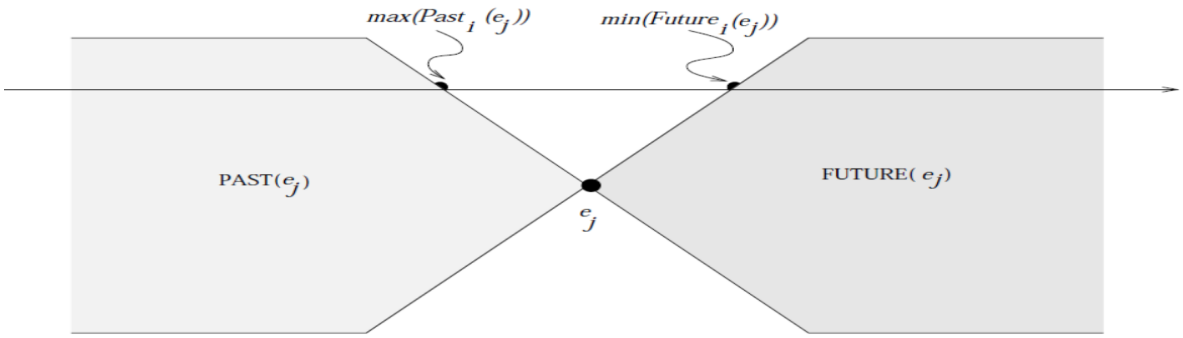


Past and Future Cones of an Event

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.

Figure 2.4: Illustration of past and future cones.



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:

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

- 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.

Load balancing

The goal of load balancing is to gain higher throughput, and reduce the userperceived latency.

Load balancing may be necessary because of a variety of factors such as high network traffic or high request rate causing the network connection to be a bottleneck, or high computational load

the objective is to service incoming client requests with the least turnaround time. S

The following are some forms of load balancing:

• Data migration The ability to move data (which may be replicated) around in the system, based on the access pattern of the users.

• Computation migration The ability to relocate processes in order to perform a redistribution of the workload.

• Distributed scheduling This achieves a better turnaround time for the users by using idle processing power in the system more efficiently.