University of Mumbai

Distributed Computing

(Course Code : CSC801) (Compulsory Subject)

Semester 8 - Computer Engineering

Strictly as per the New Syllabus (REV-2019 'C' Scheme) of
Mumbai University w.e.f. academic year 2022-2023

Dr. Nilesh Madhukar Patil

Ph.D. (Computer Engineering)

Associate Professor, Computer Engg. Department,

SVKM's D J Sanghvi College of Engineering, Mumbai

Dr. Pratik Kanani

Ph.D. (Computer Engineering)

Assistant Professor, Computer Engg. Department,

SVKM's D J Sanghvi College of Engineering, Mumbai

Mr. Aniket Kore

M.E. (Computer Engineering)

Assistant Professor, Computer Engg. Department,

SVKM's D J Sanghvi College of Engineering, Mumbai

TECH-NEO PUBLICATIONS

Where Authors Inspire Innovation

A Sachin Shah Venture

M8-78

MODULE 1

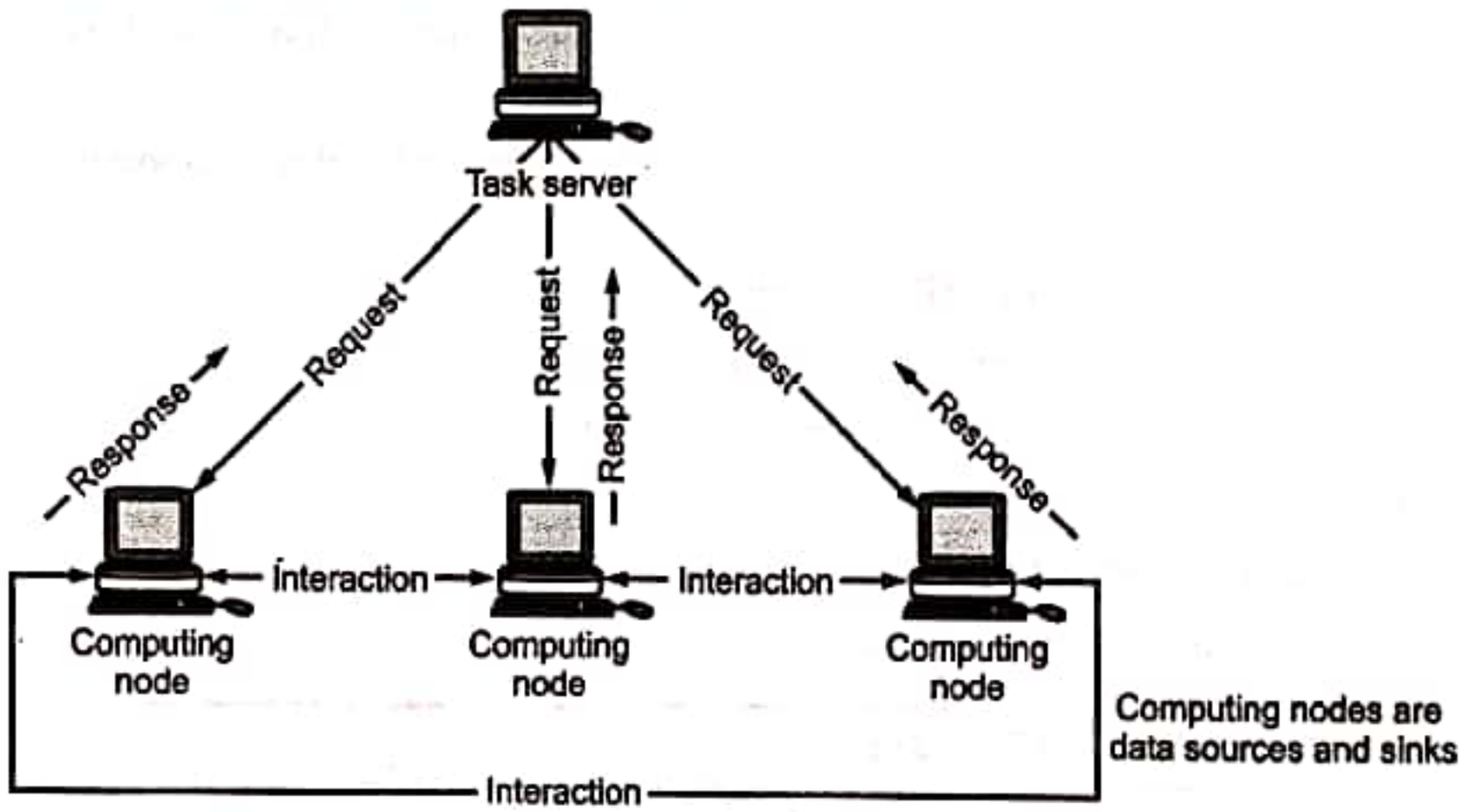
CHAPTER 1

Introduction to Distributed Systems

1.1 INTRODUCTION TO DISTRIBUTED SYSTEMS

GQ. Define Distributed System. State the advantages and disadvantages of a distributed system.

- A group of separate computers that are connected by a network and are able to work together on a job is known as a **distributed system.**
- In essence, it is a group of autonomous entities working together to address a challenge that neither of entity can handle alone.
- A distributed system is made up of several separate computers that work together to give users the impression of using a single computer.
- It is feasible to construct a single system made up of numerous computers and use it as a single consolidated system by employing high-performance computers connected by high-speed network.
- In such a system, many resources combine to provide the necessary processing speed, and the operating system is in charge of system upkeep and general maintenance.
- Computers in a distributed system are connected via a fast network rather than being in isolation.
- It implies that a large number of computers, whether workstations or desktop systems connected together, may do tasks normally performed by a high performance supercomputer.



(1A1)**Fig 1.1.1 : Architecture of Distributed Systems**

1.1.1 Characteristics of Distributed System

Distributed systems have the following characteristics :

(1) No standard physical clock.

(2) It has improved reliability.

(3) Improvement in the performance/cost ratio.

(4) Access to distant information and resources.

(5) Distributed systems are scalable.

1.1.2 Advantages of Distributed System

(1) Applications are by definition distributed when used in distributed systems.

(2) Users who are spread out geographically share information in distributed systems.

(3) Resource Exchange is possible that is Autonomous systems can share resources from remote locations.

(4) It is more flexible and offers a superior price-performance ratio.

(5) It responds faster and moves data more quickly.

1.1.3 Disadvantages of Distributed System

(1) Hardware and software requirements vary depending on the application.

(2) As resources are shared among several systems, there is a security issue caused by simple access to data.

(3) Data transport may be hampered by networking saturation; if there is a network latency, the user will have trouble getting the data.

(4) The database used by distributed systems is far more complicated and difficult to operate than the database used by a single user system.

1.1.4 Applications Area of Distributed System

(1) Finance and commerce are the focus of distributed systems, including Amazon, eBay, online banking, and e-commerce websites.

(2) Search engines, Wikipedia, social media, and cloud computing are examples of the information society which uses Distributed systems in the background.

(3) AWS, Salesforce, Microsoft Azure, and SAP all use cloud technologies.

(4) YouTube, online gaming, and music provide entertainment.

(5) Healthcare : Health Informatics and online patient records.

(6) E-learning for education.

(7) Logistics and transportation: GPS, Google Maps.

(8) Management of the environment: sensor technologies.

1.2 ISSUES IN DISTRIBUTED SYSTEM

UQ. Explain different issues and goal related to design of Distributed System. Explain transparency in detail. **(MU - May 22)**

UQ. What are the issues in designing a distributed system? **(MU - May 16)**

UQ. What are the common issues with which the designer of a heterogeneous distributed system must deal? **(MU - May-17)**

The following are the issues in distributed systems :

(1) **Heterogeneity :** Networks, computer hardware, operating systems, and developer implementations are all examples of heterogeneous systems. Middleware is a crucial element of the client-server heterogeneous distributed system environment. A set of services known as middleware allows users and applications to communicate with one another across a diverse distributed system.

(2) **Openness :** The degree to which new resource-sharing services may be made accessible to users determines the distributed system's openness.

(3) **Scalability :** The system should be scalable even when the number of users and linked resources significantly grows.

(4) **Security** : High degrees of encryption are required to secure shared resources and sensitive data since data is shared across various resources.

(5) **Failure handling** : Corrective steps should be put in place to deal with this situation because when hardware or software failures happen, they may provide inaccurate results or cease before finishing the planned calculation.

(6) **Concurrency :** It's possible for several clients to try to access a single shared resource at once. The same read, write, and update of resources are requested by several users. Any object in a distributed system that represents a shared resource must make sure that it functions properly in a multi-threaded setting.

(7) **Transparency :** Transparency makes sure that users or application programmers see the distributed system as a single, collaborating system rather than a group of independent systems. The user should not be aware of the location of the services, and moving data from a local workstation to a distant one should be seamless.

1.2.1 Types of Transparencies

GQ. Explain different types of transparencies in distributed system.

The implementation of the distributed system is very complex, as a number of issues have to be considered to achieve its final objective. The complexities should not worry the user of the distributed system from using it i.e., the complexities should be hidden from the user who uses the distributed system. This property of the distributed system is called its transparency.There are different kinds of transparencies that the distributed system has to incorporate. The following are the different transparencies encountered in the distributed systems.

(1) Access Transparency	(2) Location Transparency
(3) Concurrency Transparency	(4) Replication Transparency
(5) Failure Transparency	(6) Migration Transparency
(7) Performance Transparency	(8) Scaling Transparency

(1) Access Transparency

- Clients should be unaware of the distribution of the files. The files could be present on a totally different set of servers which are physically distant apart and a single set of operations should be provided to access these remote as well as the local files.
- Applications written for the local file should be able to be executed even for the remote files. The examples illustrating this property are the File system in Network File System (NFS), SQL queries, and Navigation of the web.

(2) Location Transparency

- Clients should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames. A location transparent name contains no information about the named object's physical location.
- This property is important to support the movement of the resources and the availability of services. The location and access transparencies together are sometimes referred as Network transparency. The examples are File system in NFS and the pages of the web.

(3) Concurrency Transparency

- Users and Applications should be able to access shared data or objects without interference between each other. This

requires very complex mechanisms in a distributed system, since there exists true concurrency rather than the simulated concurrency of a central system. The shared objects are accessed simultaneously.

- The concurrency control and its implementation is a hard task. The examples are NFS, Automatic Teller machine (ATM) network.

(4) Replication Transparency

- This kind of transparency should be mainly incorporated for the distributed file systems, which replicate the data at two or more sites for more reliability.
- The client generally should not be aware that a replicated copy of the data exists. The clients should also expect operations to return only one set of values. The examples are Distributed DBMS and Mirroring of Web pages.

(5) Failure Transparency

- Enables the concealment of faults, allowing user and application programs to complete their tasks despite the failure of hardware or software components. Fault tolerance is provided by the mechanisms that relate to access transparency.
- The distributed system is more prone to failures as any of the component may fail which may lead to degraded service or the total absence of that service. As the intricacies are hidden the distinction between a failed and a slow running process is difficult. Examples are Database Management Systems.

(6) Migration Transparency

- This transparency allows the user to be unaware of the movement of information or processes within a system without affecting the operations of the users and the applications that are running.
- This mechanism allows for the load balancing of any particular client, which might be overloaded. The systems that implement this transparency are NFS and Web pages.

(7) Performance Transparency

Allows the system to be reconfigured to improve the performance as the load varies.

(8) Scaling Transparency

- A system should be able to grow without affecting application algorithms. Graceful growth and evolution is an important requirement for most enterprises.
- A system should also be capable of scaling down to small environments where required, and be space and/or time efficient as required. The best-distributed system example implementing this transparency is the World Wide Web.

1.3 GOALS OF DISTRIBUTED SYSTEMS

UQ. State the goals of distributed system. **(MU - Dec. 16, Dec. 19)**

A number of important goals must be met in order for the effort involved in developing a distributed system to be successful. A distributed system should be open, scalable, and easy for users to access resources. The objective is to do the following at the maximum level possible.

(1) **Bringing Users and Resources Together** : A distributed system's major objective is to give users easy access to remote resources and to share those resources in a regulated way with other users.

(2) **Transparency** : Transparency indicates that a distributed system may seem to users and applications as if it were a single computer system.

(3) **Openness** : Distributed systems must work to be as open as possible. An open distributed system is one that offers services in accordance with standards that outline the syntax and semantics of those service instances.

(4) **Scalability** : The general trend in distributed systems is larger systems. Algorithms that work well for systems with 1000 machines can be used by systems with 100 machines, but cannot be used at all by systems with 10,000 machines. To begin with, the centralized algorithm does not scale well.

(5) **Reliability** : The main goal was to increase the dependability of distributed systems relative to single processor systems. According to the concept, when one machine malfunctions, another one adjusts to it.

(6) **Performance** : If a distributed system cannot achieve the desired performance, it is pointless to develop it as transparent, flexible, and reliable. A distributed system ought to offer the highest level of performance.

1.4 TYPES OF DISTRIBUTED SYSTEM

GQ. Explain different types of distributed systems with neat diagram.

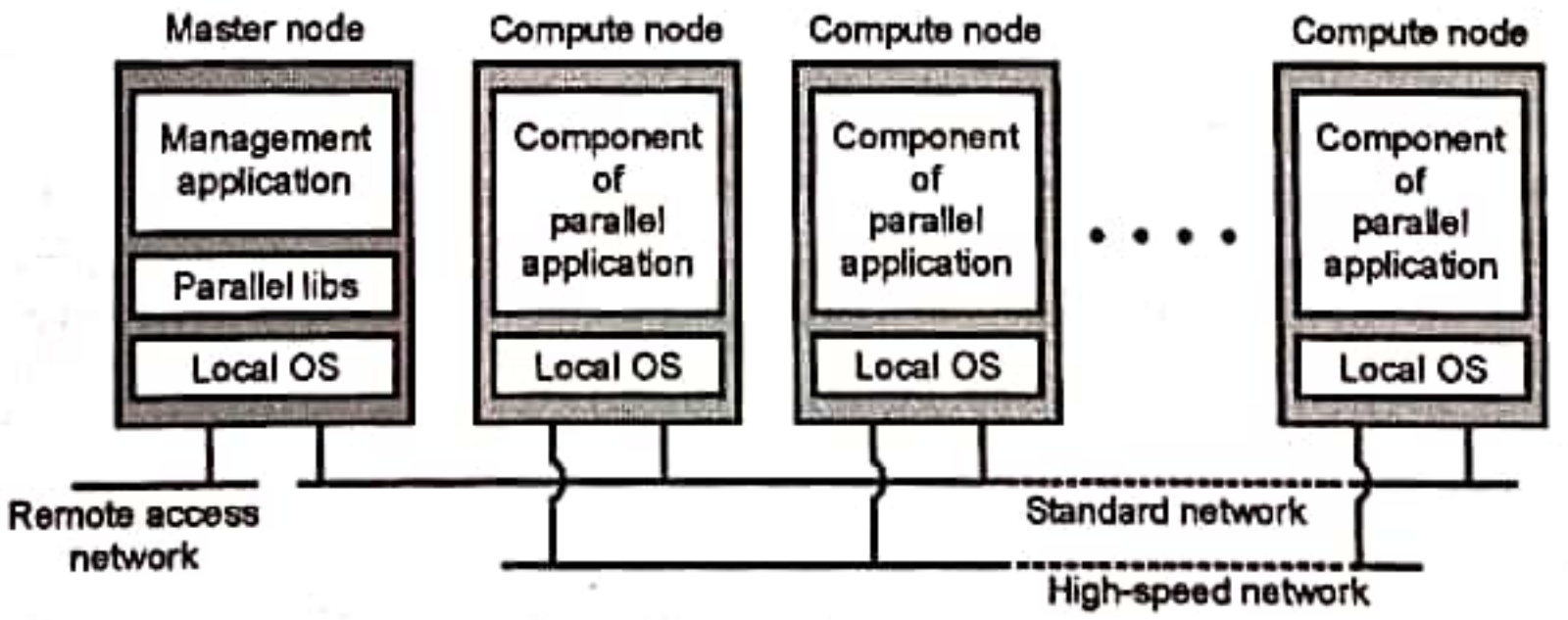
Before delving into the concepts of distributed systems, let us first examine the various types of distributed systems.

1.4.1 Distributed Computing Systems

- These distributed computing systems are designed to provide high computing performance to meet the demands of real-time applications.
- In cluster computing the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network.
- In addition, each node runs the same operating system. The situation becomes quite different in the case of grid computing.
- This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.

1.4.1(A) Cluster Computing

- Cluster computing is a type of distributed computing that involves the use of a cluster, which is a group of interconnected computers that work together as a single system.
- In a cluster computing system as shown in Fig. 1.4.1, the computers in the cluster are connected together by a high-speed network, and each computer has its own processing power, memory, and storage.
- The main advantage of cluster computing is that it allows for the parallel processing of large amounts of data and the execution of complex applications that require a lot of computational power.
- By dividing the workload among the computers in the cluster, a cluster computing system can achieve better performance and scalability than a single computer.

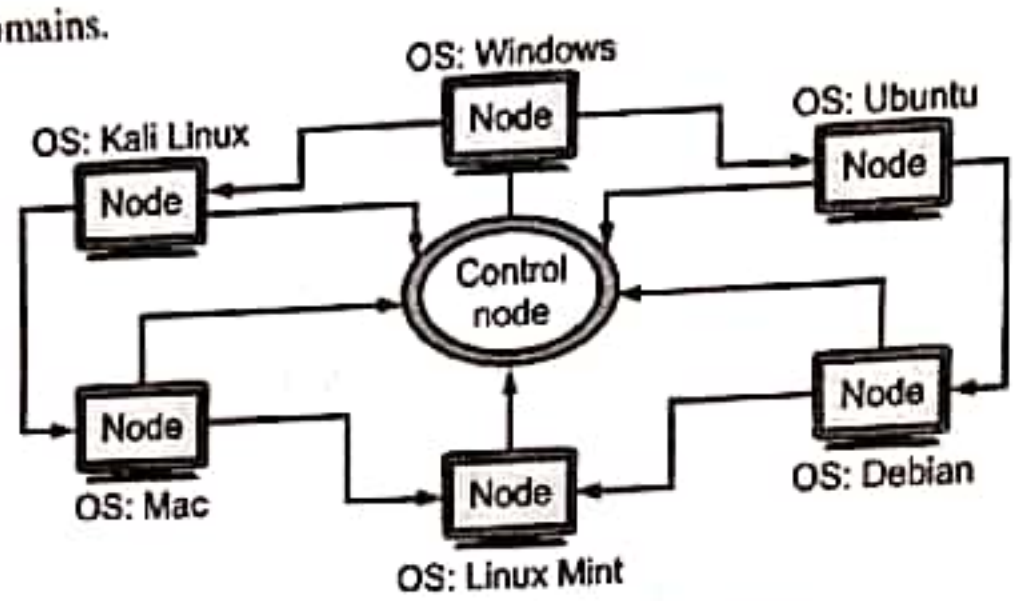


(1A2)**Fig. 1.4.1 : Cluster Computing System**

- The primary benefit of cluster computing is that it makes it possible to run complicated programs that need a lot of processing power and analyse massive volumes of data in simultaneously.
- A cluster computing system may perform and scale better than a single computer by distributing the workload across the machines in the cluster.

1.4.1(B) Grid Computing

- A key issue in a grid computing system is that resources from different organizations are brought together to allow the collaboration of a group of people or institutions.
- Typically, resources consist of compute servers, storage facilities, and databases.
- Given its nature, much of the software for realizing grid computing evolves around providing access to resources from different administrative domains.



(1A3)Fig. 1.4.2 : Grid Computing System

1.4.2 Distributed Information Systems

- Another major type of distributed systems may be found in businesses that have been presented with a plethora of networked applications yet have found interoperability to be an unpleasant experience.
- Many current middleware solutions are the consequence of working with an architecture that made integrating apps into enterprise-wide information systems simpler.
- In the following section, we concentrate on following two forms of distributed information systems.

1.4.2(A) Transaction Processing Systems

- In practise, database operations are often carried out in the form of transactions.
- There may be primitives in a mail system for sending, receiving, and forwarding mail.
- They might be substantially different in an accounting system. However, READ and WRITE are common instances.
- Ordinary statements, procedure calls, and the like are likewise permitted inside a transaction.
- The characteristic feature of a transaction is either all of these operations are executed or none are executed.
- These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation. This all-or-nothing property of transactions is one of the four characteristic properties that transactions have.
- More specifically, transactions are :
  (1) **Atomic :** To the outside world, the transaction happens indivisibly.
  (2) **Consistent :** The transaction does not violate system invariants.
  (3) **Isolated :** Concurrent transactions do not interfere with each other.
  (4) **Durable :** Once a transaction commits, the changes are permanent.

1.4.2(B) Enterprise Application Integration

- Application components, in particular, should be able to interact directly with one another rather than relying on the request/reply behaviour allowed by transaction processing systems.
- The main idea was that existing applications could directly exchange information.
- An application component may successfully submit a request to another application component via remote procedure calls (RPC) by performing a local procedure call, which results in the request being packaged as a message and transmitted to the callee.
- Similarly, as a consequence of the procedure call, the result will be transmitted back and returned to the application.
- As object technology grew in prominence, ways for calling remote objects were created, resulting in what is known as remote method invocations (RMI).
- An RMI is similar to an RPC in that it acts on objects rather than applications. RPC and RMI have the drawback of requiring both the caller and the callee to be active at the time of the communication.

1.4.3 Distributed Pervasive Systems

- The devices in the distributed pervasive systems are frequently tiny, battery-powered, transportable, and have just a wireless connection, however not all of these features apply to all devices.
- Furthermore, these traits do not have to be interpreted as restricted.
- It is the next step in integrating common things with microprocessors so that this data can interact.
- These systems are built to distribute data, processes, and control.
- Few examples of pervasive systems are :

(i) **Home System** : Many household equipments are now digital, allowing us to operate them from anywhere and at any time.

(ii) **Electronic Health Care Systems** : Smart medical wearable gadgets are already available, allowing us to frequently check our health.

(iii) **Sensor Networks** : A sensor network is often made up of tens to hundreds or thousands of small nodes, each of which is equipped with a sensing device. Most sensor networks communicate wirelessly, and the nodes are often powered by batteries. Because of their limited resources, limited communication capabilities, and limited power consumption, high efficiency is on the list of design objectives.

1.5 DISTRIBUTED SYSTEM MODELS

UQ What are various system models of distributed system? **(MU - Dec-18)**

Basically distributed system models are categorized as follows :

1.5.1 Physical Model

- A physical model represents the underlying hardware pieces of a distributed system that abstracts from the features of the computer and networking technologies used.
- The physical model may be divided into three generations of distributed systems: early distributed systems, Internet-scale distributed systems and contemporary distributed systems.
- Hardware and software components can communicate and coordinate their operations by sending and receiving messages.

1.5.1(A) Early distributed systems

- Emerged in the late 1970s and early 1980s as a result of the introduction of local area networking technology.
- The system is generally comprised of 10 to 100 nodes connected by a LAN, with limited Internet access and services (e.g., shared local printer, file servers)

1.5.1(B) Internet-scale distributed systems

- It arose in the 1990s as a result of the expansion of the Internet.
- It includes a large number of nodes from many organizations, and it has increased heterogeneity, a greater emphasis on open standards and services, and associated middleware technologies such as CORBA and Web Services.

1.5.1(C) Contemporary distributed systems

- The emergence of mobile computing leads to nodes that are location-independent nodes.
- These nodes should have additional features such as service discovery and support for ad hoc interconnectivity.
- It aided in the development of cloud computing and ubiquitous computing.
- **Cloud computing** is a collection of independent nodes that perform a specific role to pools of nodes that work together to deliver certain high-performance services.
- **Ubiquitous computing** has resulted in the transition from discrete nodes to systems in which computers are embedded in everyday objects in an environment.

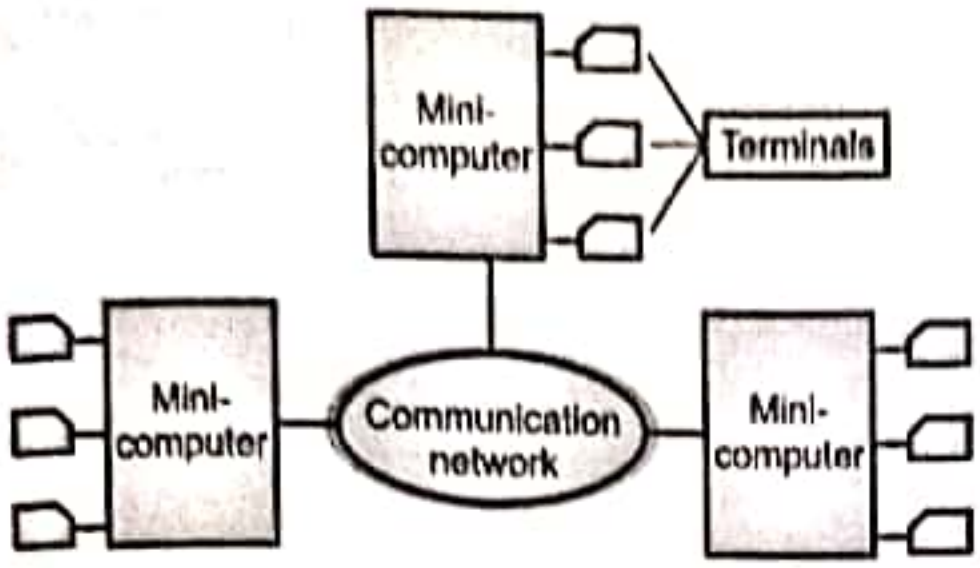
1.5.1(D) Types of Physical Models Based on Hardware and Computation

(1) Minicomputer Model

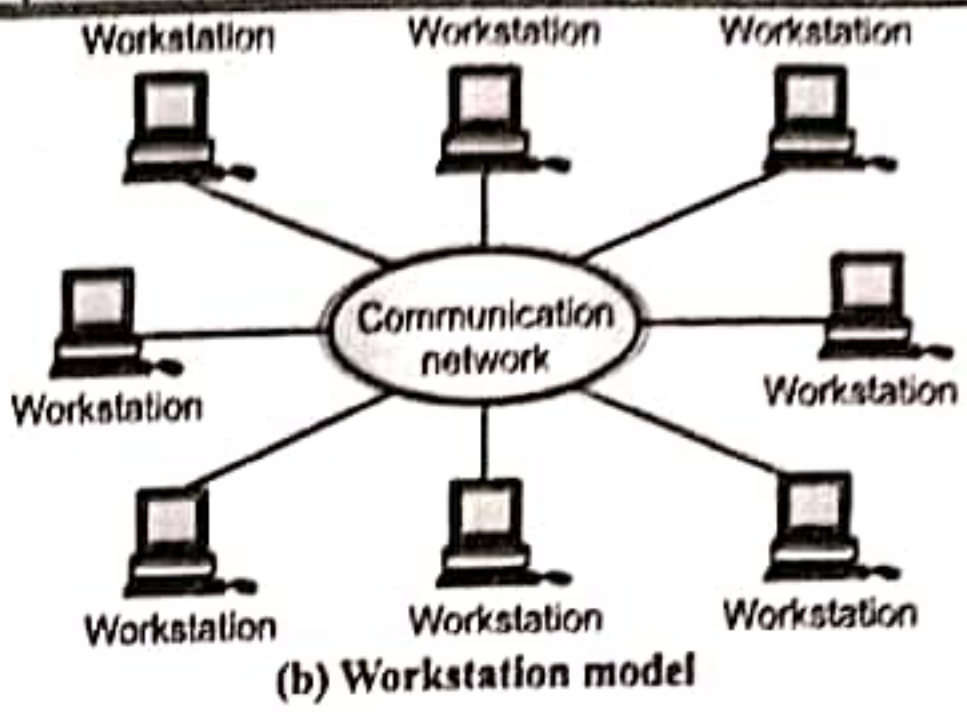
- Minicomputer models are simple extensions of centralized time-sharing systems.
- This approach uses a communication network to connect a few minicomputers or supercomputers. Multi-user minicomputers are common.
- Each minicomputer has multiple interactive terminals. Each user logs into one minicomputer with remote access to others.
- The network lets users access remote resources on other machines.
- When remote users want to share resources like information databases on various machines, the minicomputer paradigm might be employed.
- Minicomputer-based distributed computing systems include early ARPAnet.

(2) Workstation Model

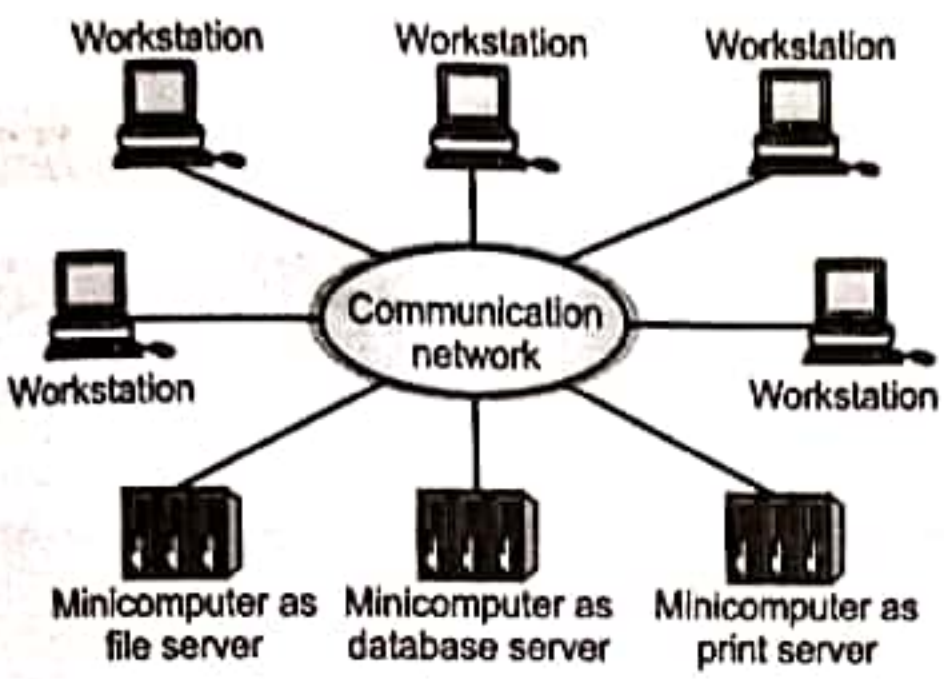
- Multiple workstations connected by a network make up a workstation-based distributed computing system.
- Each workstation in a building or campus may be a single-user computer with its own disk.
- In such a setup, many workstations are idle at any given time, wasting CPU time.
- Therefore, the workstation model connects all these workstations via a high-speed LAN so that idle workstations can process the jobs of users who are signed onto other workstations and don't have enough processing power at their own workstations.



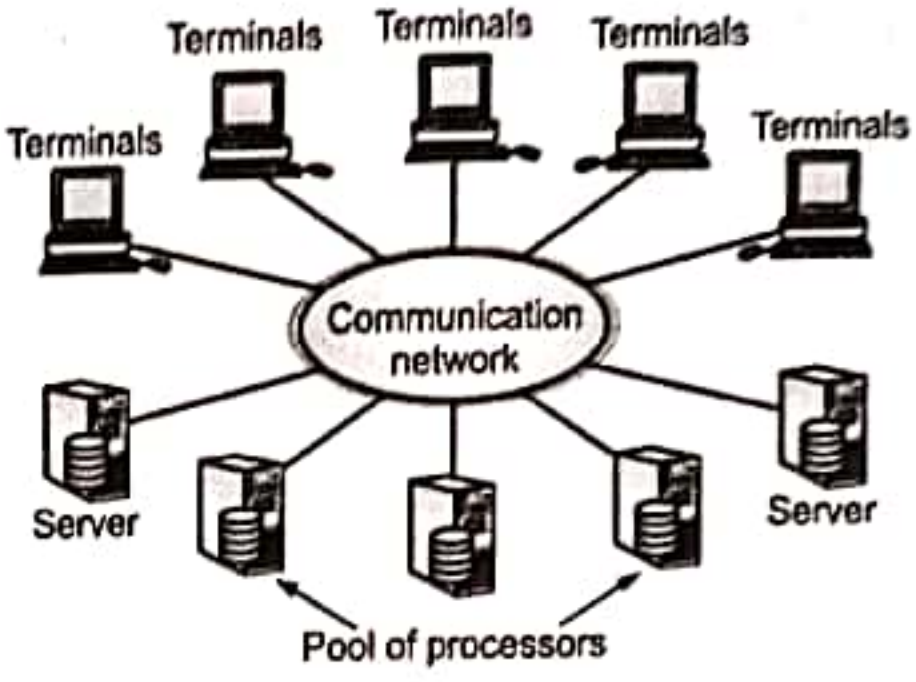
(a) Minicomputer model



(b) Workstation model



(c) Workstation-server model



(d) Processor-pool model

(1A4)Fig. 1.5.1 : Distributed System Physical Models

(3) Workstation-Server Model

- A distributed computing system based on the workstation-server architecture has a few minicomputers and several workstations (most of which are diskless, but some may be diskful) connected via a communication network.
- Diskless workstations on a network must use a file system implemented by a diskful workstation or a minicomputer with a disc.
- One or more minicomputers implement the file system. Other minicomputers can offer database and print functions. Thus, each minicomputer serves many services as a server.
- In the workstation-server approach, there are workstations and specialized computers (maybe workstations) that run server programs (called servers) to manage and provide access to shared resources.
- Multiple servers manage a type of resource in a distributed computing system for reliability and scalability.
- In this model, users log onto their home workstations. The user's home workstation does normal computing activities, but requests for special servers (such as file or database servers) are routed to a server that executes the user's requested activity and sends the output to the user's workstation.
- In this paradigm, user processes do not need to migrate to server machines to complete their task.

(4) Processor-Pool Model

- The processor-pool model is based on the fact that users rarely need computational power but occasionally need a lot of it for a short time (e.g., when recompiling a program consisting of a large number of files after changing a basic shared declaration).

- Thus, unlike the workstation-server model, the processor-pool concept pools CPUs for users to share as needed.
- Many networked microcomputers and minicomputers form the processor pool.
- Each pool processor has memory to load and run a distributed computing system or application program.

(5) Hybrid Model

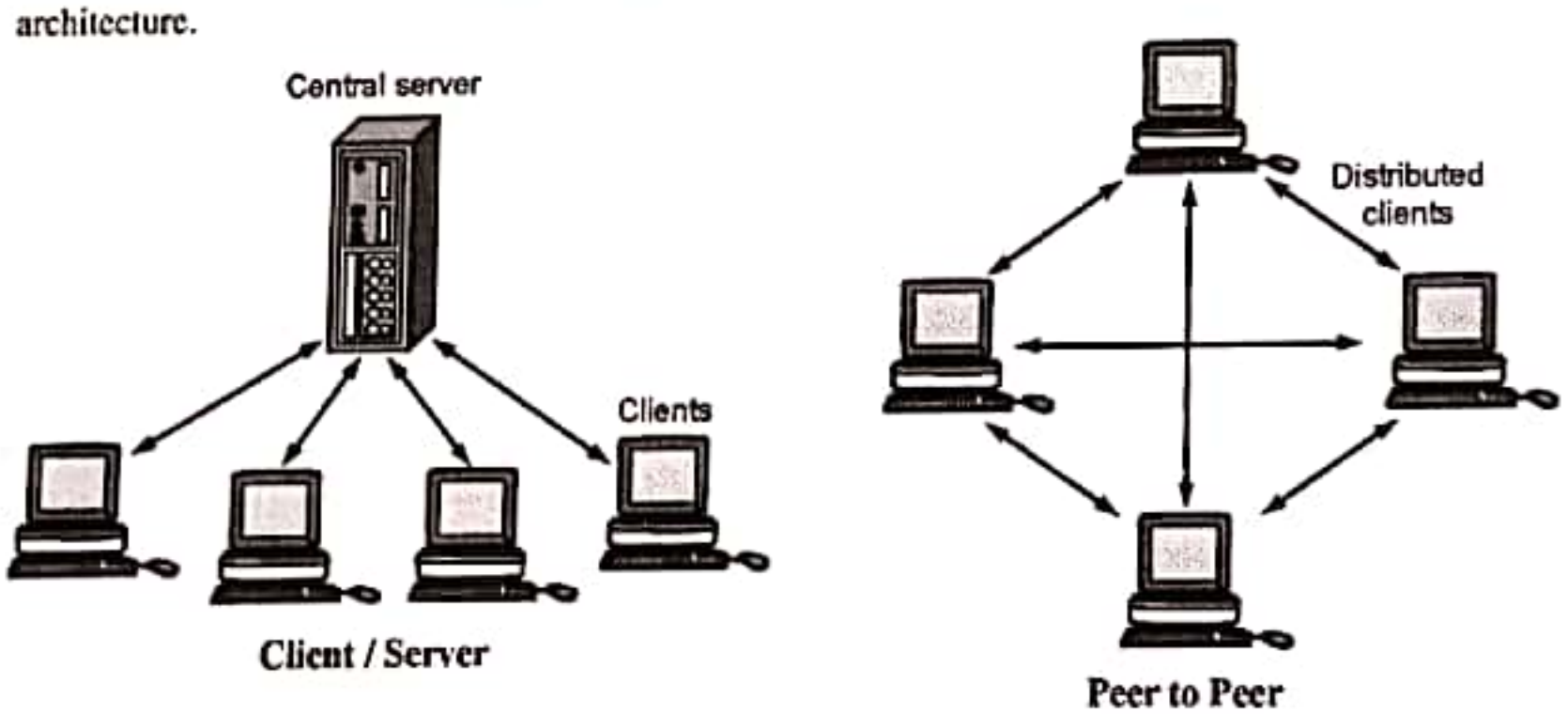
- Most distributed computing systems are built using the workstation-server model.
- Most computer users merely edit documents, send emails, and run minor programs.
- Simple usage suits the workstation-server approach.
- The processor-pool model is better in a workplace with groups of users that often do big computing.

1.5.2 Architectural Model

UQ. Explain software models supported by the distributed system. **(MU - May 18)**

UQ. Discuss in brief the different architectural models in Distributed System. Explain with neat diagram. **(MU - Dec. 17, May 22)**

- A distributed system architecture model simplifies and abstracts the functionality of the individual components of a distributed system.
- It deals with the arrangement of components throughout the network of computers, and their interdependence, i.e., how these components communicate with one other.
- It contains several architectural aspects including :
  - **(i) Communicating entities :** This section identifies the entities in the distributed system that are communicating with one another.
  - **(ii) Communication paradigm :** This refers to how various entities interact with one another and the communication paradigm they use.There are three kinds of communication paradigms : (a) inter-process communication, (b) remote invocation, and (c) indirect communication.
  - **(iii) Roles and Responsibilities :** This part describes what roles and responsibilities this entity has in the larger architecture. It is divided into two architectural styles : (a) client-server architecture and (b) peer-to-peer architecture.



(1A5)**Fig. 1.5.2 : Distributed Architectural Model**

1.5.2(A) Client-Server Model

- In a client server network, there are clients and servers.
- A client can be a device or a program. It helps the end users to access the web.
- Some examples of clients are desktop, laptops, smartphones, web browsers, etc.
- A server is a device or a program that responds to the clients with the services.
- It provides files, databases, web pages, shared resources according to its type.
- In this network, a client requests services from the server.
- The server listens to the client requests and responds to them by providing the required service.
- The main advantage of a client server network is that it more secure as the server always manages the access and security.
- It is also easier to take backups.
- On the other hand, it is not very reliable as a failure in the server will affect the functioning of the clients. Furthermore, it is expensive to set up and maintain.

1.5.2(B) Peer-to-Peer Model

- In a peer to peer network, there is no specific client or a server.
- A device can send and receive data directly with each other.
- Each node can either be a client or a server. It can request or provide services accordingly.
- A node is also called a peer.
- In peer to peer network, a node joins the network and start providing services and request for services from other nodes. There are two methods to identify which node provides which service.
- A node registers the service it provides into a centralized lookup service.
- When any node requires obtaining a service, it checks the centralized lookup to find which node provides which facilities. Then, the service providing node and service requesting node communicate with each other.
- In the other method, a node that requires specific services can send a broadcast message to all other nodes requesting a service. Then, the node that has the required service responds to the requested node by providing the service.
- There are multiple advantages in peer to peer network. It is easier to maintain. It is not necessary to have a specialized expert to maintain the network. The entire network does not depend on a single machine. Moreover, it does not require extensive hardware to set up the network.
- On the other hand, a peer to peer network is not very secure. It can also be difficult to maintain an organized file structure. Furthermore, the users need to manage their own backups.

1.5.2(C) Comparison between Client-server and Peer-to-Peer Model

Table 1.5.1 : Client-Server Model Vs Peer-to-Peer Model

Basis of Comparison	Client-Server Model	Peer-to-Peer Model
Basic	In a client-server network, we have a specific server and specific clients connected to the server.	In a peer-to-peer network, clients are not distinguished; every node act as a client and server.

Basis of Comparison	Client-Server Model	Peer-to-Peer Model
Expense	A Client-Server network is more expensive to implement.	A Peer-to-Peer is less expensive to implement.
Stability	It is more stable and scalable than a peer-to-peer network.	It is less stable and scalable, if the number of peers increases in the system.
Data	In a client-server network, the data is stored in a centralized server.	In a peer-to-peer network, each peer has its own data.
Server	A server may get overloaded when many customers make simultaneous service requests.	A server is not bottlenecked since the services are dispersed among numerous servers using a peer-to-peer network.
Focus	Sharing the information.	Connectivity.
Service	The server provides the requested service in response to the client's request.	Each node has the ability to both request and delivers services.
Performance	Because the server does the bulk of the work, performance is unaffected by the growth of clients.	Because resources are shared in a big peer-to-peer network, performance will likely to suffer.
Security	A Client-Server network is a secured network because the server can verify a client's access to any area of the network, making it secure.	The network's security deteriorates, and its susceptibility grows as the number of peers rises.

1.5.3 Fundamental Model

UQ. Explain different types of Failure Models. **(MU - May 22)**

- A system's fundamental model identifies the system's primary elements and specifies how they interact with one another.
- A model's objective is to express all of the underlying assumptions about the system being described.
- To explain distributed systems, three models are used: the Interaction Model, the Failure Model, and the Security Model.

1.5.3(A) Interaction Model

- Processes in a distributed system engage with one another via passing messages, resulting in communication (message transmission) and coordination (synchronization and ordering of operations) amongst processes.
- Each process has a distinct state. Process interaction in distributed systems is influenced by two major factors: (1) Communication speed is frequently a limiting factor; (2) there is no one global concept of time since clocks on various systems drift.
- Communication across a computer network has performance characteristics in terms of latency, bandwidth, and jitter.
- The Interaction model has two variants: Synchronous distributed system and Asynchronous distributed system.

- **Synchronous distributed systems** are those in which the time required to execute each step of a process has a known lower and upper constraint; each sent message is received within a known bounded time and each process has a local clock with a known drift rate from real-time. It is challenging to arrive at realistic values and to offer assurances about the values selected.
- **Asynchronous distributed systems** have different process execution speeds, message transmission delays, and clock drift rates and they are all unbounded. They are identical to the Internet, in which there is no inherent limit on server or network load, and hence on how long it takes, for example, to upload a file over FTP. Asynchronous distributed systems are common in practice.

1.5.3(B) Fault Model

- Both processes and communication routes may fail in a distributed system.
- Failures are classified into three types : omission failures, arbitrary failures, and timing failures.
  - (1) **Omission Failures :** These are situations in which a process or communication channel fails to accomplish the acts that it is expected to do. It includes both process errors and communication omissions.
  - (2) **Arbitrary Failures :** A process continues to run, but responds with a wrong value in response to an invocation. It might also arbitrarily omit to reply. This kind of failure is the hardest to detect.
  - (3) **Timing Failures :** These are only relevant to synchronous distributed systems with time constraints on process execution, message delivery, and clock drift rate. Any of these errors might result in replies not being provided to customers within a certain time frame.

1.5.3 (C) Security Model

- The security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.
- It contains the following :
  - (1) **Object protection :** Access rights are used to indicate who is authorized to do an object's actions, such as who is allowed to read and write a file.
  - (2) **Securing processes and their interactions :** The same is true for a client that gets the result of an invocation but is unsure if it is from the appropriate server.
  - (3) **Adversary :** Privacy and information integrity might be compromised. An 'enemy' who has the ability to copy, change, or insert messages as they move through the network.Another kind of attack is to save copies of communications and respond to them later.
  - (4) **Use of Security Models :** There are several methods for enhancing security measures such as cryptography, authentication, and authorization.

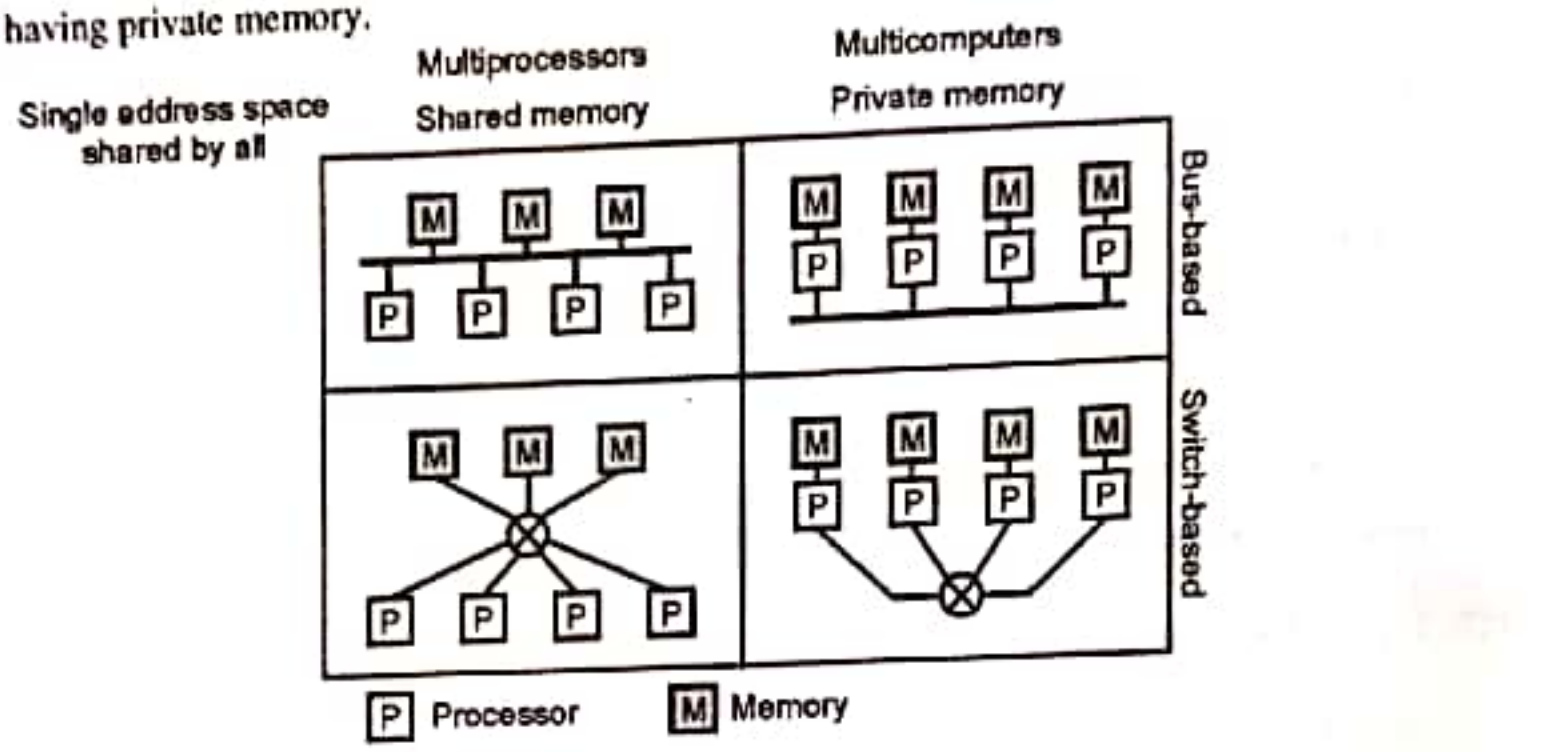
1.6 HARDWARE CONCEPTS

Hardware in distributed systems can be organized in several different ways :

(1) Multiprocessor System	(2) Multicomputer Systems

(1) Multiprocessor System

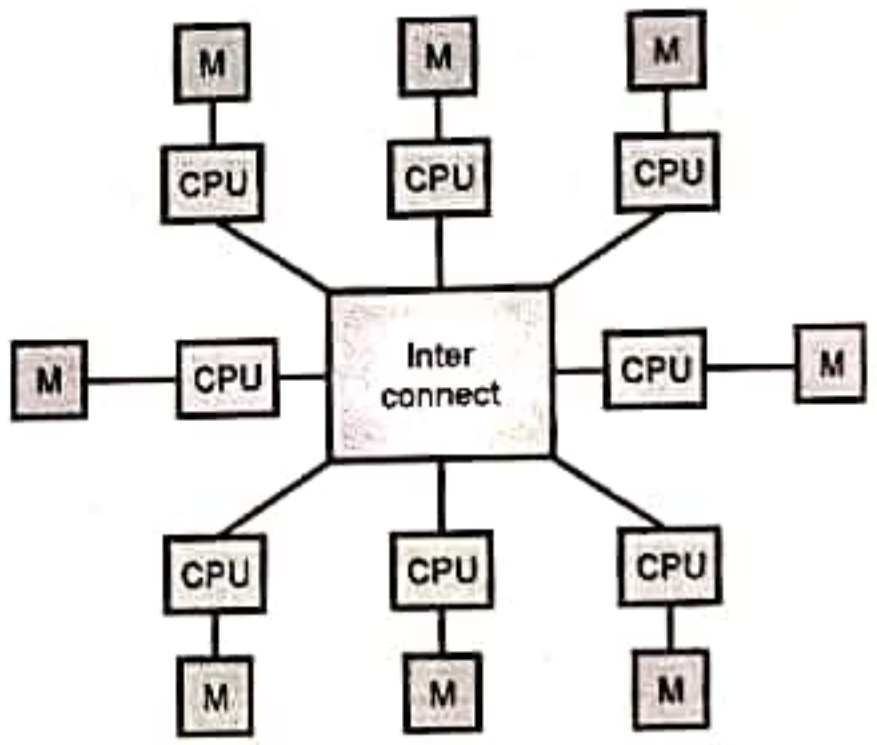
- Multiple processors are used in distributed systems, either spread throughout the network or present at a fixed location.
- A multiprocessor system is a computer system with two or more CPUs and complete access to RAM.
- Increased system execution speed is the main objective of using a multiprocessor, with fault tolerance and effective application execution serving as secondary objectives.
- It might have shared memory or private memory when it comes to storage and processing as shown in Fig.1.6.1.
  - **(i) Shared Memory (Tightly Coupled) :** Shared memory is memory that many processors may access simultaneously in order to improve collaboration or reduce duplication. Shared memory is a helpful tool for transmitting data across different processes.
  - **(ii) Private memory (Loosely Coupled) :** Multicomputer with each CPU directly connected to its local memory are referred to as having private memory.



(1A5)**Fig.1.6.1 : Shared and private memory architecture with multiple processors**

(2) Multicomputer Systems

- A computer system with many processors that are linked together to solve a problem is referred to as a multicomputer system.
- Each CPU has a separate memory that is only accessible by that processor, and those processors are connected by an interconnection network that allows them to interact with one another.
- A multicomputer system can be homogeneous where all CPUs and memory are identical. It can also be heterogeneous where all CPUs and memory are not identical.



(1A7)**Fig. 1.6.2 : Multicomputer System**

1.6.1 Difference Between Multiprocessor and Multicomputer Systems

Table 1.6.1: Multiprocessor System Vs Multicomputer System

Sr. No.	Multiprocessor System	Multicomputer System
(1)	A multiprocessor is a system with two or more central processing units (CPUs) that can perform multiple tasks.	A multicomputer is a system with multiple processors that are attached via an interconnection network to perform a computation task.
(2)	A multiprocessor system is a single computer that operates with multiple CPUs.	A multicomputer system is a cluster of computers that operate as a singular computer.
(3)	Construction of multiprocessor is difficult and cost ineffective than a multicomputer.	Construction of multicomputer is easier and cost effective than a multiprocessor.
(4)	In multiprocessor system, program tends to be easier.	In multicomputer system, program tends to be more difficult.
(5)	Multiprocessor supports parallel computing.	Multicomputer supports distributed computing.

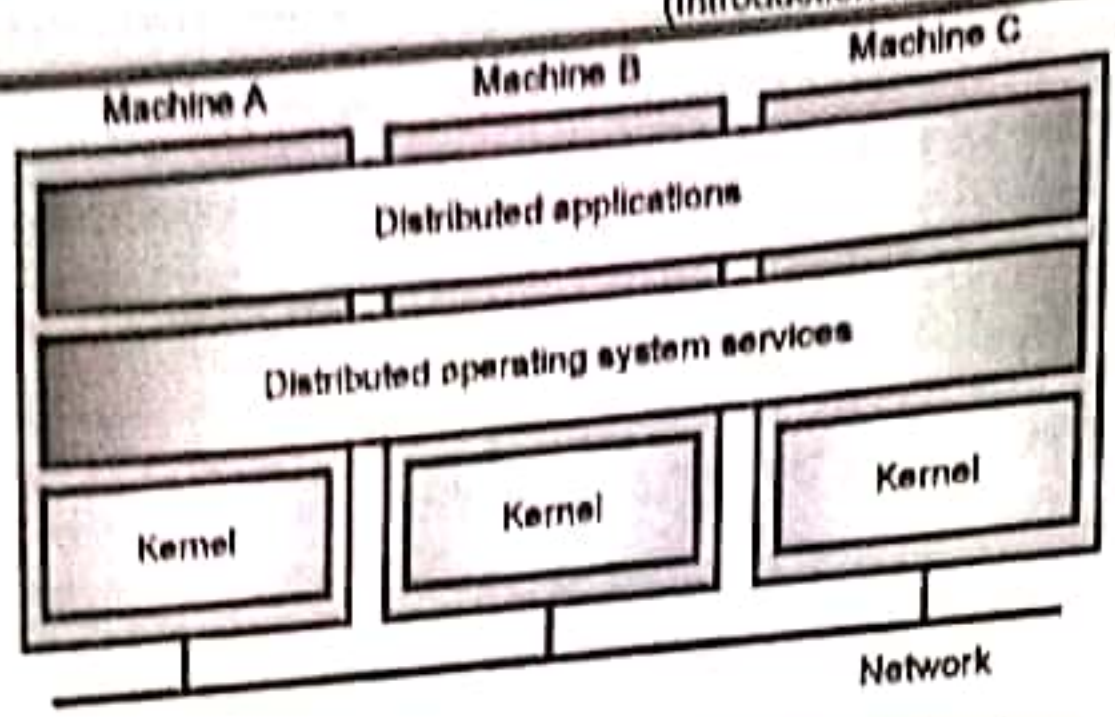
1.7 SOFTWARE CONCEPTS

UQ. Differentiate between NOS, DOS and Middleware in the design of a Distributed Systems. **(MU - May 22)**

- The software of a distributed system consists of an appropriate operating system that is used to facilitate interaction between a user and the hardware. This interaction is necessary for the system to perform.
- There are three types of software widely used in distributed systems: distributed operating system, network operating system and middleware.

1.7.1 Distributed Operating Systems

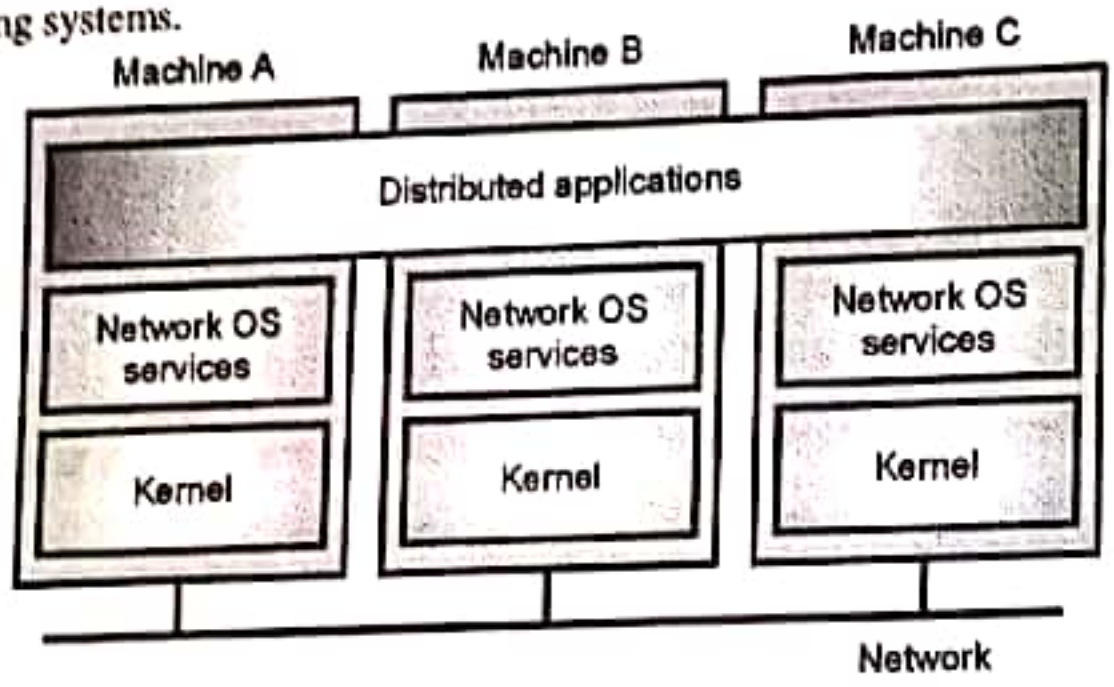
- A fundamental kind of operating system is a distributed operating system (DOS).
- A distributed operating system is a distributed system that abstracts resources, such as memory or CPUs, and exposes common services and primitives that in turn are used by (distributed) applications.
- In a nutshell, it's the same thing that Android or iOS does for mobile phone apps or Linux, macOS or Windows does for desktop apps, just that it happens to coordinate and orchestrate resources that are distributed, typically in a cluster of machines and makes the cluster look like one big box to the app.
- Through a single communication channel, it links several computers.
- Each of these systems also has its own processor and memory.
- These CPUs may also communicate through high-speed buses.
- Distributed operating system is also called as "tightly coupled system".



(1A8)**Fig. 1.7.1 : Structure of Distributed Operating System**

1.7.2 Network Operating System

- An operating system called a network operating system (NOS) is created specifically to handle workstations, database sharing, application sharing, file and printer access sharing, and other network-wide computer functions.
- It is specifically designed for the heterogeneous multicomputer system, where multiple hardware and network platforms are supported.
- It has multiple operating systems that are running on different hardware platforms.
- It is also called as "loosely coupled system".
- Microsoft Windows NT and Digital's OpenVMS are two examples of standalone operating systems that have multipurpose features and can also function as network operating systems.
- Linux, Mac OS X, Microsoft Windows Server 2003, and Microsoft Windows Server 2008 are some of the most well-known network operating systems.

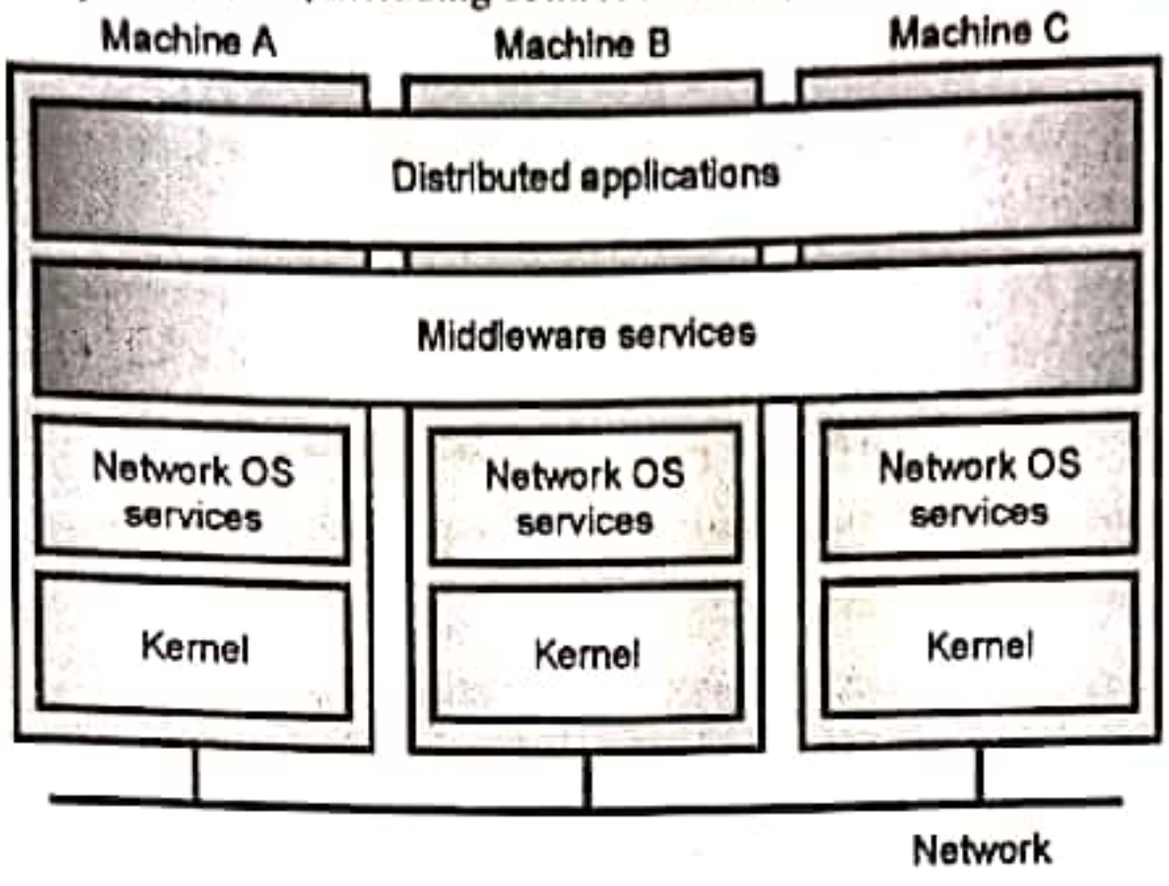


(1A9)**Fig. 1.7.2 : Structure of Network Operating System**

1.7.3 Middleware

- In the context of distributed applications, middleware refers to software that offers additional services above and beyond those offered by the operating system to allow data management and communication across the various distributed system components.
- Complex distributed applications are supported and made easier by middleware.
- Middleware often enables interoperability between applications that run on different operating systems, by supplying services so that the application can exchange data in a standards-based way.

- Middleware sits "in the middle" between application software that may be working on different operating systems.
- Middleware comes in many forms, including database middleware, transactional middleware, intelligent middleware, content-centric middleware, and message-oriented middleware.
- Middleware provides a variety of services, including control services, communication services, and security services.



(1A10)Fig. 1.7.3 : Structure of Middleware

1.7.4 Comparison between the DOS, NOS, and Middleware

Table 1.7.1 : Comparison of DOS, NOS, and Middleware

Sr. No.	Features	Distributed Operating System		Network Operating System	Middleware
		Multiprocessor	Multicomputer		
(1)	Degree of Transparency	Very High	High	Low	High
(2)	Same OS on all nodes	Yes	Yes	No	No
(3)	Number of copies of OS	1	N	N	N
(4)	Basis for Communication	Shared Memory	Messages	Files	Model Specific
(5)	Resource Management	Global, central	Global, distributed	Per node	Per node
(6)	Scalability	No	Moderately	Yes	Varies
(7)	Openness	Closed	Closed	Open	Open

1.8 SERVICES OFFERED BY MIDDLEWARE

There are many services that are offered by the middleware system. Many of them are common in most systems.

(1) **Communication services :** Middleware is used to facilitate communication. It offers services like procedure calls across networks (sending and receiving messages over networks in either a local or remote call), remote-object method invocation (connecting one system to another local system that is remotely located), message queuing system (messages are stored in queues and move through them based on priority), and advanced communication streams (it provides notification of any event going to occur in future).

(2) **Information services :** Services used to handle data in a distributed system are known as information services. It offers services like large-scale, system-wide naming services (where data is managed using naming services), advanced directory services (where data is stored in the directory), location services for tracking mobile objects, persistent storage facilities, data caching, and replication (cache data and replicate without losing consistency).

(3) **Control Services :** These services allow applications to choose when, where, and how to access data. Examples include distributed transaction processing (which aids in the processing of transactions that occur across a distributed network), code migration (helps in migration of function, for e.g., if in two workstations, one workstation is heavily loaded then the process is transferred to another workstation using control service).

(4) **Security Services :** Services for safe processing and communication are referred to as security services. It offers features like as authentication and authorization, encryption, and inspection of whole system.

(5) **Persistence :** It is used to store distributed objects permanently onto the data store.

(6) **Messaging :** It is used to send or receive messages in terms of request/reply primitive.

(7) **Querying :** It is used to query on distributed objects.

(8) **Concurrency :** It is used to share and access resources concurrently.

1.9 TYPES OF MIDDLEWARE

GQ. Explain different types of middleware in the distributed system.

Middleware comes in a variety of forms, some of which are listed here.

(1) MessageOriented Middleware (MOM)	(2) Object Middleware
(3) RPC Middleware	(4) Database Middleware
(5) Transaction Middleware	(6) Portal Middleware
(7) Content-centric Middleware	(8) Embedded Middleware

(1) Message Oriented Middleware (MOM)

- As a kind of middleware, message-oriented middleware facilitates communication between applications. It provides a messaging framework for applications to communicate with one another asynchronously, without requiring both parties to be online at the same time.
- By using MOM, apps may communicate without worrying about how the other works physically or technically. This makes it possible for programs written in various programming languages and built for various platforms to communicate with one another.
- MOM systems often have characteristics like message routing, message persistence, and message transformation to allow the fast and reliable flow of messages among applications. Examples of MOM systems include Apache ActiveMQ, IBM WebSphere MQ, and Microsoft Message Queue.

(2) Object Middleware

- Object middleware facilitates communication between different programs and parts. It provides a set of services and an underlying architecture that allow programs to use each other's data and objects as if they were their own.
- Object middleware often includes features like object-oriented messaging, distributed object management, and object transaction support to promote efficient and reliable communication between programs.
- Object middleware technologies include CORBA and Java's Remote Method Invocation. In distributed systems, where applications run on a number of different computers connected through a network and must communicate with one another to get access to a set of shared resources and services, object middleware is often used. It allows programs to

interact in a way that is independent of the underlying network and operating system, which simplifies the design and maintenance of distributed applications.

(3) RPC Middleware

- RPC middleware is a sort of middleware that enables programs to invoke procedures or functions on remote systems as if they were local. It provides a messaging architecture that allows programs to make requests to remote processes and get responses concurrently.
- RPC middleware often includes features like message routing, message persistence, and message transformation to facilitate the efficient and reliable transfer of requests and responses between applications.
- One such RPC middleware framework is Apache Thrift. Programs using RPC middleware are often employed in distributed systems when access to shared resources or services requires the execution of remote programs.

(4) Database Middleware

- Database middleware is a kind of middleware that provides an interface between applications and databases, letting programs read from and write to databases without understanding their underlying structure or operations.
- Database middleware often includes features like data access libraries, data mapping tools, and query languages to facilitate high-level, abstracted communication between programs and the database.

(5) Transaction Middleware

- The architecture for handling distributed transactions is provided by transaction middleware, a kind of middleware. It enables programs to perform atomic operations that may either finish or roll back any changes, ensuring that information is always correct and consistent.
- Transaction middleware often includes features like transaction management, transaction coordination, and transaction recovery to facilitate the execution of transactions across many systems. Transaction middleware systems like the Java Transaction API are one example

(6) Portal Middleware

- By definition, portal middleware is software that provides a gateway through which users may access and integrate a wide variety of applications and services via the Internet. It eliminates the need for users to switch between applications or websites in order to have access to the many programs and data that are available to them.
- Portal middleware often includes features like authentication and authorization, user profile management, and content publishing to facilitate the development and maintenance of portal applications.
- A few of popular portal middleware options include Liferay and Microsoft SharePoint. Whenever there is a need for several applications and data sources to be accessed and managed via a unified portal, portal middleware is often used.

(7) Content-centric Middleware

- This kind of middleware is known as "content-centric" because it concentrates on the distribution and management of content rather than on the management of applications and data. It's a set of resources that may be used to provide apps more control over data and make it easier for them to share resources with one another.
- Content-centric middleware often includes capabilities like content management, content routing, and content transformation to facilitate the efficient and reliable distribution of data. Some examples of content-centric middleware are content delivery networks and content management systems (CDN).

(8) Embedded Middleware

- Embedded middleware is software designed to run on devices with limited hardware and software resources, such as those found in embedded systems, sensors, and IoT devices.
- By using the offered infrastructure and services, applications may interact with the device's hardware and software, communicate with one another, and have access to shared resources.

- The development of software for embedded devices is facilitated by embedded middleware, which often includes features like resource management, communication protocols, and device abstraction. Examples of embedded middleware include the IoT platform and the Real-Time Operating System (RTOS).
- Embedded middleware is often used in applications for embedded systems and Internet of Things (IoT) devices because to the limited hardware and software resources of such devices, which must be carefully regulated to ensure stable and effective operation.

1.10 MODELS OF MIDDLEWARE

(MU - May 16)

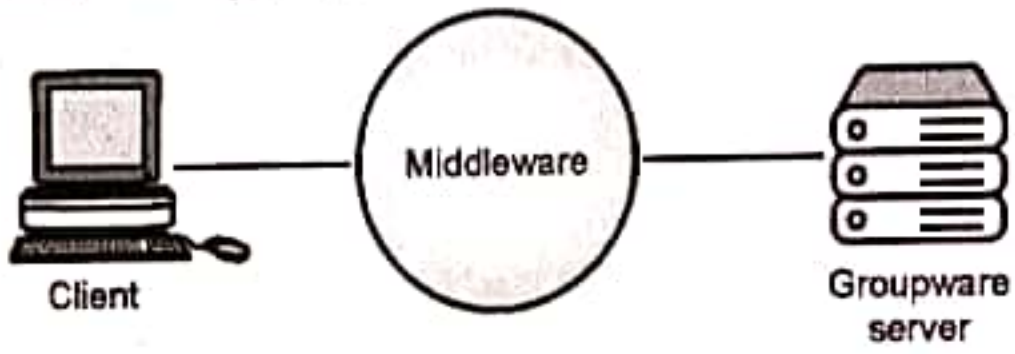
UQ. What are the different models of middleware ?

There are five distinct models of middleware discussed in this section.

(1) Client-server model	(2) Vertical distribution model
(3) Horizontal distribution model	(4) Peer-to-Peer Model
(5) Hybrid Model	

(1) Client-server model

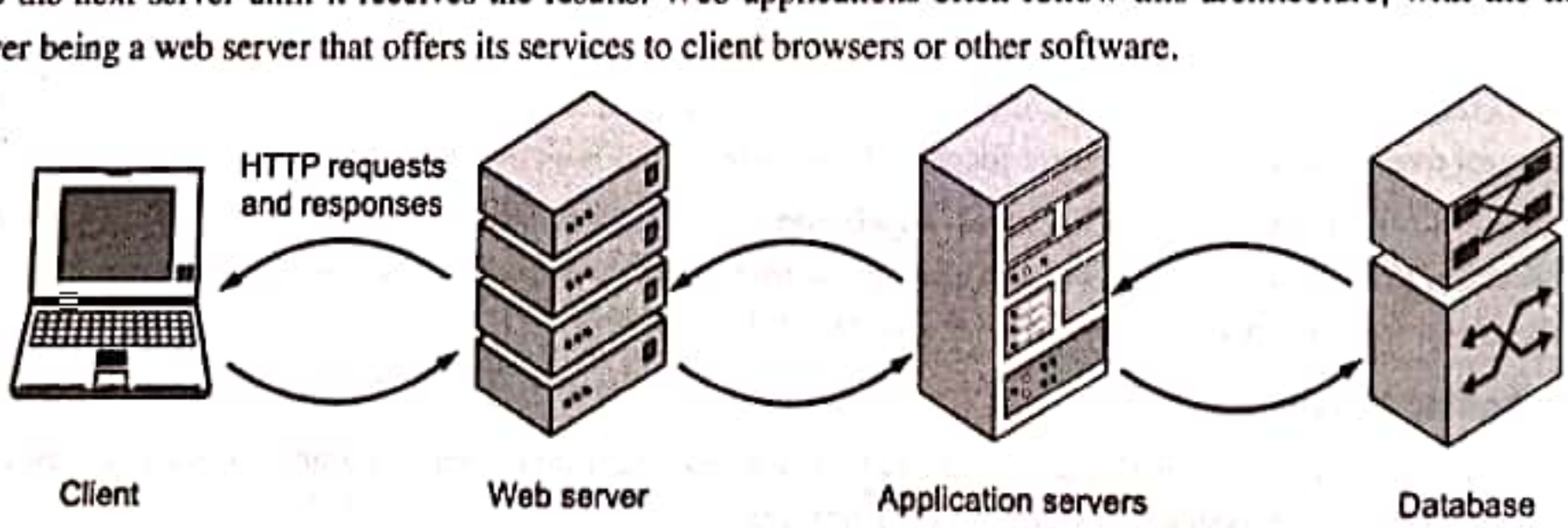
- The client-server model is a way of organizing and implementing middleware, where the middleware is a separate set of services that provides services to client applications.
- In this model, the client applications send requests to the middleware, which processes the requests and sends back the results. This model is commonly used for web applications, where the middleware server is a web server that provides services to web browsers or other client applications.



(1A11)**Fig. 1.10.1 : Middleware facilitating Client-Server Communication**

(2) Vertical distribution model

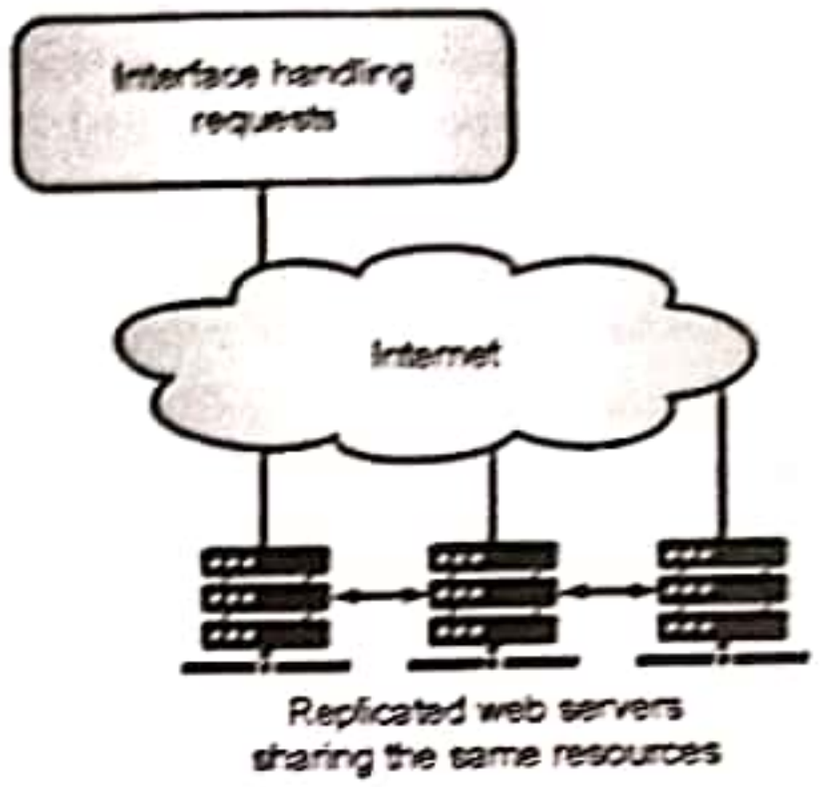
- Also known as Muti-tier Model. In other words, it's a more advanced form of client-server architecture. This approach incorporates a cluster of servers.
- Requests from client applications are sent to the first middleware server, which then processes the request and transmits it to the next server until it receives the results. Web applications often follow this architecture, with the middleware server being a web server that offers its services to client browsers or other software.



(1A12)**Fig.1.10.2: Middleware as a Vertical Distribution Server**

(3) Horizontal distribution model

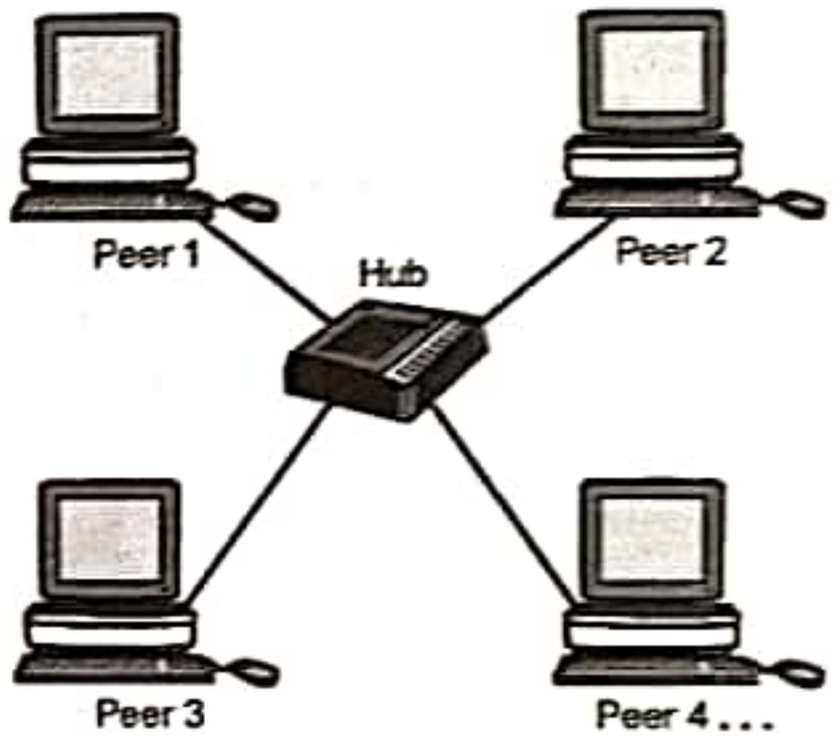
- The limitations of the preceding models in terms of scalability and reliability necessitated the adoption of a horizontally distributed model todistribute the tasks of a single server over several workstations.
- Rather of relying on a centralised server or broker, each server's applications are responsible for communicating directly with the applications on other servers. Applications running on multiple computers often employ this architecture.



(IA12)Fig. 1.10.3 : Middleware as a Horizontal Distribution Model

(4) Peer-to-Peer Model

- It operates on a decentralized approach to communication known as the "peer-to-peer" (P2P) paradigm. In a P2P network, each machine is treated equally and may perform the roles of both client and server.
- This kind of technology eliminates the need for a centralized server, allowing users to exchange data and services between themselves. File sharing is a common use case for peer-to-peer systems because any user may publish files to the network and get files from other users.
- This allows for more efficient use of bandwidth and reduces the workload on each individual server.



(IA14)Fig.1.10.4 : Peer-to-peer Model

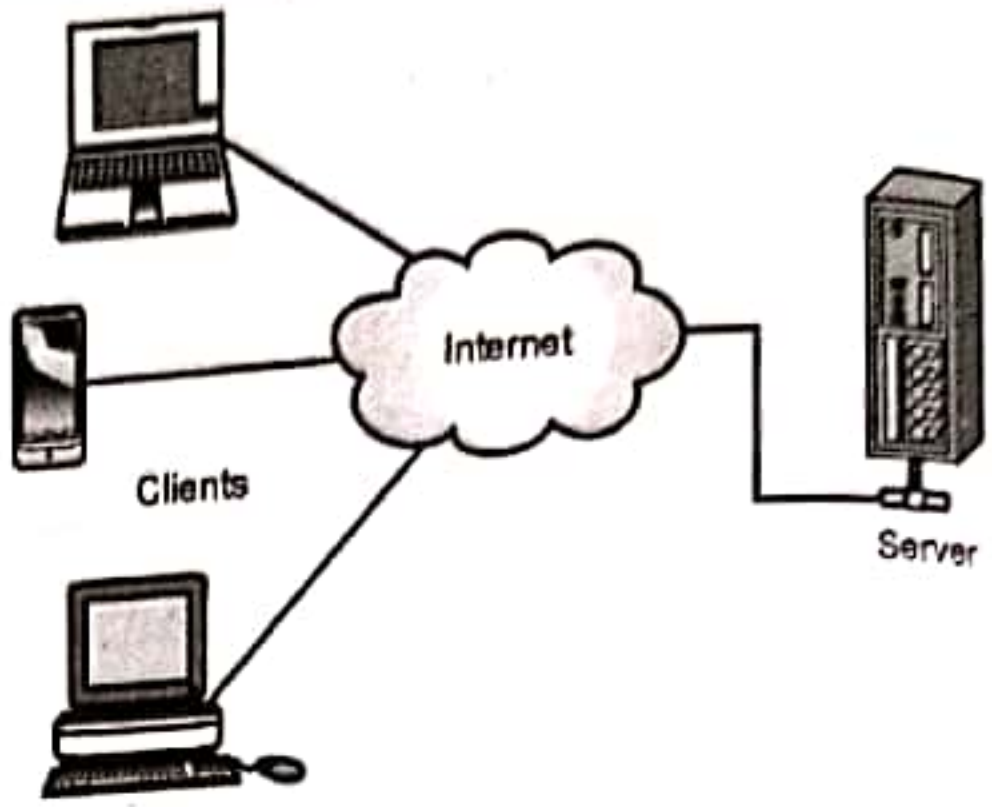
(5) Hybrid Model

- A hybrid model includes elements from many middleware architectures, such as client-server and peer-to-peer. This allows the best features of both architectures to be combined into one, resulting in a system with more adaptability and scalability.
- A client-server architecture may be used in a hybrid middleware system to provide services like financial transactions with the highest possible degree of security and reliability.
- Services such as file sharing may benefit from the scalability and adaptability of a peer-to-peer architecture, and they may be utilized simultaneously.

1.11 CLIENT-SERVER ARCHITECTURAL MODEL

- Clients in a client-server architecture are often users of the system, who are using personal computers, whereas servers are stationed elsewhere on the network and are typically more powerful computers.
- In a client-server system, the server is responsible for providing and managing most of the data and functions that end users, or "clients," need to complete their tasks.

- As all requests and services are sent digitally, this computing paradigm is sometimes called the client server network or networking computing model.
- Client-server architecture has three main parts that must all be present for it to function properly. Clients, servers, and network hardware are the three main parts.
- To get the required resources, the client first makes a network request to the server. client requests are processed by the server, and the server then sends back a response to the client through the network.



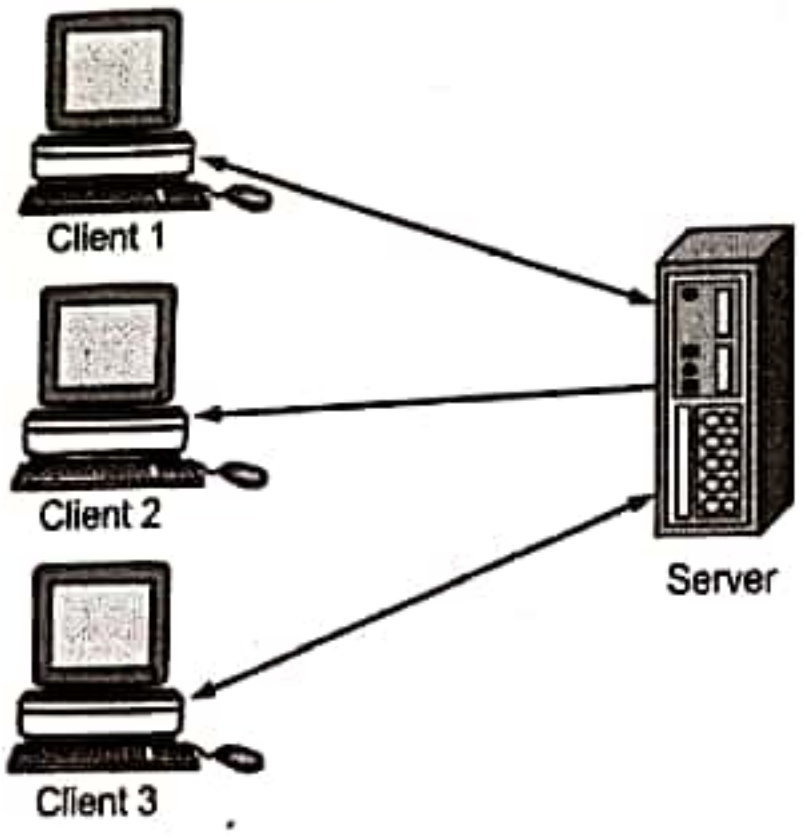
(1A15)**Fig. 1.11.1 : Client-Server Architecture**

1.11.1 Different Types of Client-Server Architecture

This section discusses different types of client-server architectures.

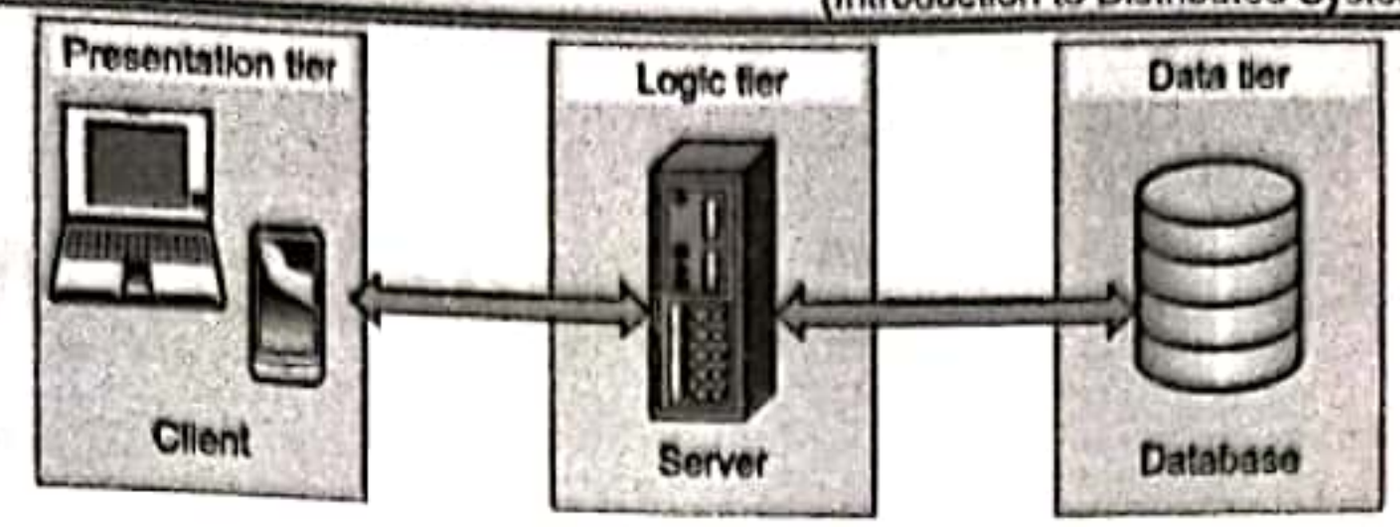
(1) **1-Tier Architecture :** The term "1-tier architecture," or "single-tier architecture," is used to describe a specific type of software architecture in which all the necessary components for the functioning of an application are accessible under the same package.

(2) **2-Tier Architecture :** An example of a two-tier architecture is one in which the user interface and the database are hosted on separate computers. In this setup, the client and server are on different machines.



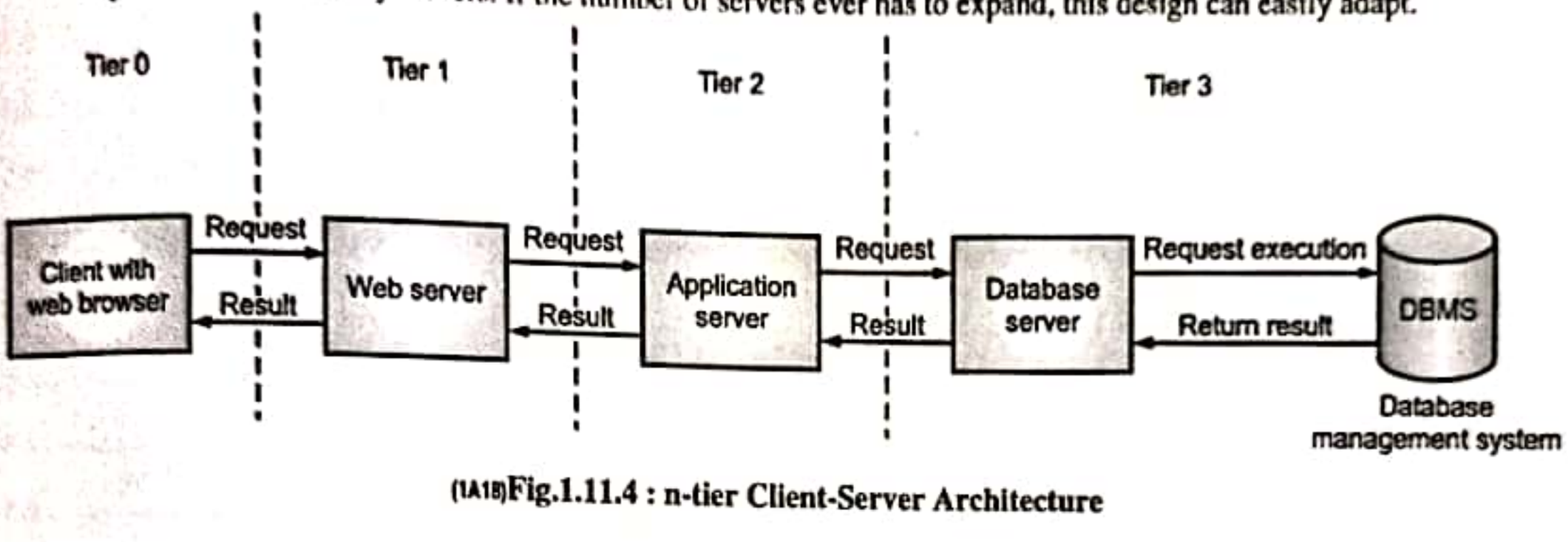
(1A16)**Fig. 1.11.2 : 2-Tier Client-Server Architecture**

(3) **3-Tier Architecture :** As opposed to the direct connection between the client and the server in 2-tier design, a middle tier is present in 3-tier client server architecture. The middle tier will be the first to receive a request from the client to get data from the server. After that, it will be sent off to the server for processing. When the server responds to the client, it will follow the same format. In a 3-tier design, the presentation layer, the application/logic layer, and the data tier are the three fundamental components of the structure.



(1A17)Fig.1.11.3 : 3-Tier Client Server Architecture

(4) **n-Tier Architecture** : There is a kind of architecture called n-tier architecture, where 'n' is the number of tiers, and it is also known as distributor architecture. Between the UI and the DB, many application servers mediate in an n-tier architecture, which is different from the more common 3-tier architecture. This is done to ensure that the business logics are spread out across many servers. If the number of servers ever has to expand, this design can easily adapt.



(1A18)Fig.1.11.4 : n-tier Client-Server Architecture

Descriptive Questions

Q. 1 Define distributed system. State the advantages and disadvantages of a distributed system.

Q. 2 State the goals and issues in designing distributed system.

Q. 3 Explain different types of transparencies in distributed systems.

Q. 4 Explain different distributed system models.

Q. 5 Explain different types of distributed system.

Q. 6 Compare and contrast client-server and peer-to-peer models.

Q. 7 Explain in brief the hardware and software models supported by the distributed system.

Q. 8 Explain in detail architectural models of distributed system.

Q. 9 Differentiate between DOS, NOS and middleware based distributed system.

Q. 10 Explain different client-server models with neat diagram.

Chapter Ends...

❑❑❑

MODULE 2

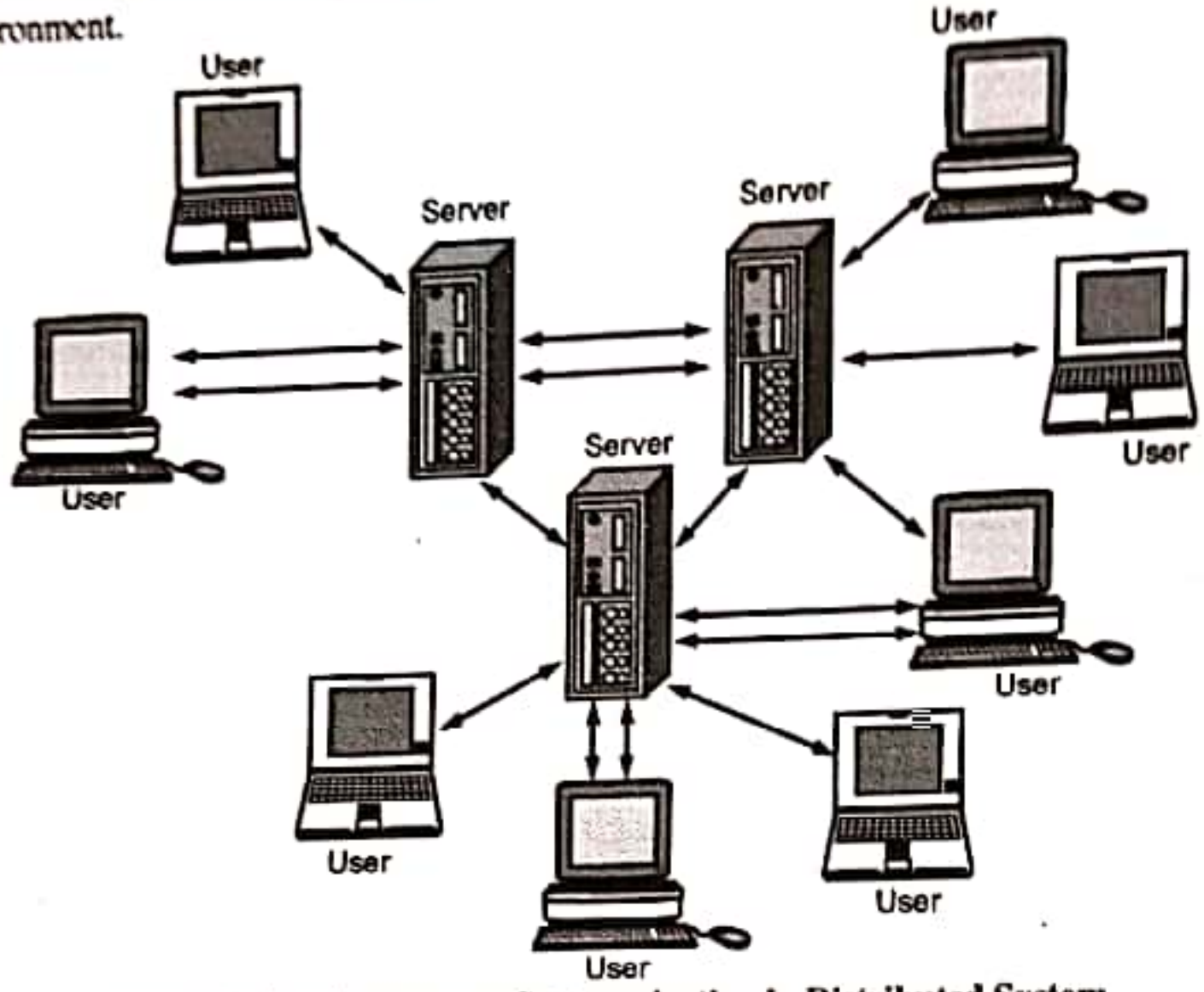
CHAPTER 2

Communication

2.1 INTRODUCTION

GQ. Define Inter-process Communication. Explain different types of Communications.

- Communication refers to the exchange of messages and information via various processes that run on different machines. Communication between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes.
- **Inter-process Communication** is a process of exchanging data between two or more independent processes in a distributed environment.



(IBI)**Fig. 2.1.1 : Inter-process Communication in Distributed System**

- Inter-process communication has two functions: Synchronization and Message Passing.
- Inter-process communication requires **synchronization.** It refers to a situation in which the data used to communicate between processors is control information. It is either provided by the inter-process control mechanism or handled by the communicating processes.
- In **message-passing systems,** processors communicate with one another by sending and receiving messages over a communication channel. Message passing takes several forms such as pipes, FIFO, Shared Memory, and Message Queues.

2.1.1 Types of Communications

Following are the different types of communications as shown in Fig. 2.1.2.

(1) **Persistence** : Persistence means that the network can store messages for an arbitrary period until the next receiver is ready. Email and ground deliveries are good examples.

(2) **Transient** : Message is stored only so long as the next receiver is ready. For example, Transport- level communication discards the message if the process crashes for any reason. The system will not store the message.

(3) **Asynchronous** : Asynchronous communication means that the sender is doing non-blocking sending. It continues immediately after submitting the message.

(4) **Synchronous** : Synchronous communication blocks the process until the message is received or the sender gets a response from the server.

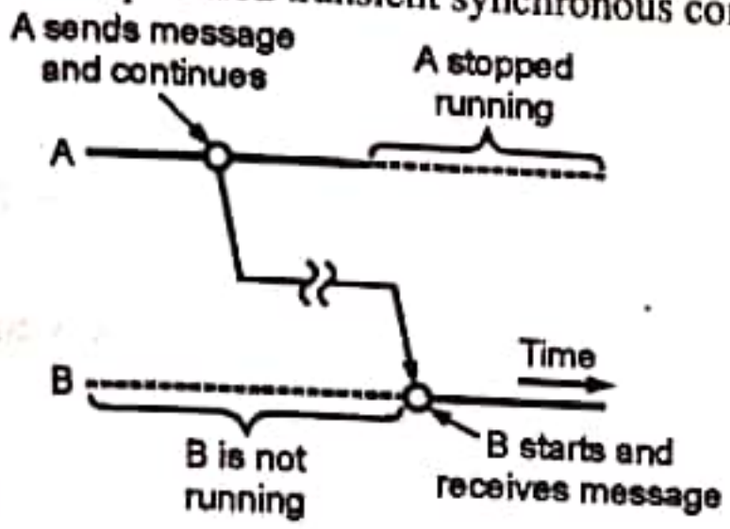
(5) **Persistent synchronous communication** : The sender is blocked when it sends the message, waiting for an acknowledgement to come back. The message is stored in a local buffer, waiting for the receiver to run and receive the message. Some instant message applications, such as Blackberry messenger, are good examples. When you send out a message, the app shows you the message is "delivered" but not "read". After the message is read, you will receive another acknowledgement.

(6) **Transient asynchronous communication :** Since the message is transient, both entities must be running. Also, the sender doesn't wait for responses because it is asynchronous. UDP is an example.

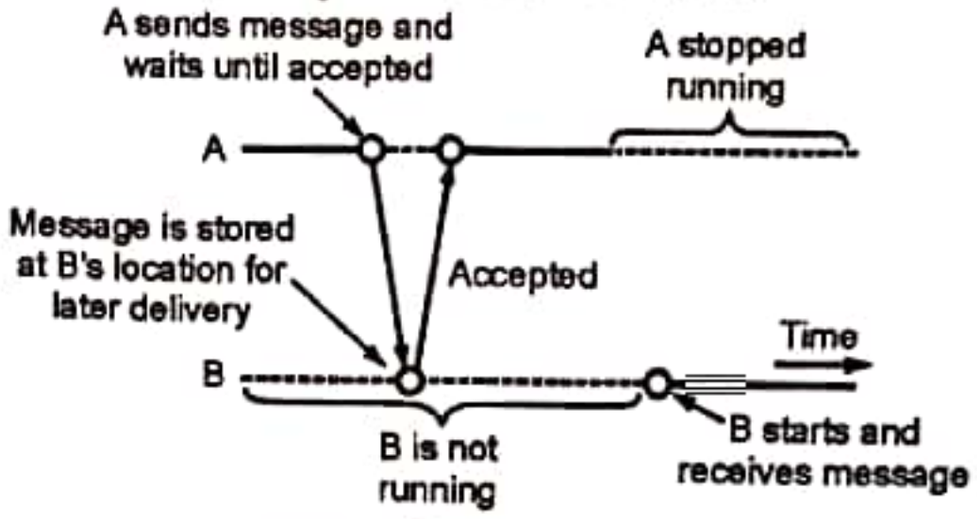
(7) **Receipt-based transient synchronous communication :** The acknowledgement sent back from the receiver indicates that the message has been received by the other end. The receiver might be working on some other process.

(8) **Delivery-based transient synchronous communication :** The acknowledgement comes back to the sender when the other end takes control of the message. Asynchronous RPC is an example.

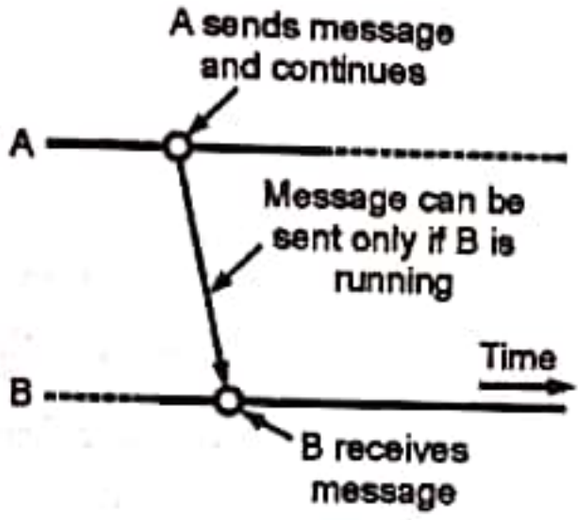
(9) **Response-based transient synchronous communication :** The sender blocks until the receiver processes the request and sends back a response. RPC is an example. There is no clear mapping of TCP to any type of communication. From an application standpoint, it maps to transient asynchronous communication. However, from a protocol standpoint, it maps to a receipt-based transient synchronous communication if it only has a one-size window.



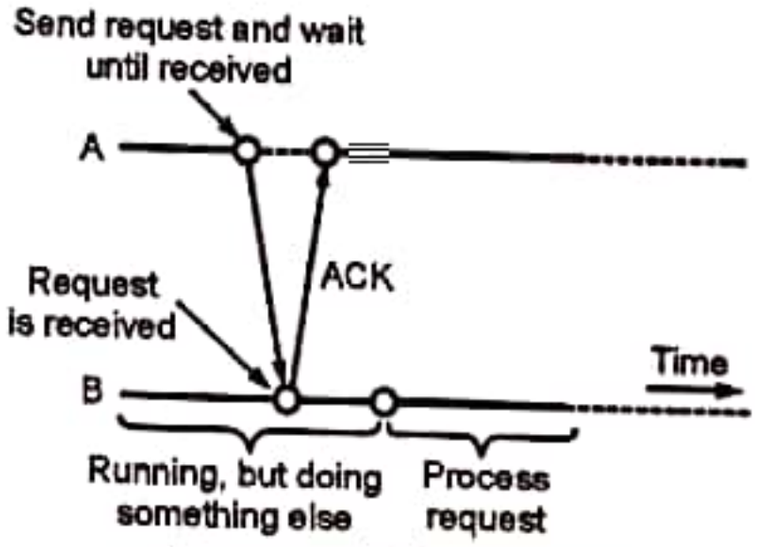
(a) Persistent asynchronous



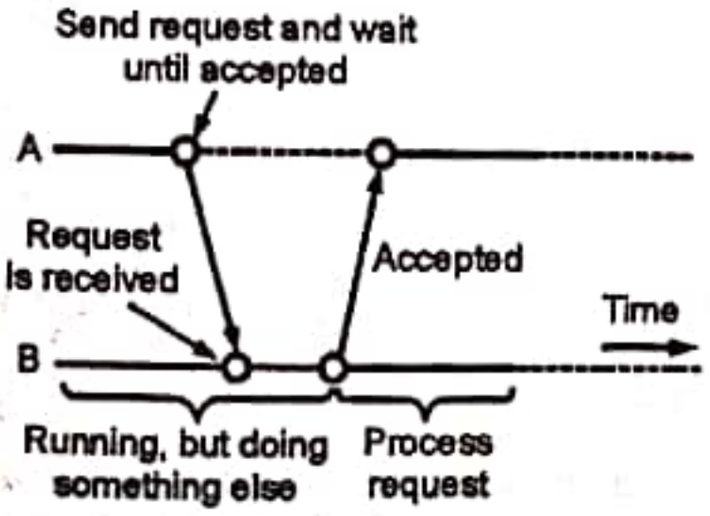
(b) Persistent synchronous



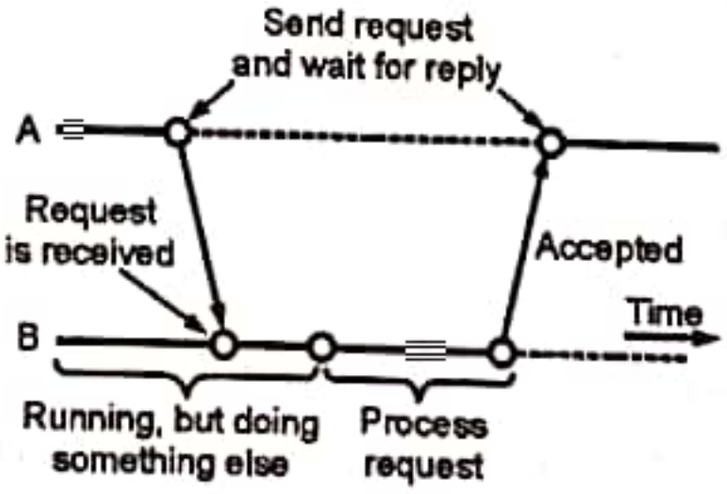
(c) Transient asynchronous



(d) Transient synchronous (Receipt-based)



(e) Transient synchronous (Delivery-based)



(f) Transient synchronous (Response-based)

(182)**Fig. 2.1.2 : Types of Communications**

2.1.2 Message Passing Interface (MPI)

GQ. Explain the concept of Message Passing Interface (MPI) in detail.

- Sockets are designed for simple send-and-receive primitives and one-to-one communication using general-purpose protocol stacks such as TCP/IP.
- The message-passing interface (MPI) is a standardized interface for exchanging messages between multiple computers running a parallel program across distributed memory.
- MPI is designed for parallel applications and as such is tailored to transient communication.
- MPI can be used for communication between clusters of clients and servers, which have intensive communication, and the overhead of TCP/IP is very high for this scenario.
- The MPI standard defines the syntax and semantics of library routines helpful in writing portable message-passing programs.
- MPI isn't a programming language. It's a library of functions that programmers can call from C, C++, or Fortran code to write parallel programs.
- With MPI, an MPI communicator can be dynamically created and have multiple processes concurrently running on separate nodes of clusters.
- Each process has a unique MPI rank to identify it, its own memory space, and executes independently from the other processes.
- Processes communicate with each other by passing messages to exchange data.
- MPI's goals are high performance, scalability, and portability.
- Parallelism occurs when a program task gets partitioned into small chunks and distributes those chunks among the processes, in which each process processes its part.
- MPI also has more advanced primitives of different forms of buffering and synchronization.

(1) **MPI_bsend** : Transient asynchronous communication is supported by means of this primitive.

(2) **MPI_send** : The primitive MPI_send may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side or until the receiver has initiated a receive operation. (receipt based).

(3) **MPI_ssend** : Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI_ssend primitive. (delivery based).

(4) **MPI_sendrecv** : When a sender calls MPI_sendrecv, it sends a request to the receiver and blocks until the latter returns a reply, which provides the strongest form of synchronous communication. Basically, this primitive corresponds to a normal RPC. (reply based).

(5) **MPI_Isend** : A sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. When the messages to be sent are very large, memory copy overhead can be substantial. So, it sends a pointer to a local buffer instead of the whole copy of the data.

(6) **MPI_Issend** : Variant of MPI_ssend with sending pointer instead of a copy.

(7) **MPI_recv** : The operation MPI_recv is called to receive a message; it blocks the caller until a message arrives.

(8) **MPI_Irecv** : It's an asynchronous variant of MPI_recv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether a message has indeed arrived, or block until one does.

2.1.2 (A) MPI Communication Methods

MPI provides three different communication methods that MPI processes can use to communicate with each other. The communication methods are discussed as follows:

(1) Point-to-Point Communication

- Point-to-Point communication is the most used communication method in MPI.
- It involves the transfer of a message from one process to a particular process in the same communicator.
- MPI provides blocking (synchronous) and non-blocking (asynchronous) Point-to-Point communication.
- With blocking communication, an MPI process sends a message to another MPI process and waits until the receiving process completely and correctly receives the message before it continues its work.
- On the other hand, a sending process using non-blocking communication sends a message to another MPI process and continues its work without waiting to ensure that the message has been correctly received by the receiving process.

(2) Collective Communication

With this type of MPI communication method, a process broadcasts a message is to all processes in the same communicator including itself.

(3) One-Sided Communication

With the MPI One-sided communication method, a process can directly access the memory space of another process without involving it.

2.2 LAYERED PROTOCOLS

- Due to the lack of shared memory in distributed systems, all communication is based on sending and receiving (low-level) messages.
- When process A wishes to communicate with process B, it constructs a message in its own address space first.
- Then it makes a system call, which instructs the operating system to send the message to B over the network.
- Although this basic concept appears straightforward, A and B must agree on the meaning of the bits being sent to avoid chaos.
- A variety of agreements are required. How many volts should be used to signal a 0-bit, and how many volts should be used to signal a 1-bit? How does the receiver know which part of the message is the last? How does it know if a message is damaged or lost, and what should it do if it does? How long and how are numbers, strings, and other data items represented?
- In short, agreements are required at multiple levels, ranging from low-level details of bit transmission to high-level details of how information is to be expressed.
- To easily deal with the numerous levels and issues involved in communication, the **International Standards Organization (ISO)** has developed a reference model that clearly identifies the various levels involved, gives them standard names, and indicates which level should do which job.
- This model is known as the Open Systems Interconnection Reference Model, abbreviated as ISO-OSI or simply the OSI model.
- An open system is one that can communicate with any other open system by following standard rules that govern the format, content, and meaning of messages sent and received. Protocols are documents that formalize these rules.

- A protocol is a set of rules and regulations that govern the communication describing how two entities should exchange information.
- Networking tasks, such as file transfer, frequently necessitate the use of more than one protocol.
- The OSI model separates two types of protocols into generic categories.
- With **connection-oriented protocols**, the sender and receiver must deliberately establish a connection before data can be sent. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system.
- **Connectionless protocols** don't require any setup beforehand. When the first message is prepared, the sender simply sends it. A connectionless communication example is placing a letter in a mailbox.
- Both connection-oriented and connectionless communication are common with computers.
- As seen in Fig. 2.2.1, the OSI model divides communication into seven levels or layers. Each layer addresses a distinct communication-related issue.

☞ Layer 1 : Physical Layer

This layer defines the physical media that connects hosts and networks, as well as the procedures for transferring data between machines using a specified media. This layer is commonly referred to as the model's hardware layer.

☞ Layer 2 : Data Link Layer

This layer has the responsibility of ensuring that data is delivered reliably across the physical network. For example, it provides an abstraction of a reliable connection over the potentially unreliable physical layer.

☞ Layer 3 : Network Layer

This layer has the responsibility of directing the flow of communications between different machines. The address of the machine to which the transmission is being sent is used as the basis for its determination of the route that the transmission must travel. This layer is also responsible for addressing issues related to network congestion.

☞ Layer 4 : Transport Layer

This layer delivers data in a sequential order from start to finish. It is the lowest layer that provides end-to-end service to applications and higher layers. This layer conceals the underlying network's topology and characteristics from users. If the service characteristics demand it, it provides reliable end-to-end data delivery.

☞ Layer 5 : Session Layer

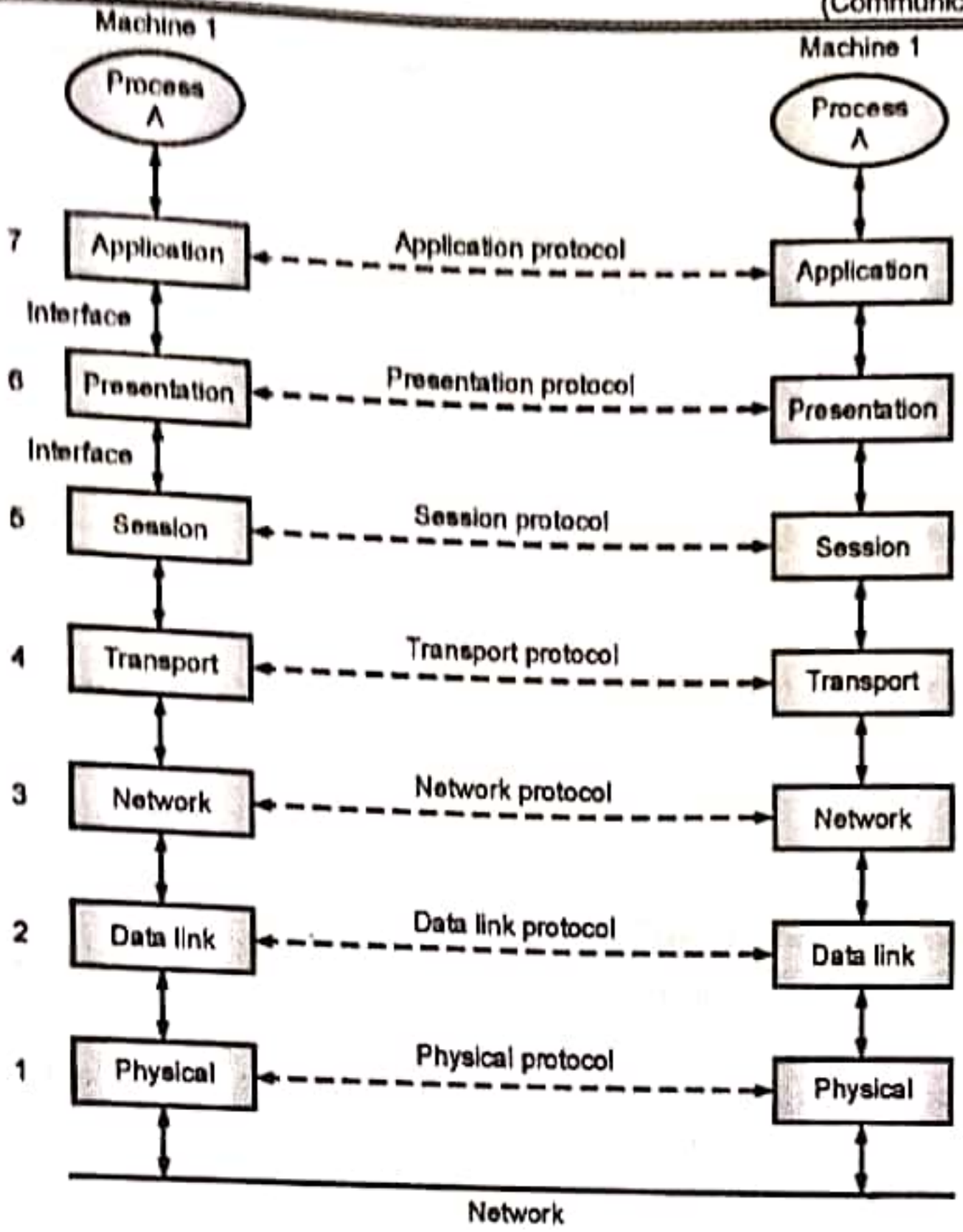
This layer manages sessions between cooperating applications.

☞ Layer 6 : Presentation Layer

This layer converts data from the computer's local representation to the processor-independent format that is sent across the network. It can also negotiate transfer formats in some protocol suites. Standard routines that compress text or convert graphic images into bit streams for network transmission are common examples.

☞ Layer 7 : Application Layer

This layer includes the user-level programs as well as the network services. Telnet, FTP, and TFTP are a few examples of application layer protocols.



(183)**Fig. 2.2.1 : ISO-OSI Reference Model**

2.3 REMOTE PROCEDURE CALL (RPC)

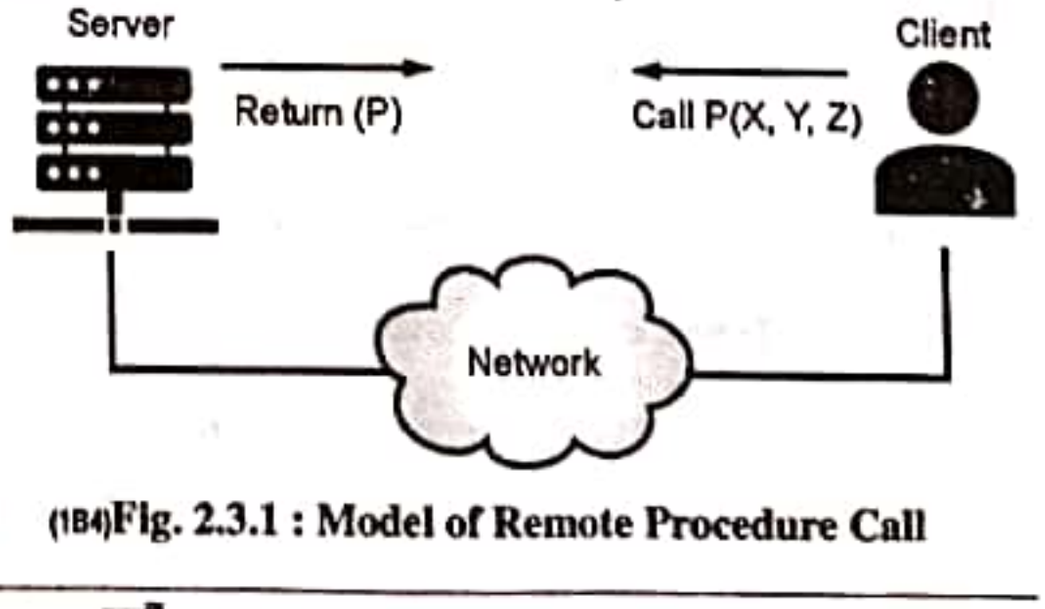
UQ. Explain the concept of Remote Procedure Call. **(MU - Dec-16)**

UQ. Define Remote Procedure Call (RPC). Explain the working of RPC in detail. **(MU - May-17,18, 22, Dec-19)**

UQ. Explain call semantics of RPC. **(MU - Dec. 17)**

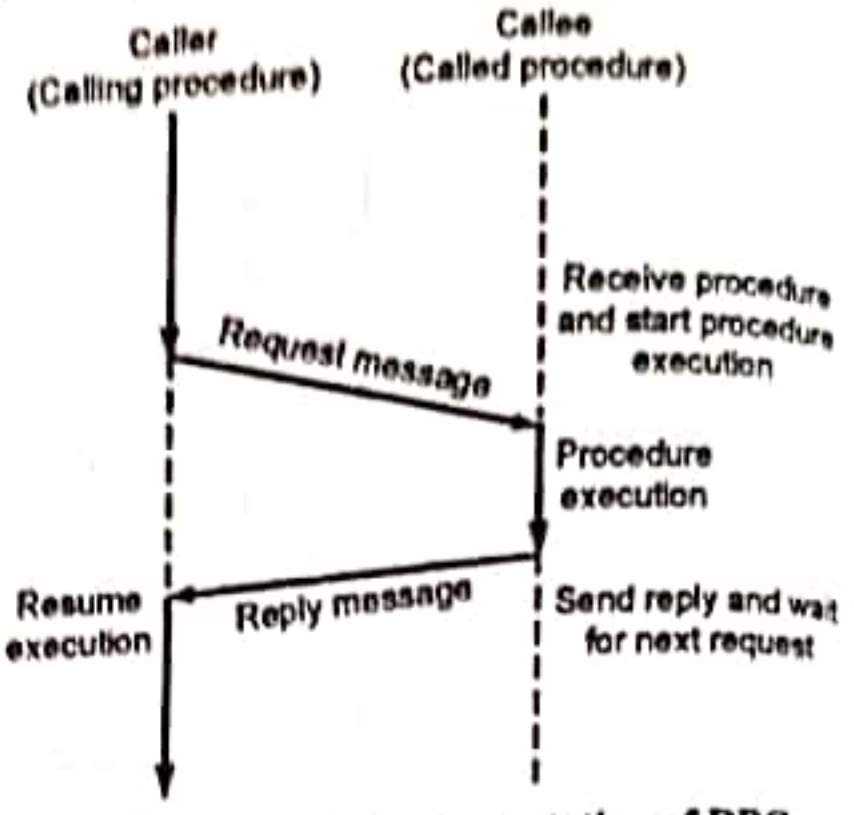
UQ. Write a note on RPC and RMI. **(MU - May 19)**

- Remote Procedure Call is an inter-process communication technique used to construct distributed and client-server applications. It is also known as a function call or a subroutine call.
- A remote procedure call is when a computer program causes a procedure to execute in a different address space, coded as a local procedure call, without the programmer explicitly stating the details for the remote interaction.
- The programmer writes essentially the same code whether the subroutine is local to the executing program or remote.
- This is a form of client-server interaction implemented via a request-response message-passing system.



(184)**Fig. 2.3.1 : Model of Remote Procedure Call**

- The RPC model implies **location transparency** which indicates that calling procedures are largely the same, whether local or remote.
- Usually, they are not identical. Remote calls are usually orders of magnitude slower and less reliable than local calls.
- RPCs are an instance of inter-process communication (IPC), where different processes use different address spaces.
- On the same host computer, they each have their own virtual address space, despite sharing the same physical address space. Nevertheless, the physical address space is different if they are on separate hosts.
- During a Remote Procedure Call(RPC) :
  (1) The OS suspends and transfers the calling environment and procedure parameters respectively, across the network and to the environment where the procedure executes.
  (2) The OS transfers back the result produced by a procedure to the calling environment. Execution also resumes just like a regular procedure call.



(185)**Fig. 2.3.2 : Implementation of RPC**

- The two major components of the RPC model are :
  (1) **Calling Procedure** : It is in the local machine and contacts the remote procedure to be executed by placing a request message to the called procedure.
  (2) **Called Procedure (remote Procedure)** : It may be located on the same computer as the calling procedure or on a different computer. Gets the parameters from the request message sent by the calling procedure and executes accordingly giving the results via a reply message to the calling process.
- The two types of messages involved in the implementation of an RPC system are as follows :
  (1) **Call Messages** : They are sent by the client to the server for request execution of a particular remote procedure.
  (2) **Reply Messages** : They are sent by the server to the client for returning the result of remote procedure execution.

2.3.1 Features of RPC

In an operating system, remote procedure call (RPC) has the following features, such as :

(1) RPC hides the complexity of the message passing process from the user.
(2) RPC only uses specific layers of the OSI model like the transport layer.
(3) Clients can communicate with the server by using higher-level languages.
(4) RPC works well with both local environments and remote environments.
(5) The program of RPC is written in simple code and is easily understood by the programmer.
(6) The operating system can handle processes and threads involved in RPC easily.
(7) The operating system hides the abstractions of RPC from the user.

2.3.2 Types of RPC

These are the five types of the remote procedure call.

(1) **Synchronous RPC** : This is the normal method of operation. The client makes a call and does not continue until the server returns the reply.

(2) **Nonblocking RPC :** The client makes a call and continues with its own processing. The server does not reply.

(3) **Batch-mode RPC :** This is a facility for sending several client nonblocking calls in one batch.

(4) **Functions of Batch-mode RPC :**

(i) It minimizes the overhead involved in sending a request as it sends them over the network in one batch to the server.

(ii) This type of RPC protocol is only efficient for application that needs lower call rates.

(iii) It needs a reliable transmission protocol.

(5) **Broadcast RPC :** RPC clients have a broadcast facility, that is, they can send messages to many servers and then receive all the consequent replies.

Functions of Broadcast RPC

(1) Allows you to specify that the client's request message must be broadcasted.

(2) You can declare broadcast ports.

(3) It helps to reduce the load on the physical network

(4) **Callback RPC :** The client makes a nonblocking client/server call, and the server signals completion by calling a procedure associated with the client.

Functions of Callback RPC

(i) Remotely processed interactive application problems

(ii) Offers server with clients handle

(iii) Callback makes the client process wait

(iv) Manage callback deadlocks

(v) It facilitates a peer-to-Peer paradigm among participating processes.

2.3.3 Extended RPC Models

The extended RPC models use RPC for their communication. The extended RPC models like lightweight RPC (Doors), Asynchronous RPC, Deferred Synchronous RPC, and One-way RPC are discussed in this section.

(1) Lightweight RPC (Doors)	(2) Asynchronous RPC
(3) Deferred Synchronous RPC	(4) One-way RPC

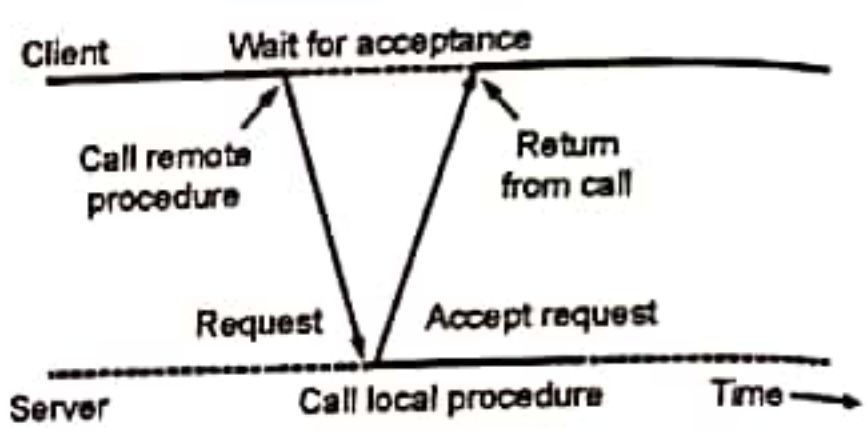
(1) Lightweight RPC (Doors)

- RPCs can be optimized by using lightweight RPC when the client and server processes are running on the same computer.
- Although the kernel tries to reduce overhead when it discovers that the packets are directed to itself, message construction overhead is still unavoidable.
- Therefore, the client simply sends a buffer from the client to the server over a shared memory region, inserting the RPC request and the parameters, and instructing the server to access it from there rather than sending an explicit network communication.
- Since memory is shared for message sending and receiving, explicit message forwarding is not required.
- The client sends the arguments to the stack, which is a trap for the kernel. The kernel, in turn, changes the memory map of the client in such a way that the memory location where the arguments were pushed in the stack is now available to the server.

- After that, the server retrieves the request from the memory region and begins processing it. Following the completion of the execution on the server's end, the reply is then transmitted back to the client in the same manner.
- It might be possible to avoid the overhead of sending messages over a network if one used this unstructured method of communication.
- The decision of using Lightweight RPCs can be made either at compile time or at run time.
- For compile-time support, linking against Lightweight RPCs is forced over the usual RPC using a compiler flag bit.
- The other option is to compile in both Lightweight and usual RPC support and choose one at runtime.
- It is then handled by the RPC runtime system; it can decide whether to send a message over TCP or use shared memory buffers.
- Doors in Solaris OS was the first implementation of Lightweight RPCs.

(2) Asynchronous RPC

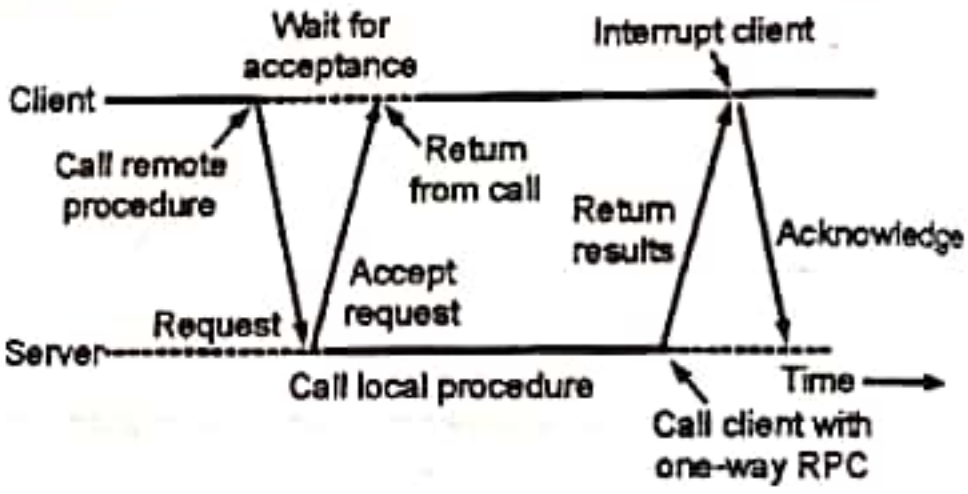
- Client makes an RPC call, and it waits only for an acknowledgement from the server and not the actual response.
- The server then processes the request asynchronously and sends back the response asynchronously to the client which generates an interrupt for the client to read the response received from the server. This is useful when the RPC call is a long-running computation on the server, meanwhile, the client can continue execution.



(186)Fig. 2.3.3 : Asynchronous RPC

(3) Deferred Synchronous RPC

- Here, the client and server interact through two asynchronous RPCs.
- The client sends an RPC request to the server, and the client waits only for acknowledgement of the received request from the server, post that the client carries on with its computation.
- Once the server processes the request, it sends back the response to the client which generates an interrupt on the client side, the client then sends a response received acknowledgement to the server.



(187)Fig. 2.3.4 : Deferred Asynchronous RPC

(4) One-way RPC

- The client sends an RPC request and doesn't wait for an acknowledgement from the server, it just sends an RPC request and continues execution.
- The reply from the server is handled through an interrupt generated on receipt of the response on the client side.
- The downside here is that this model is not reliable. If it is running on a non-reliable transport medium such as UDP, there will be no way to know if the request was received by the server.

2.3.3(A) Implementing RPC Mechanism

- The implementation of an RPC mechanism is built on the idea of stubs, which offer an abstraction of a procedure call that is entirely normal by hiding the interface to the underlying RPC System from clients.

- The implementation of an RPC mechanism usually involves the following five elements of the program: The Client, The Client Stub, The RPC Runtime, The Server Stub, and The Server.
- **The Client** is the initiating process for a remote procedure call. To invoke a remote procedure, the client makes a normal local call, which invokes the corresponding procedure in the client stub.
- **The Client Stub** is responsible for the following two responsibilities :
  (1) Upon receiving a call request from the client, it requests the local RPC Runtime to send a message containing the target procedure and its parameters to the server stub.
  (2) When it receives the result of a procedure's execution, it unpacks it and sends it to the client.
- **The RPC Runtime** manages the delivery of messages between client and server machines across the network. It manages retransmissions, acknowledgements, packet routing, and encryption. The client machine's RPC runtime gets the call request message from the client stub and then transmits it to the server machine. The RPC runtime on the server machine receives the message containing the procedure's execution result from the server stub and transmits it to the client machine.
- **The Server Stub** is responsible for the following two responsibilities :
  (1) When the server stub receives the call request message from the local RPC Runtime, it unpacks it and makes a perfectly normal call to run the proper server procedure.
  (2) The server stub requests the local RPC Runtime to transmit the result of procedure execution to the client stub upon receiving the result of procedure execution from the server.
- When the Server receives a call request from the server stub, it executes the necessary procedure and provides the result to the server stub.

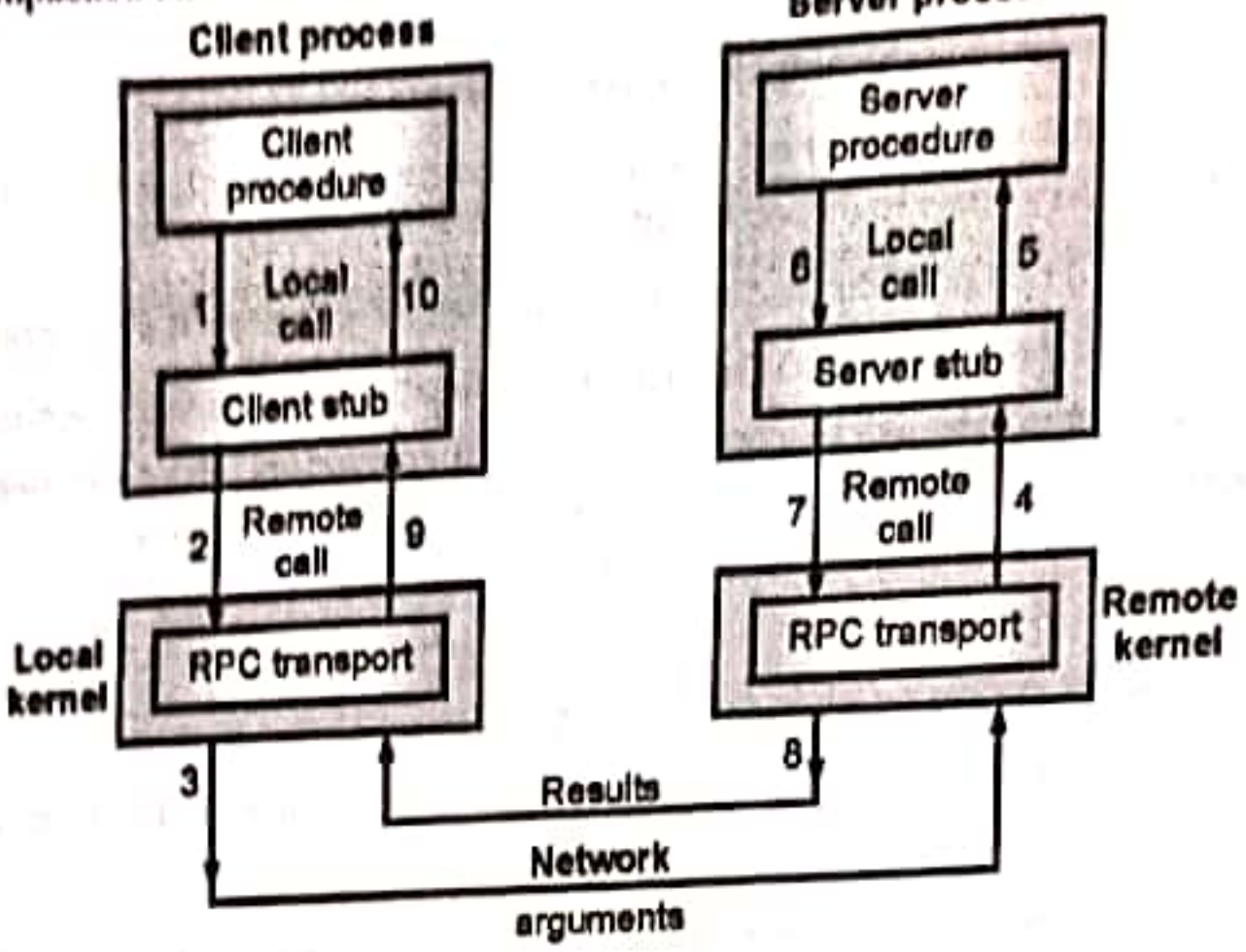
2.3.4 Working of RPC

- RPC is used to implement the client/server model. It is easier to program than sockets.
- When the client process makes a call to the remote function, the following steps are executed to complete the RPC (Fig. 2.3.5)
  (1) The client procedure makes a local procedure call to the client stub. A client stub is conceptually a collection of functions with the same name as the remote function. This function contains code that sends a request message to the server with the actual parameters to the remote function, waits for the response, reads the response, and returns the results. Stub compilers, such as **rpcgen**, can generate stubs automatically. During compilation, the stub is inserted into the code.
  (2) The client stub invokes network routines via system calls. It consists of preparing the message buffer, packing parameters into the buffer, constructing the message, and sending the message to the kernel.
  (3) The message is sent to the remote kernel by the local kernel in this step. The kernel is in charge of the process. The message is transferred to the kernel, and the destination address is determined. The network interface is then configured, and a timer is started to re-transmit the message.
  (4) When the message is received successfully by the remote kernel, it is transferred to the server stub via system calls. The packet is first validated, and then the stub to which the message is to be sent is determined. The message is copied onto the server stub if the stub is waiting.
  (5) The server stub unpacks the parameters and calls the server procedure once the control is passed to it. This is known as unmarshalling.
  (6) The process is carried out by the server, and the results are returned to the server stub.
  (7) Finally, the server stub packages the results into a message and sends it to the kernel.

(8) The results are then transmitted to the client's kernel by the server's kernel.

(9) The result is then passed to the client stub by the client's kernel.

(10) The results are unpacked and returned to the client procedure by the client stub.



(108)Fig. 2.3.5 : Working of RPC

(Courtesy : Distributed Component Architecture, G. Sudha Sadasivam)

2.3.5 Issues of Remote Procedure Call (RPC)

In an operating system, Remote procedure call or RPC faced some issues that must be addressed, such as :

(1) RPC Runtime	(2) Stub
(3) Binding	(4) The calling semantics associated with RPC

(1) RPC Runtime

The RPC runtime system is a collection of routines and services that handle the network communications that are at the core of the RPC mechanism. Client-side and server-side runtime systems code handle binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors during an RPC call.

(2) Stub

- The stub's purpose is to provide transparency to the programmer-written application code.
- On the client side, the stub handles the interface between the client's local procedure call and the runtime system, marshaling and unmarshalling data, invoking the RPC runtime protocol, and performing some of the binding steps if requested.
- On the server side, the stub acts as an interface between the runtime system and the server's local manager procedures.

(3) Binding

- It is how does the client know who to call and where to find the service? The most adaptable solution is to use dynamic binding and locate the server at runtime when the RPC is made for the first time.
- When the client stub is first invoked, it contacts a name server to determine the transport address of the server. The binding is made up of two parts :

(i) **Naming** : A server that provides a service export an interface for it. When an interface is exported, it is registered with the system so that clients can use it.

(ii) **Locating** : Before communication can begin, a client must import a (exported) interface.

(4) The calling semantics associated with RPC

It is mainly classified into the following types :

(i) **Retry request message** : When a server fails or the message is not received, whether to retry sending the request message.

(ii) **Duplicate filtering** : Remove the duplicate server requests.

(iii) **Retransmission of results** : To resend lost messages without re-executing server-side operations.

2.3.6 Advantages of RPC

Some of the advantages of RPC are as follows :

(1) Remote procedure calls support process-oriented and thread-oriented models.

(2) The internal message-passing mechanism of RPC is hidden from the user.

(3) The effort to re-write and re-develop the code is minimum in remote procedure calls.

(4) Remote procedure calls can be used in a distributed environment as well as the local environment.

(5) Many of the protocol layers are omitted by RPC to improve performance.

2.3.7 Disadvantages of RPC

Some of the disadvantages of RPC are as follows :

(1) The remote procedure call is a concept that can be implemented in different ways. It is not a standard.

(2) There is no flexibility in RPC for hardware architecture. It is only interaction based.

(3) There is an increase in costs because of the remote procedure call.

2.3.8 Parameter Passing in RPC

- In RPC, parameters are passed in two ways, namely, call by value and call by reference.
- **In call by value,** actual parameters are their data types are copied into a stack and passed to the called procedure.
- **In call by reference,** a pointer to data is passed instead of value to the called procedure at the server end. However, it is very difficult to implement as the server needs to keep track of the pointer to the data at the client's address space.

2.3.9 RPC Call Semantics

RPC has the same semantics as a local procedure call, the calling process calls the procedure, gives inputs to it, and then waits while it executes. When the procedure is finished, it can return results to the calling process. The different types of call semantics used in RPC systems are discussed here.

(1) **Possibly or May-Be Call Semantics :** It is the weakest semantics. Here, the caller is waiting until a predetermined timeout period and then continues with its execution. It is used in services where periodic updates are required.

(2) **Last-one Call Semantics :** Based on timeout, retransmission of call message is made. After the elapsing of the timeout period, the obtained result from the last execution is used by the caller. It sends out orphan calls. It finds its application in designing simple RPC.

(3) **Last-of-Many Call Semantics :** It is like Last-one Call semantics but the difference here is that it neglects orphan calls through call identifiers. A new call-id is assigned to the call whenever it is repeated. It is accepted by the caller only if the caller id matches the most recent repeated call.

(4) **At-least-once Call Semantics :** The execution of the call is there one or more times, but the results given to the caller are not specified. Here also retransmissions rely on the time-out period without giving a thought to orphan calls. In the case of nested calls, the result is taken from the first response message if there are orphan calls and others are ignored irrespective of the accepted response is whether from an orphan or not.

(5) **Exactly-once Call Semantics :** No matter how many times the call is transmitted, the potential of the procedure being conducted more than once is eliminated. When the server receives an acknowledgment from the client then only it deletes information from the reply cache.

2.3.10 Communication Protocols for RPC

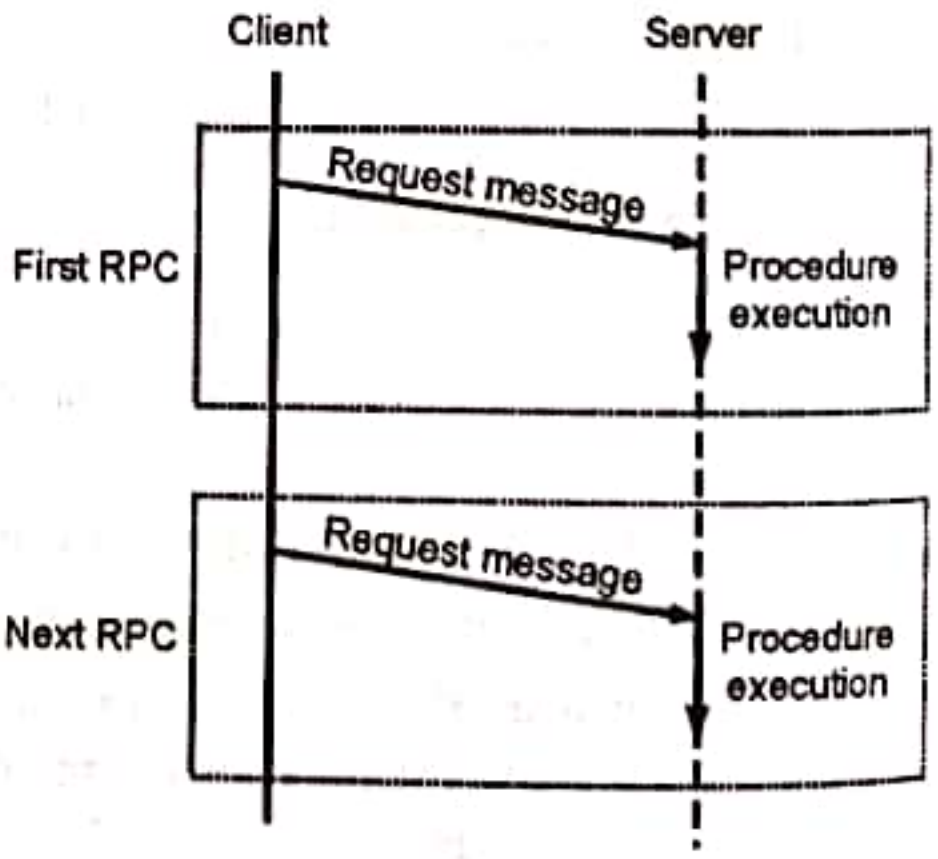
GQ. Explain different communication protocols used in RPC.

The following are the communication protocols that are used in RPC.

(1) Request Protocol
(2) Request/Reply Protocol
(3) The Request/Reply/Acknowledgement-Reply Protocol

(1) Request Protocol

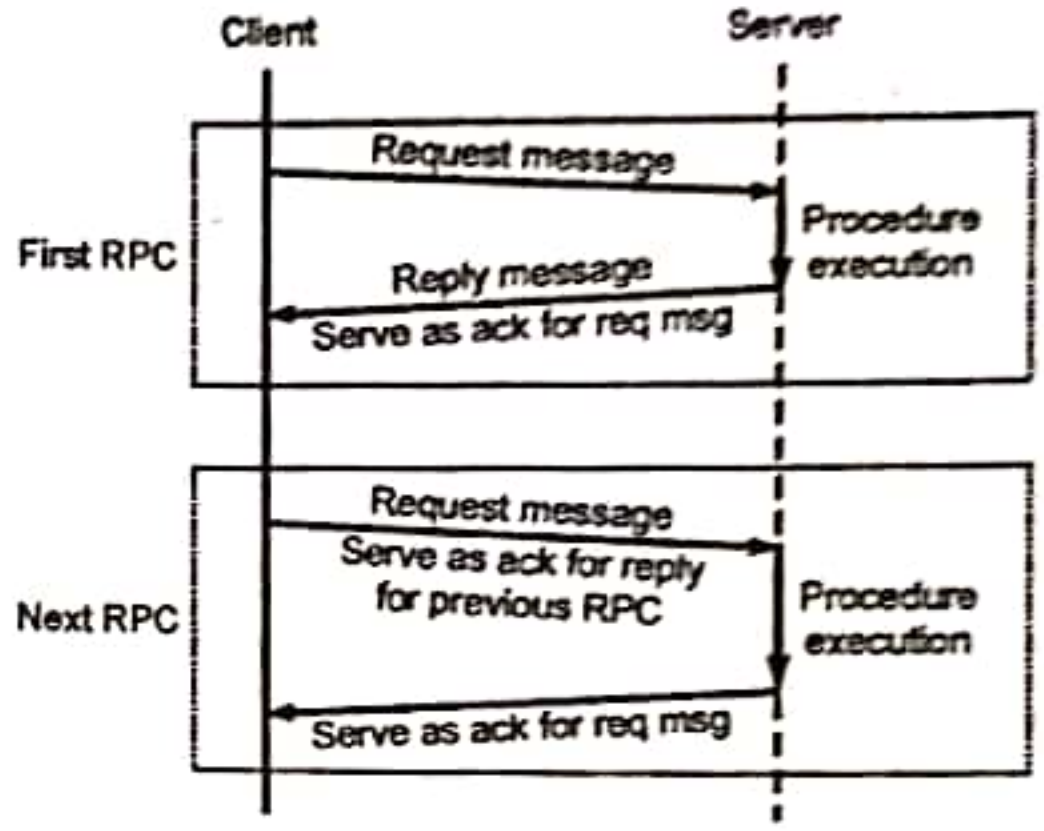
- The Request Protocol is also known as the R protocol.
- It is used in Remote Procedure Call (RPC) when a request is made from the calling procedure to the called procedure. After execution of the request, a called procedure has nothing to return and there is no confirmation required of the execution of a procedure.
- Because there is no acknowledgement or reply message, only one message is sent from client to server.
- A reply is not required so after sending the request message the client can further proceed with the next request.
- May-be call semantics are provided by this protocol, which eliminates the requirement for retransmission of request packets.
- Asynchronous Remote Procedure Call (RPC) employs the R protocol for enhancing the combined performance of the client and server. By using this protocol, the client need not wait for a reply from the server and the server does not need to send that.
- In an Asynchronous Remote Procedure Call (RPC) in case communication fails, the RPC Runtime does not retry the request. TCP is a better option than UDP since it does not require retransmission and is connection oriented.
- In most cases, asynchronous RPC with an unstable transport protocol is utilized to implement periodic update services. One of its applications is the Distributed System Window.



(189)**Fig. 2.3.6 : The request (R) Protocol**

(2) Request/Reply Protocol

- The Request-Reply Protocol is also known as the RR protocol.
- It works well for systems that involve simple RPCs.
- The parameters and result values are enclosed in a single packet buffer in simple RPCs. The duration of the call and the time between calls are both briefs.
- This protocol has a concept base of using implicit acknowledgements instead of explicit acknowledgements.
- Here, a reply from the server is treated as the acknowledgement (ACK) for the client's request message, and a client's following call is considered as an acknowledgement (ACK) of the server's reply message to the previous call made by the client.
- To deal with failure handling e.g. lost messages, the timeout transmission technique is used with RR protocol.
- If a client does not get a response message within the predetermined timeout period, it retransmits the request message.
- Exactly-once semantics is provided by servers as responses get held in reply cache that helps in filtering the duplicated request messages and reply messages are retransmitted without processing the request again.
- If there is no mechanism for filtering duplicate messages then at least-call semantics is used by RR protocol in combination with timeout transmission.

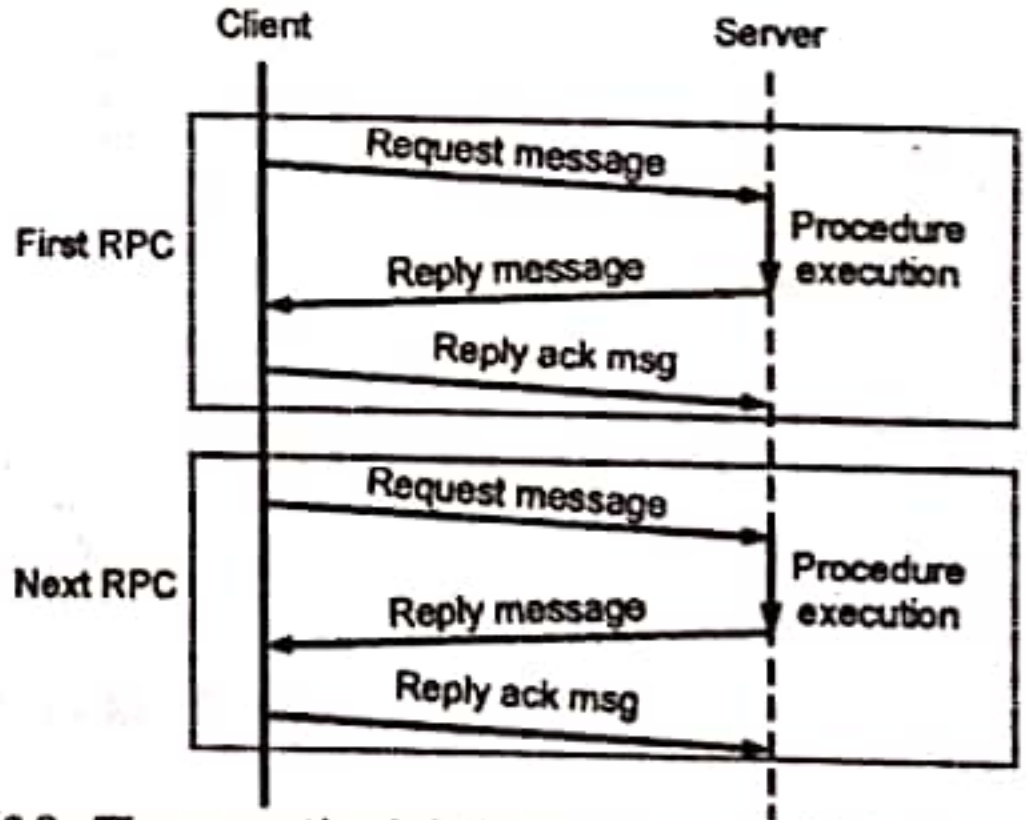


(1B10)Fig. 2.3.7 : The request/reply (RR) Protocol

(3) The Request/Reply/Acknowledgement-Reply Protocol

- This protocol is also known as the RRA protocol (request/reply/acknowledge-reply).
- Exactly-once semantics is provided by RR protocol which refers to the responses getting held in reply cache of servers resulting in loss of replies that have not been delivered.
- The RRA (Request/Reply/Acknowledgement-Reply) Protocol is used to get rid of the drawbacks of the RR (Request/Reply) Protocol.

- In this protocol, the client acknowledges the receiving of reply messages and when the server gets back the acknowledgement from the client then only deletes the information from its cache.
- Because the reply acknowledgement message may be lost at times, the RRA protocol requires unique ordered message identities. This keeps track of the acknowledgement series that has been sent.



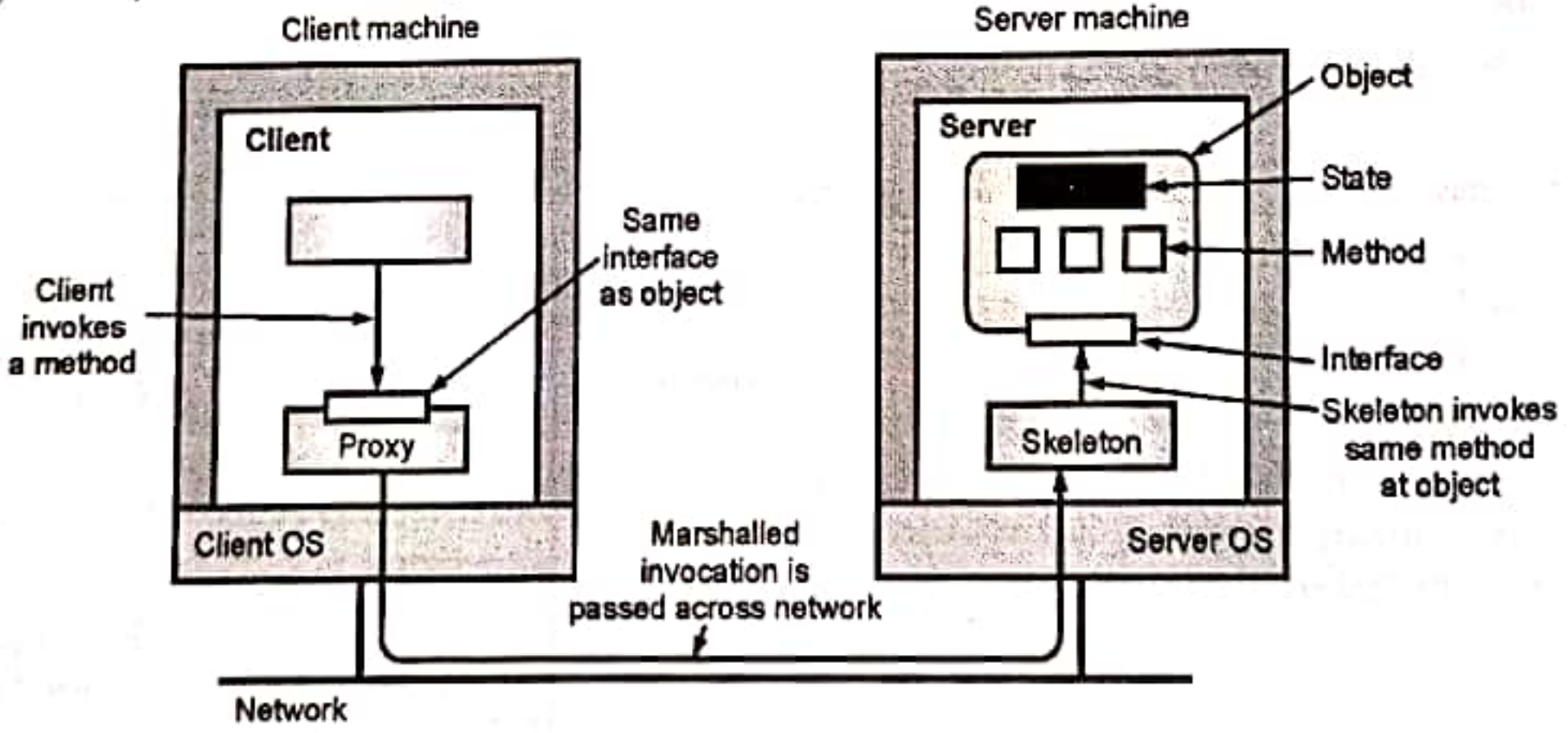
(1B11)Fig. 2.3.8 : The request/reply/acknowledge-reply (RRA) Protocol

2.4 REMOTE OBJECT INVOCATION

- Objects that are distributed across various address spaces, either on multiple computers connected by a network or even in different processes on the same computer, are referred to as distributed objects.
- These objects cooperate by exchanging information and invoking procedures. This frequently involves location transparency when remote objects appear to be local objects.
- The main method of distributed object communication is with remote method invocation (RMI) generally by message passing.
- In message passing, one object sends a message to another object in a remote machine or process to perform some tasks. The results are sent back to the calling object.

2.4.1 Distributed Object Model

- The term "distributed objects" often refers to software modules that are intended to operate together but are located either in different processes running on the same computer or in different machines connected by a network.
- Due to the logical partitioning of object-based programs and the physical distribution of objects among several processes or computers in a distributed system, the state of an object is determined by the values of its instance variables.
- Object systems that are distributed may use a client-server setup. Remote method invocation is used to call the methods of objects that are handled by servers and their clients.
- The primary property of an object is that it contains data, known as the state, and the actions performed on that data, known as the methods. An interface is used to make methods available.
- It is critical to recognize that there is no "legal" way for a process to access or alter an object's state other than via executing methods made accessible to it through an object's interface. An object can implement several interfaces.
- Similarly, given an interface specification, there may be numerous objects that implement it.
- This distinction between interfaces and the objects that implement them is critical in distributed systems. A rigorous separation enables us to establish an interface on one machine while the item itself is located on another.This organization, which is shown in Fig. 2.4.1, is commonly referred to as a distributed object.



(1B12)**Fig. 2.4.1 : Distributed Object**

- When a client connects to a distributed object, an implementation of the object's interface, known as a proxy, is loaded into the client's address space.
- In RPC systems, a proxy is like a client stub. It simply marshals method invocations into messages and unmarshals reply messages to return the method invocation's result to the client.
- The real object exists on a server system, where it provides the same interface that it does on the client machine.
- Incoming invocation requests are initially given to a server stub, which unmarshals them to make method invocations at the object's interface on the server.
- The server stub is also in charge of marshalling responses and passing reply messages to the client-side proxy.
- The server-side stub is often referred to as a skeleton since it offers a minimal way for the server middleware to access the user-defined objects. In practice, it frequently contains incomplete code in the form of a language-specific class that the developer must further specialize.
- Most distributed objects have a state that is not distributed; instead, it is stored on a single system. The object's state is only accessible from other computers through the interfaces it provides. These objects are also known as remote objects. Though a generic distributed object's state may be spread out across multiple computers, its clients wouldn't know this since the distribution is concealed by the object's interfaces.

2.4.2 Compile-Time versus Runtime Objects

- Objects in distributed systems can take many different forms.
- The most obvious form is compile-time objects, which are directly related to language-level objects such as those supported by Java, C++, or other object-oriented languages. In this case, an object is defined as a class instance.
- A class is a module-based description of an abstract type that contains data elements and operations on that data.
- Using compile-time objects in distributed systems often makes developing distributed applications much easier.
- In Java, for example, an object can be fully defined by its class and the interfaces that the class implements.
- When the class definition is compiled, it produces code that allows it to instantiate Java objects.
- The interfaces can be compiled into client-side and server-side stubs, allowing Java objects to be invoked remotely.
- A Java developer may be completely unaware of object distribution because he only sees Java programming code.
- The obvious disadvantage of compile-time objects is their dependence on a specific programming language. As a result, an alternative method of creating distributed objects is to do so explicitly during runtime.
- This approach is used in many object-based distributed systems because it is independent of the programming language used to write distributed applications.
- An application, in particular, can be built from objects written in multiple languages.
- When dealing with runtime objects, the implementation of the objects is largely unknown.
- A developer, for example, may decide to create a C library that contains a number of functions that can all work on the same data file.
- The key issue is figuring out how to make such an implementation appear to be an object whose methods can be called from a remote machine.
- An object adapter, which acts as a wrapper around the implementation with the sole purpose of giving it the appearance of an object, is a common approach.
- The term adapter comes from a design pattern that allows an interface to be transformed into what a client expects.
- An object adapter that dynamically binds to the aforementioned C library and opens an associated data file representing an object's current state is an example.

- Object adapters are critical components of object-based distributed systems. Objects are defined solely in terms of the interfaces they implement to make wrapping as simple as possible.
- An interface implementation can then be registered with an adapter, which can then make that interface available for (remote) invocations. The adapter will ensure that invocation requests are fulfilled, and thus provide its clients with an image of remote objects.

2.4.3 Persistent and Transient Objects

- A persistent object is one that exists even if it is not currently in the address space of any server process. A persistent object, in other words, is not dependent on its current server.
- In practice, this means that the server currently managing the persistent object can save the object's state to secondary storage before exiting.
- A newly launched server can later read the object's state from storage into its own address space and handle invocation requests. A transient object, on the other hand, is one that exists only for the duration of the server that hosts it.
- When that server terminates, the object stops existing.
- There used to be a lot of debate about whether or not to have persistent objects; some people believe that transient objects are sufficient.
- To avoid the debate over middleware, most object-based distributed systems simply support both types.

2.5 REMOTE METHOD INVOCATION (RMI)

- The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java.
- The RMI allows an object to invoke methods on an object running in another JVM.
- The RMI provides remote communication between the applications using two objects stub and skeleton.

2.5.1 Stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks :

(1) It initiates a connection with remote Virtual Machine (JVM),

(2) It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),

(3) It waits for the result

(4) It reads (unmarshals) the return value or exception, and

(5) It finally, returns the value to the caller.

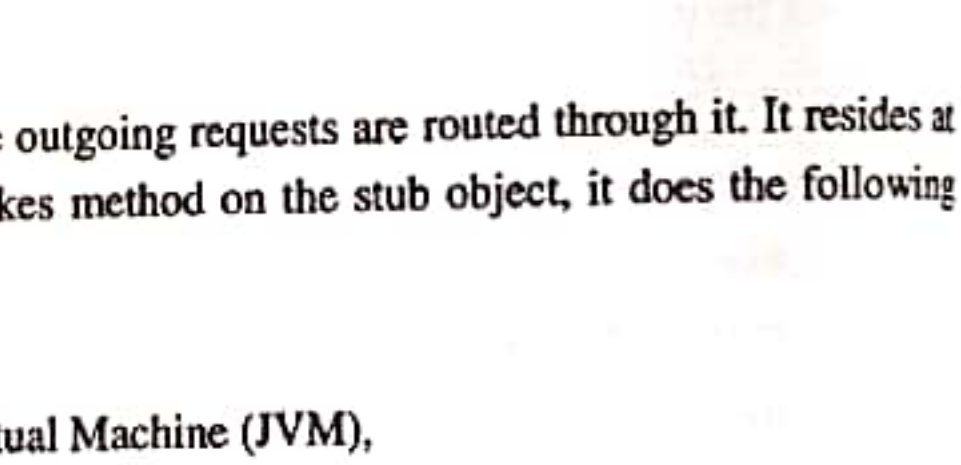
2.5.2 Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks :

(1) It reads the parameter for the remote method

(2) It invokes the method on the actual remote object, and

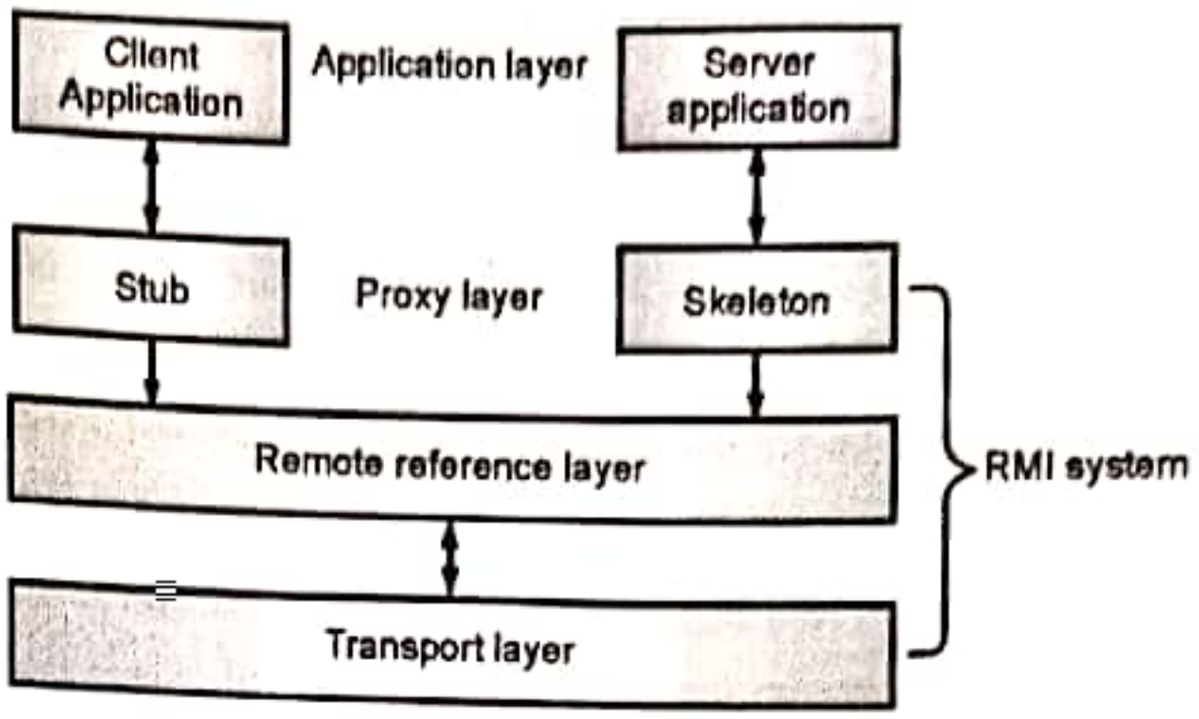
(3) It writes and transmits (marshals) the result to the caller.



(1813)**Fig. 2.5.1 : Skeleton and stub communication**

2.5.3 RMI Architecture

- Fig. 2.5.2 depicts the RMI architecture, which describes how remote objects behave and how parameters are passed between remote methods.
- Remote method invocation allows the programme to define the behavior and the code that implements the behavior separately, allowing them to run on separate JVMs. This principle allows clients to concentrate on the definition of a service while servers concentrate on providing the service.



(1B14)**Fig. 2.5.2 : RMI Architecture**

- A remote method can be invoked by referring to the remote object. The object is exported through a server application and a remote client, by either looking up the object name in the registry or by checking the value returned from the remote method.
- This object must implement at least one interface that is extended by the **java.rmi.Remote** interface. All interactions with the remote object will be performed through this interface only. Basically, this interface describes the methods, which can be invoked remotely, and the remote object then implements it.
- When a remote object is referenced, the object is not sent over the network to the client that requests it. Instead, a proxy object (often referred to as stub), which is a client-side proxy for the remote object, is sent.
- The client interacts only with this stub class. The stub class is responsible for handling data between the local and the remote systems. One remote object can be referenced by many clients. In that case, each client has its own stub object, which represents that remote object. But in no case is the remote object duplicated.
- A skeleton class is responsible for handing over the method, class and data to the actual object being referenced on the server side. This acts as the server side proxy for the object that is being exported.
- The actual implementation of the client-server application takes place at the Application layer. Any application that makes some of its methods available to its remote clients must first declare the methods in an interface that extends **java.rmi.Remote**, which is an empty interface.
- The main difference between the remote and normal interface is that the remote interface throws the remote exception.
- If a class is to implement these remote interfaces, it must first implement the **UnicastRemoteObject** in the **java.rmi.server** package.
- Finally, the application registers itself with a naming server or the registry with the name. This name can be used to contact the application and obtain reference to its remote objects.
- The client simply requests the remote object from the registry with the given name. The reference to the remote object is cast to one of its remote interfaces in order to call the remote methods. The high-level calls can then access and export remote objects.
- The Java RMI architecture consists of three layers: (i) Proxy Layer (or Stub/Skeleton layer) (ii) Remote Reference Layer (RRL) (iii) Transport Layer. These are described below:

(1) Proxy layer or stub/skeleton layer

- The proxy layer is responsible for managing the remote object interface between the server and client.
- In this layer, the stub class is the client-side Proxy for the remote object. stub is responsible for initiating a call to the remote object it represents and is the application interface to that object.
- It is also responsible for marshalling the method arguments to a marshal stream. This stream is a stream object which simply transports the parameters, exceptions and errors needed for the method dispatch and returns the results.
- Both the stub and skeleton classes use this stream to communicate with each other. Once the results are returned to the client stub, the results are un-marshalled.
- The skeleton class is similar to the stub class in many ways. The only difference is that it exists on the server side.
- The responsibility of the skeleton class is to send parameters to the method implementation and sending the return values.
- They are generated using the RMI stub compiler (RMIC). When the remote object from the naming server is requested, the stub class is downloaded to the client machine.
- The stub and skeleton classes implement the same interfaces as the remote object.

(2) Remote reference layer

- This layer gets a stream-oriented connection from the transport layer.
- It is responsible for dealing with the semantics of remote invocations.
- It is also responsible for handling duplicated objects and for performing implementation specific tasks with remote objects.

(3) Transport layer

- The transport layer is responsible for setting up the connection and transportation of data from one machine to another.
- The default connection is set up only in the TCP/IP protocol.

2.5.4 RMI Process

The process of RMI is done using the following steps. When the client process makes call to the remote function, the following steps are executed to complete the RPC.

(1) A remote object that wishes to provide a service registers in the RMI registry. The RMI register contains information about the remote object's name and reference.

(2) The client uses the object name to navigate to the RMI registry.

(3) The RMI registry returns the object reference (OR) of the remote object.

(4) Using a local procedure call, the client procedure calls the client stub. A client stub is conceptually a collection of functions that share the same name as a remote function. This function's core sends a request message to the server with the actual parameters to the remote function, waits for a response, reads the response, and returns the result. Stubs are automatically generated by stub compilers such as the RMI compiler and are included in code during compilation.

(5) The client stub makes system calls to the network routines. It consists of preparing the message buffer, packing parameters into the buffer, constructing the message, and sending the request to the kernel.

(6) The message is sent to the remote kernel by the local kernel in this step. The message is copied to the kernel, the destination address is determined, the network interface is configured, and the timer for message retransmission is started.

(7) The message is transferred to the RRL once it has been successfully received by the remote kernel. The packet is first validated, and then the stub and RRL to which the message is to be sent are determined..

(8) RRL then uses system calls to transfer the call to the server stub. If the stub is idle, the message is forwarded to the server stub.

(9) The server stub unpacks the parameters and calls the server procedure once the control is passed to it. This is referred to as unmarshalling.

(10) The server performs the work and returns the result to the server stub.

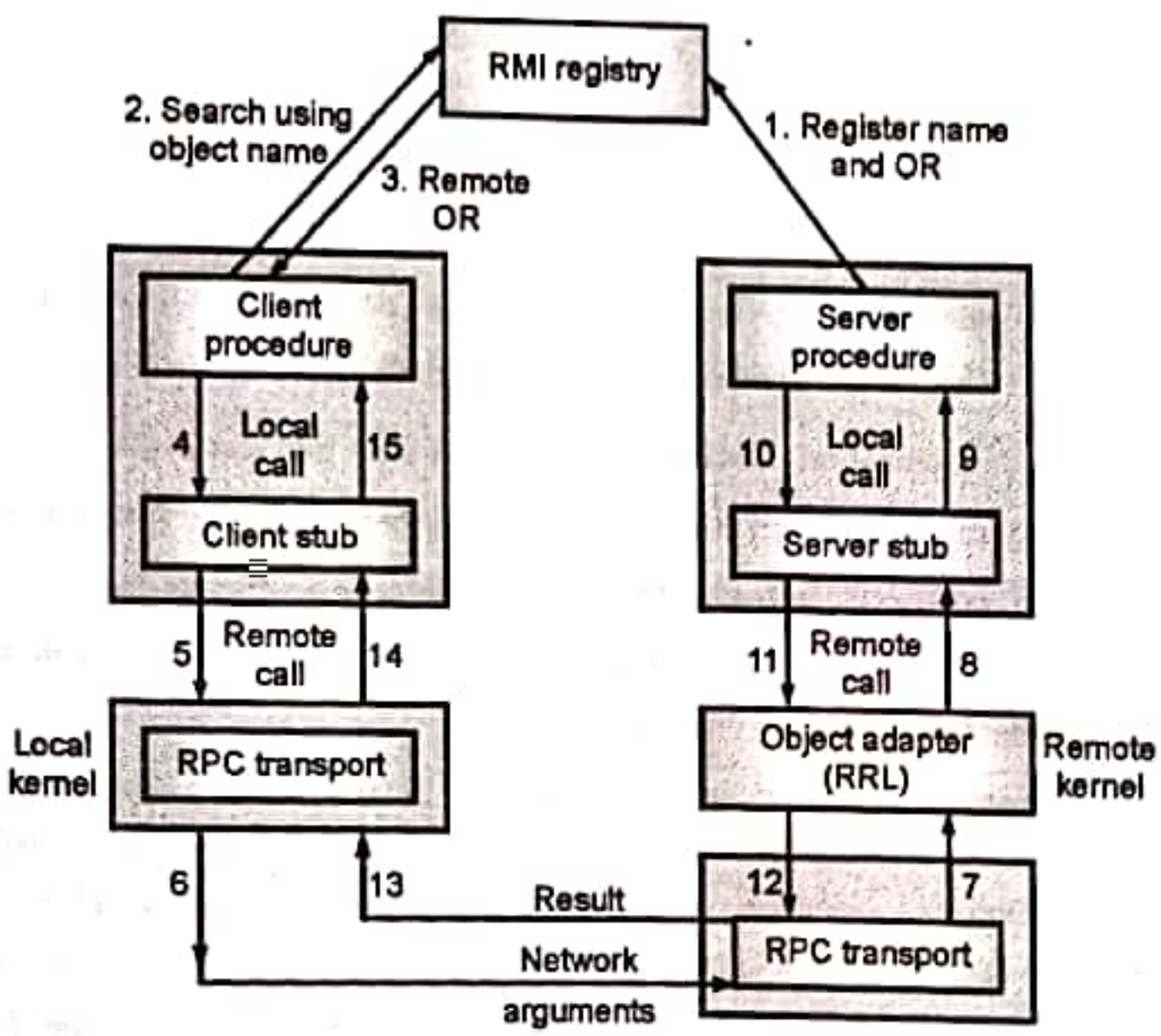
(11) The server stub then sends the results to RRL in a message.

(12) RRL communicates with the request-response protocol layer.

(13) The result is then transmitted to the Client's kernel by the server's kernel.

(14) The result is passed to the client stub by the Client's kernel.

The Client stub decompresses the results and returns them to the client procedure.



(1815)**Fig. 2.5.3 : Remote Method Invocation Process**

(Courtesy : Distributed Component Architecture, G. Sudha Sadasivam)

2.5.5 Advantages of RMI

(1) Simple and clean to implement that leads to more robust, maintainable and flexible applications.

(2) Distributed systems creations are allowed while decoupling the client and server objects simultaneously.

(3) It is possible to create zero-install client for the users.

(4) No client installation is needed except java capable browsers.

(5) At the time of changing the database, only the server objects are to be recompiled but not the server interface and the client remain the same.

2.5.6 Disadvantages of RMI

(1) Less efficient than Socket objects.

(2) Assuming the default threading will allow ignoring the coding, being the servers are thread- safe and robust.

(3) Cannot use the code out of the scope of java.

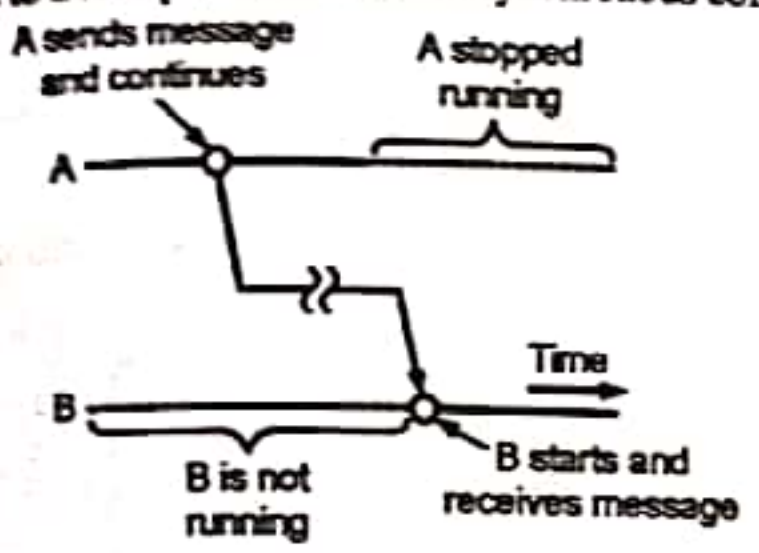
(4) Security issues need to be monitored more closely.

2.6 MESSAGE ORIENTED COMMUNICATION

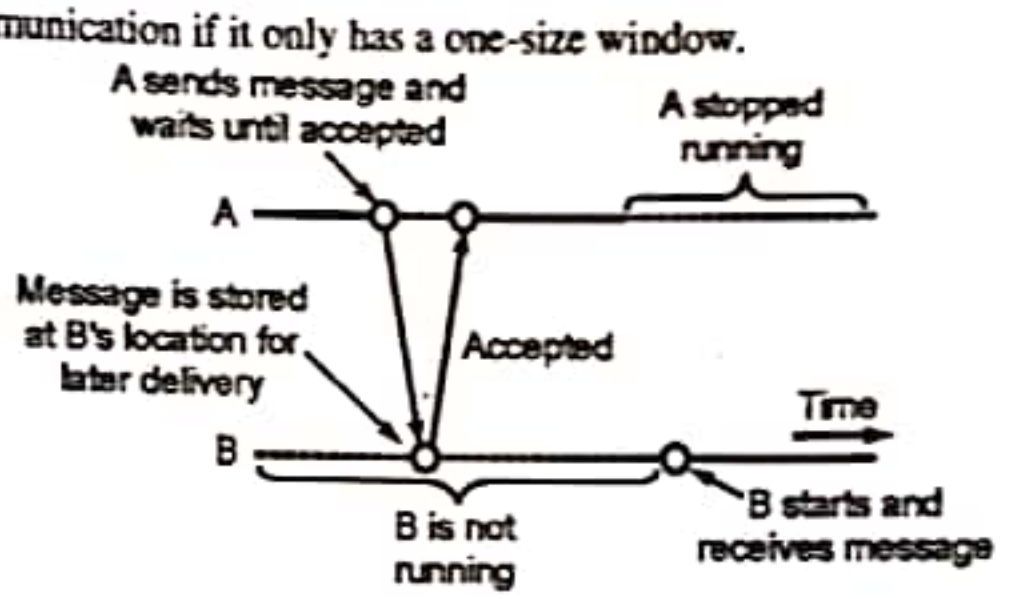
UQ. Give examples for the following message communication models. **(MU - May-19)**
(a) Transient Synchronous (b) Response based synchronous communication
(c) Transient Asynchronous (d) Persistent Asynchronous
(e) Receipt based communications

- Processes in a distributed system must communicate with each other in order to work together.
- Message Oriented Communication can be viewed along 2 axes: persistence (whether the system is persistent or transient); and synchronicity (whether it is synchronous or asynchronous).
- **Persistence :** In ***persistent*** communication, messages are stored at each intermediate hop along the way until the next node is ready to take delivery of the message. It is also called a store-and-forward based delivery paradigm. Example: Postal system (pony express), email, etc. In ***transient*** communication, messages are buffered only for small periods of time (as long as sending/receiving applications are executing). If the message cannot be delivered or the next host is down, it is discarded. Example: General TCP/IP communication.
- **Synchronicity :** In ***synchronous*** communication. the sender blocks further operations until some sort of an acknowledgement or response is received, hence the name blocking communication. In ***asynchronous*** or non-blocking communication, the sender continues execution without waiting for any acknowledgement or response. This form needs a local buffer at the sender to deal with it at a later stage.
- **Persistent synchronous communication :** The sender is blocked when it sends the message, waiting for an acknowledgement to come back. The message is stored in a local buffer, waiting for the receiver to run and receive the message. Some instant message applications, such as Blackberry messenger, are good examples. When you send out a message, the app shows you the message is "delivered" but not "read". After the message is read, you will receive another acknowledgement.

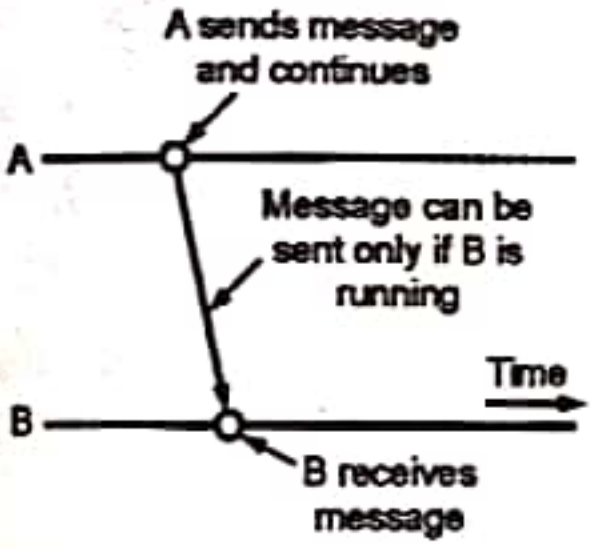
- **Transient asynchronous communication :** Since the message is transient, both entities must be running. Also, the sender doesn't wait for responses because it is asynchronous. UDP is an example.
- **Receipt-based transient synchronous communication :** The acknowledgement sent back from the receiver indicates that the message has been received by the other end. The receiver might be working on some other process.
- **Delivery-based transient synchronous communication :** The acknowledgement comes back to the sender when the other end takes control of the message. Asynchronous RPC is an example.
- **Response-based transient synchronous communication :** The sender blocks until the receiver processes the request and sends back a response. RPC is an example. There is no clear mapping of TCP to any type of communication. From an application standpoint, it maps to transient asynchronous communication. However, from a protocol standpoint, it maps to a receipt-based transient synchronous communication if it only has a one-size window.



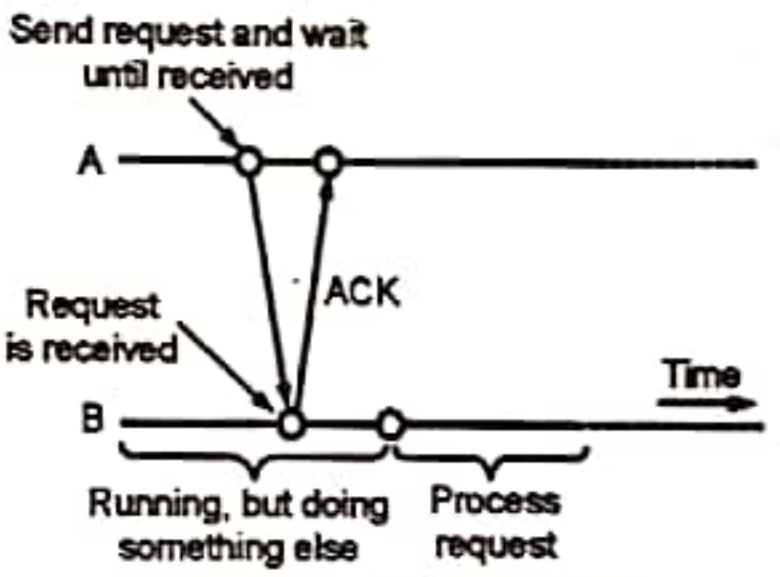
(a) Persistent asynchronous



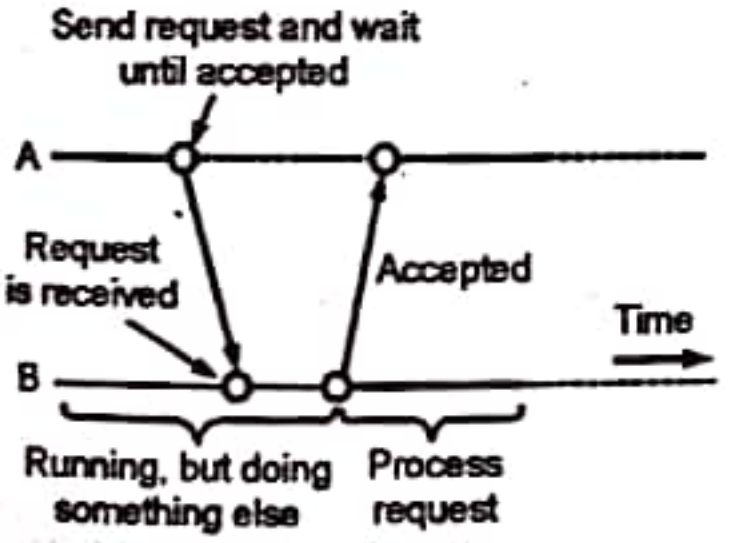
(b) Persistent synchronous



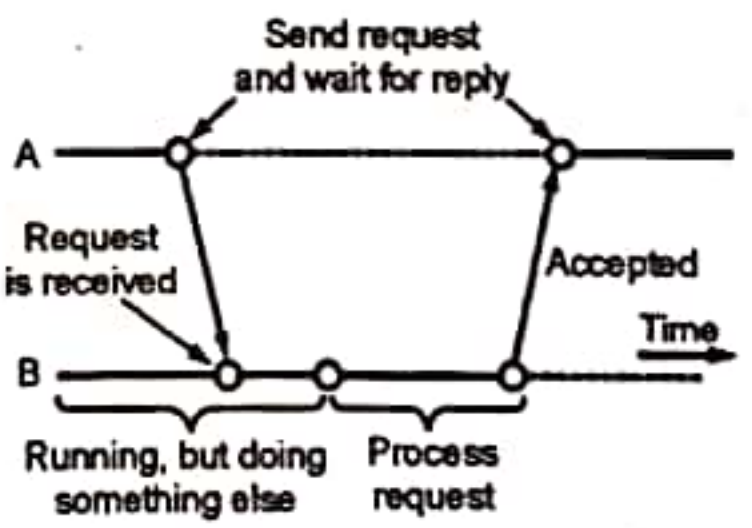
(c) Transient asynchronous



(d) Transient synchronous (Receipt-based)



(e) Transient synchronous (Delivery-based)



(f) Transient synchronous (Response-based)

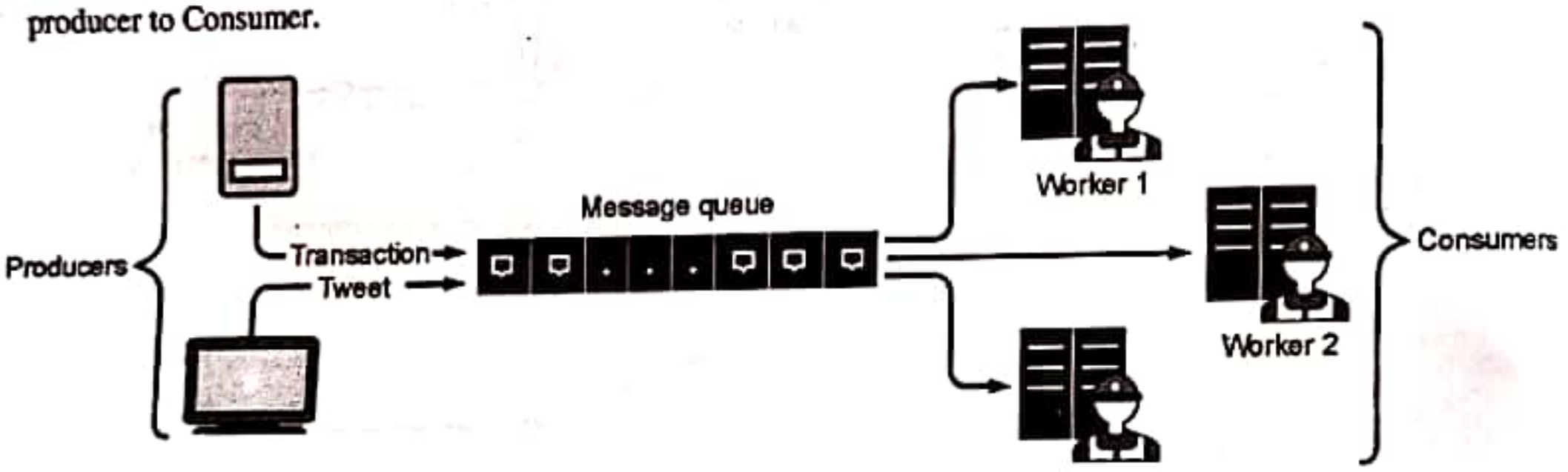
(1816)Fig. 2.6.1 : Types of Message Oriented Communications

2.6.1 Message Queue

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message queue provides an **asynchronous communications protocol,** which is a system that puts a message onto a message queue and does not require an immediate response to continuing processing.
- Let's make things more concrete by giving an example. When I need to communicate with a friend, I just get my phone and send her a WhatsApp message or an e-mail, so I don't have to disturb the person at that exact moment. I can send as many messages as needed as they will be buffered in her phone until she is able to get it and read the messages, taking any action required. This is a message queue between two people communicating with each other, but the same pattern applies to distributed systems.

The message queue is comprised of two terms as shown in Fig. 2.6.2 :

- **Message** : this is the object passed from the producer to the Consumer. The object can be requests, information, meta-data, etc.
- **Queue** : this is a temporary buffer that stores messages. It uses the First-In-First-Out method to pass the messages from producer to Consumer.



(1817)**Fig. 2.6.2 : Message Queue**

(1) The producer creates the message and sends it to the message queue if the Consumer is busy processing it immediately, the queue stores it until the Consumer is available.

(2) The Consumer retrieves the message from the queue and starts processing it.

(3) The message queue then temporarily locks the message to prevent it from being read by another consumer.

(4) After the Consumer completes the message processing, it deletes the message from the queue to prevent it from being read by other consumers.

2.6.2 Advantages of Message Queue

The following are the advantages of message queueing model.

(1) **Better Performance** : Message queues enable asynchronous communication, which means that the endpoints that are producing and consuming messages interact with the queue, not each other. Producers can add requests to the queue without waiting for them to be processed. Consumers process messages only when they are available. No component in the system is ever stalled waiting for another, optimizing data flow.

(2) **Increased Reliability** : Queues make your data persistent and reduce the errors that happen when different parts of your system go offline. By separating different components with message queues, you create more fault tolerance. If one part of the system is ever unreachable, the other can still continue to interact with the queue. The queue itself can also be mirrored for even more availability.

(3) **Granular Scalability** : Message queues make it possible to scale precisely where you need to. When workloads peak, multiple instances of your application can all add requests to the queue without risk of collision. As your queues get longer with these incoming requests, you can distribute the workload across a fleet of consumers. Producers, consumers and the queue itself can all grow and shrink on demand.

(4) **Simplifed Decoupling** : Message queues remove dependencies between components and significantly simplify the coding of decoupled applications. Software components aren't weighed down with communications code and can instead be designed to perform a discrete business function.Message queues are an elegantly simple way to decouple distributed systems, whether you're using monolithic applications, microservices or serverless architectures.

(5) **Break Up Apps** : Use message queues to decouple your monolithic applications. Rather than performing multiple functions within a single executable, multiple programs can exchange information by sending messages between processes, making them easier to test, debug, evolve and scale.

(6) **Migrate to Microservices** : Microservices integration patterns that are based on events and asynchronous messaging optimize scalability and resiliency. Use message queuing services to coordinate multiple microservices, notify microservices of data changes, or as an event firehose to process IoT, social and real-time data.

(7) **Shift to Serverless** : Once you've built microservices without servers, deployments onto servers, or installed software of any kind, you can use message queues to provide reliable, scalable serverless notifications, inter-process communications, and visibility of serverless functions and PaaS.

2.7 STREAM-ORIENTED COMMUNICATION

UQ. Explain Stream Oriented Communication with a suitable example. (MU - May 16)
UQ. Differentiate between Message oriented and Stream oriented communications. (MU - Dec. 18)
UQ. Compare and contrast Message oriented and Stream oriented communications. (MU - Dec. 19)

- Stream-oriented communication is usually used in audio and video streaming.
- A stream is transmitted by sending sequences of related messages.
- Message communication can be thought of as a request-response.
- Data streams are typically used to represent and transmit huge amounts of data Examples: JPEG images, MPEG movies.
- In Stream communication, you might start receiving data without it being requested. One aspect of stream communication is timing constraints.
- Playback freezes if data is not delivered on time, affecting the streaming experience (as opposed to browsing, where it does not matter as much). This form of communication is called isochronous communication. If we wish to watch 30 fps, a frame needs to come in every 33 ms. This implies that the end-to-end delay cannot be arbitrary, it must have an upper bound.
- In the case of continuous media, time is relevant to understand the data, e.g., audio streams.
- In the case of discrete media, time is not relevant to understand the data, e.g., still images
- Stream communication is not client pull-based (like message communication), it is server push-based.
- There are two kinds of stream-oriented communication: live streaming and stored streaming.
- Live streaming (Example: Video conferencing) is when both parties are live. For live streaming, the end-to-end delay must be of the order of 10 ms. Typical human perception needs 150ms.
- In Stored streaming(Example: YouTube, Netflix), one endpoint is actually a disk that stores data which the server streams. Here, the end-to-end delay does not matter as much, but the data rate should be a constant.

- Another way of splitting stream communication is as One-to-one (Example: Video conference) or One-to-many (Example: Webcast).
- Transmission of time-dependent information can be done in three different modes.
- **In asynchronous transmission mode,** data items of a stream are transmitted in sequence without further constraints, e.g., a file representing a still image.
- **In synchronous transmission mode,** data items of a stream are transmitted in sequence with a maximum end-to-end delay, e.g., data generation by a pro-active sensor
- **In isochronous transmission mode,** data items of a stream are transmitted in sequence with both a maximum and a minimum end-to-end delay, e.g., audio & video.

2.7.1 Quality of Service (QoS) in Stream Oriented Communication

- QoS is a way to encode the requirements of audio and video stream. Streaming needs a different architecture as compared to best-effort networks like the Internet that provide no guarantees.
- The requirements include :
  (1) **Bandwidth** : A HD video will need a minimum of a few Mbs per second.
  (2) **Maximum end-to-end delay bound** : Needs to be fixed (100ms) to avoid playback glitches (Skype, Facetime)
  (3) **Jitter** : Refers to the variation in the end-to-end delay. The fluctuation in the delay is jitter. We want to minimize jitter to ensure a steady data rate.
  (4) **Loss** : Refers to the loss in data packets. With TCP, loss is handled using re-transmission. Fundamentally, in video streaming, retransmission may not be an option (especially for live transmissions). Late data might be as good as no data for live streaming. Time requirements can sometimes be a trigger for complex re-transmissions - hence we need to lower this.

2.7.2 Comparison of Message-Oriented and Stream-Oriented Communication

Table 2.7.1 : Message-Oriented Vs Stream-Oriented Communication

Message-oriented communication	Stream-oriented communication
UDP (user datagram protocol) uses message-oriented communication	TCP (transmission control protocol) uses stream-oriented communication
Data is sent by application in discrete packages called messages.	Data is sent with no structure.
Communication is connectionless, data is sent without any setup.	Communication is connection oriented, connection established before communication.
It is unreliable, data delivery without acknowledgement.	It is reliable, data acknowledged.
Retransmission is not performed.	Lost data is reframed automatically.
Low overhead.	High overhead.
No flow control.	Flow control using sent protocol like sliding window.
Transmission speed is very high as compared to stream oriented.	Transmission speed is lower as compared to message oriented.
Suitable for applications like e-mail system where data must be persistent through delivered late.	Suitable for applications like audio, video where speed is critical than loss of messages.

2.8 GROUP COMMUNICATION

GQ Explain group communication in detail with its types.

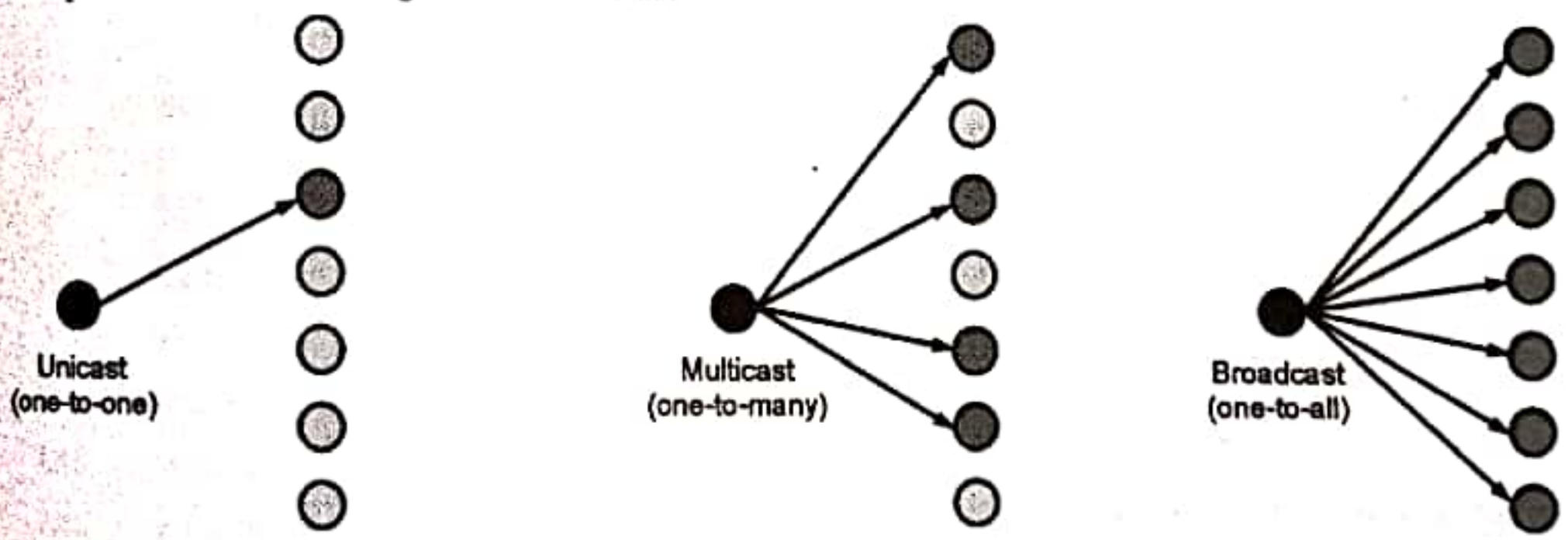
- Communication between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes.
- When one source process tries to communicate with multiple processes at once, it is called **Group Communication.**
- A group is a collection of interconnected processes with abstraction. This abstraction is to hide the message passing so that the communication looks like a normal procedure call.
- Group communication also helps the processes from different hosts to work together and perform operations in a synchronized manner, therefore increasing the overall performance of the system.

2.8.1 Types of Group Communication in a Distributed System

(1) **Broadcast Communication :** It is used when the host process tries to communicate with every process in a distributed system at the same time. Broadcast communication comes in handy when a common stream of information is to be delivered to every process in the most efficient manner possible. Since it does not require any processing whatsoever, communication is very fast in comparison to other modes of communication. However, it does not support many processes and cannot treat a specific process individually.

(2) **Multicast Communication** : It is used when the host process tries to communicate with a designated group of processes in a distributed system at the same time. This technique is mainly used to find a way to address problem of a high workload on host system and redundant information from process in system. Multitasking can significantly decrease time taken for message handling.

(3) **Unicast Communication** : It is used when the host process tries to communicate with a single process in a distributed system at the same time. Although, same information may be passed to multiple processes. This works best for two processes communicating as only it must treat a specific process only. However, it leads to overheads as it must find exact process and then exchange information/data.



(1818)**Fig. 2.8.1 : Types of Group Communication**

2.8.2 Group Communication Properties

- An important property in the group communication mechanism is atomicity, also known as an **all-or-nothing** property.
- With this property, the process that sends the message to a group will receive an error message if one or more members of the group have a problem receiving it.
- The ordering property of the messages is responsible for controlling the sequence of messages to be delivered.

- Some types of message ordering :
  - (1) **No order :** Messages are sent to the group without concern for ordering.
  - (2) **FIFO ordering :** Messages are delivered in the order in which they were sent.
  - (3) **Casual ordering :** Messages are sent after receiving another message.
  - (4) **Total ordering :** All group members receive all messages in the same order.

2.8.3 Design Issues in Group Communication

A number of design alternatives for group communication are available. These will affect how the groups behave and send messages.

(1) **Closed group vs. open group :** With closed groups, only the group members may send a message to the group. This is useful when multiple processes need to communicate with others in solving a problem, such as in parallel processing applications. The alternative is open groups, where non-members can send a message to a group. An example use of this type of group is an implementation of a replicated server (such as a redundant filesystem).

(2) **Peer groups vs. hierarchical groups :** With peer groups, every member communicates with each other. The benefits are that this is a decentralized, symmetric system with no point of failure. However, decision-making may be complex since all decisions must be made collectively (a vote may have to be taken). The alternative is hierarchical groups, in which one member plays the role of a group coordinator. The coordinator makes decisions on who carries out requests. Decision-making is simplified since it is centralized. The downside is that this is a centralized, asymmetric system and therefore has a single point of failure.

(3) **Centralized group membership vs. distributed membership :** If the control of group membership is centralized, we will have one group server that is responsible for getting all membership requests. It maintains a database of group members. This is easy to implement but suffers from the problem that centralized systems share – a single point of failure. The alternative mechanism is to manage group membership in a distributed way where all group members receive messages announcing new members (or the leaving of members).

Descriptive Questions

Q. 1 Define Inter-process Communication. Explain different types of Communications.

Q. 2 Explain the concept of Message Passing Interface (MPI) in detail.

Q. 3 Define Remote Procedure Call (RPC). Explain the working of RPC in detail.

Q. 4 Explain different call semantics in RPC.

Q. 5 Define Remote Object Invocation. Explain the working of RMI in detail.

Q. 6 Explain different communication protocols used in RPC.

Q. 7 Explain Stream Oriented Communication with a suitable example.

Q. 8 Differentiate between Message oriented and Stream oriented communications.

Chapter Ends...

□□□

MODULE 3

CHAPTER 3

Synchronization

3.1 INTRODUCTION

- Distributed System is a collection of computers connected via a high-speed communication network.
- In the distributed system, the hardware and software components communicate and coordinate their actions by message passing.
- Each node in distributed systems can share its resources with other nodes. So, there is a need for proper allocation of resources to preserve the state of resources and help coordinate between the several processes.
- To resolve such conflicts, synchronization is used. Synchronization in distributed systems is achieved via clocks.
- Synchronization is coordination with respect to time and refers to the ordering of events and execution of instructions in time.
- It is often important to know when events occurred and in what order they occurred.

3.2 CLOCK SYNCHRONIZATION

UQ. Explain Berkeley Physical Clock Algorithm. (MU - May 18)

UQ. What are Physical Clocks ? Explain any one Physical Clock Synchronization Algorithm. (MU - May 22)

- Clock synchronization is the mechanism to synchronize the time of all the computers in the distributed environments or system.
- Clock synchronization deals with understanding the temporal ordering of events produced by concurrent processes.
- It is useful for synchronizing senders and receivers of messages, controlling joint activity, and serializing concurrent access to shared objects.
- The goal is that multiple unrelated processes running on different machines should be in agreement with and be able to make consistent decisions about the ordering of events in a system.
- Another aspect of clock synchronization deals with synchronizing time-of-day clocks among groups of machines.
- In this case, we want to ensure that all machines can report at the same time, regardless of how imprecise their clocks may be or what the network latencies are between the machines.
- Assume that there are three systems present in a distributed environment. To maintain the data i.e., to send, receive and manage the data between the systems at the same time in a synchronized manner you need a clock that must be synchronized. This process to synchronize data is known as Clock Synchronization.
- Synchronization in a distributed system is more complicated than in a centralized system because of the use of distributed algorithms.
- Properties of Distributed algorithms to maintain Clock synchronization:
  (i) Relevant and correct information will be scattered among multiple machines.
  (ii) The processes make the decision only on local information.
  (iii) Failure of the single point in the system must be avoided.
  (iv) No common clock or other precise global time exists.
  (v) In distributed systems, the time is ambiguous.
- As the distributed systems havetheir own clocks, the time among the clocks may also vary. So, it is possible to synchronize all the clocks in a distributed environment.

3.2.1 Physical Clock

- A computer clock typically has a quartz crystal, a counter register, and a constant register.
- The constant register stores a constant value dependent on the quartz crystal's oscillation frequency.
- The counter register decrements by 1 for each quartz crystal oscillation.
- When the counter register reaches zero, an interrupt is issued, and its value is reinitialized to the constant register. Each interrupt is termed a clock tick.
- To make the computer clock work like a regular clock, the following steps are taken:
  (i) The constant register value is set to 60 clock ticks per second.
  (ii) Computer clocks are synchronized with real-time (external clock). Two other values are recorded in the system: a defined starting date and time and the number of ticks.
  (iii) In UNIX, time starts at 0000 on January 1, 1970.
  (iv) The system prompts the user to enter the current date and time during bootup.
  (v) The system calculates the ticks after the fixed starting date and time from the entered value.
  (vi) To keep the clock running, the interrupt service routine advances the tick count at each clock tick.

3.2.2 Drifting of Clocks

- A clock always runs at a constant rate because its quartz crystal oscillates at a fixed frequency. However, due to crystal variances, the rates at which two clocks run are generally different.
- The difference in oscillation period between two clocks may be quite minor, but the difference collected over many oscillations leads to an observable difference in the timings of the two clocks, regardless of how precisely they were initialized to the same value. As a result, as time passes, a computer clock drifts away from the real-time clock that was utilized for its initial setup.
- The drift rate for quartz crystal clocks is around 10^{-6}, resulting in a difference of 1 second every 1,000,000 seconds, or 11.6 days. To keep a computer clock from becoming defective, it must be resynchronized with the real-time clock on a regular basis. Even non-faulty clocks do not always keep accurate time.
- A clock is considered non-faulty if the amount of drift from real-time for every given finite time interval is restricted. After synchronization with a perfect clock, slow and fast clocks drift in opposing directions from the perfect clock, as shown in Fig. 3.2.1.
- This is because dC/dt < 1 for slow clocks and dC/dt> 1 for fast clocks where C denotes the time value of a clock and t is the real time.

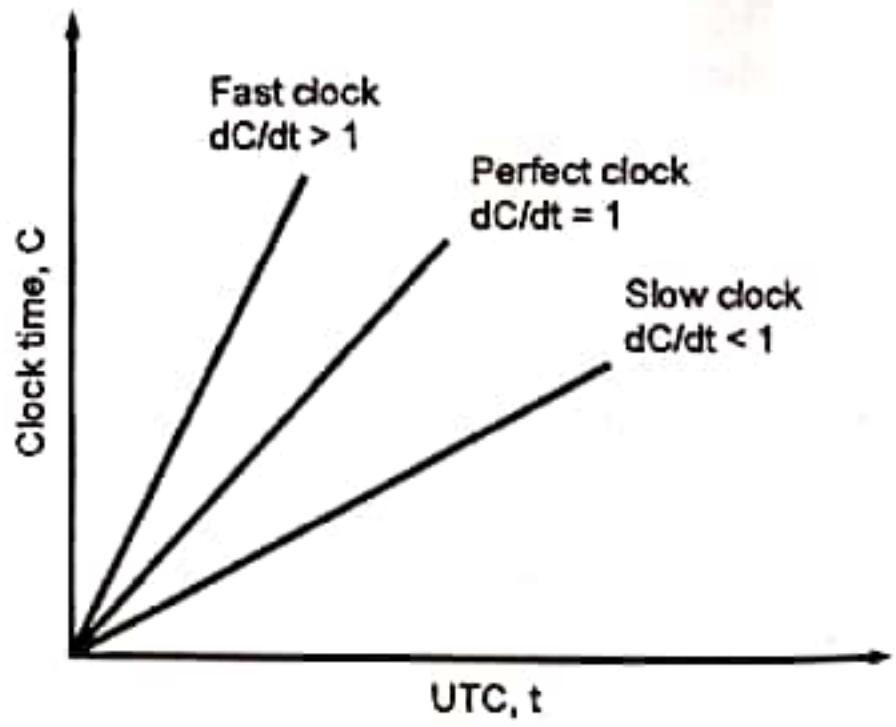


Fig. 3.2.1 : The relation between clock time and UTC when clocks tick at different rates

- A distributed system requires the following types of clock synchronization:
  1. *Synchronization of the computer clocks with real-time (or external) clocks* : This type of synchronization is primarily essential for real-time applications. That is, external clock synchronization enables the system to exchange information about the timing of events with other systems and users. The Coordinated Universal Time (UTC)is a popular external time source for synchronizing computer clocks with real time.

2. Mutual (or internal) synchronization of the clocks of different nodes of the system : This sort of synchronization is mostly necessary for applications that demand a consistent view of time across all nodes of a distributed system, as well as for measuring the duration of distributed activities that end on a node other than the one on which they begin.

3.2.3 Issues in Clock Synchronization

A simple method of clock synchronization is that each node must send a request message 'time=?' to the real-time server. The node gets a reply message with 'time=t'. This method has the following issues:

(a) The ability of each node to read another node's clock value. This can raise errors due to delays in message communication between nodes. Delay can be computed by computing the time needed to prepare, transmit and receive an empty message in the absence of transmission errors and system load.

(b) Time must never run backward since it may lead to the repetition of events or transactions creating disorder in the system. Time running backward is just a perception, not actually it goes backward.

Reasons for Delay in Synchronization

There are many reasons for a communication delay that needs to be minimized and get nearby accurate time:

(a) **Communication Link Failure :** For example, while sending a request message, the communication link is operational, and the message is sent to the server. When receiving a message, the communication link may fail owing to a break. And the client may not receive a response message. Following recovery, the client receives a response containing a false time value.

(b) **Fault Tolerance :** During message passing, if any component fails, it may cause an inaccurate reading of clock time. So, the system should be fault tolerant which can work in a faulty situation and minimize the clock drift value.

(c) **Propagation Time :** Due to heavy traffic or congestion in the network, it may cause a large propagation time from server to client. It may cause an inaccurate reading of the clock value in the reply.

(d) **Non-Receipt of Acknowledgement :** It may be possible that due to the above reasons,the client will not get a reply within a round trip time, and therefore it sends multiple requests to the server for synchronization.

(e) **The Bandwidth of the Communication Link :** Due to the low bandwidth of the communication link, congestion may occur in the network. Therefore, request for time will not be able to reach the server or the reply message will fail to reach client which will affect clock synchronization.

3.2.4 Clock Synchronization Algorithms

- Clock synchronization is a method of synchronizing the clock values of any two nodes in a distributed system with the use of an external reference clock or internal clock value of the node.
- During the synchronization, many factors affect on a network. As discussed above, these factors need to be considered before correcting the actual clock value. Based on the approach, clock synchronization algorithms are divided as Centralized Clock Synchronization and Distributed Clock Synchronization Algorithm.

3.2.4(A) Centralized Algorithms

There is one time-server node which is used as a reference time for the following centralized algorithms :

1. Passive Time Server

Each node periodically sends a request message 'time=?' to the time-server to get accurate time. Time-server responds with 'time=t'. Assume that the client node sends a message at time t_0 and get a reply at time t_1, then message propagation time from server to client node is $(t_1 - t_0) / 2$.

Client node receives a reply, and it adjusts its clock time to $t + (t_1 - t_0) / 2$. For a more accurate time, there is a need to calculate accurate message propagation time. There are two methods proposed to improve the estimated time value.

(a) Additional information available : Assume that to handle interrupt and to process request message sent by the client, the time-server takes time t_i. Hence, a better estimated time is $(t_1 - t_0 - t_i)/2$. Therefore, clock is readjusted to $t + (t_1 - t_0 - t_i) / 2$.

(b) No additional information available (Cristian Method) : This method uses time-server connected to a device that receives a signal from a source of UTC to synchronize nodes externally. A simple estimated time $t + (t_1 - t_0)/2$, is accurate if nodes are on same network. For nodes on different networks, the minimum transmission time can be estimated as follows:

The time t_{min} refers to the time needed by the server to prepare reply message. And the same time t_{min} is needed by client process to dispatch reply message as shown in the Fig. 3.2.2.

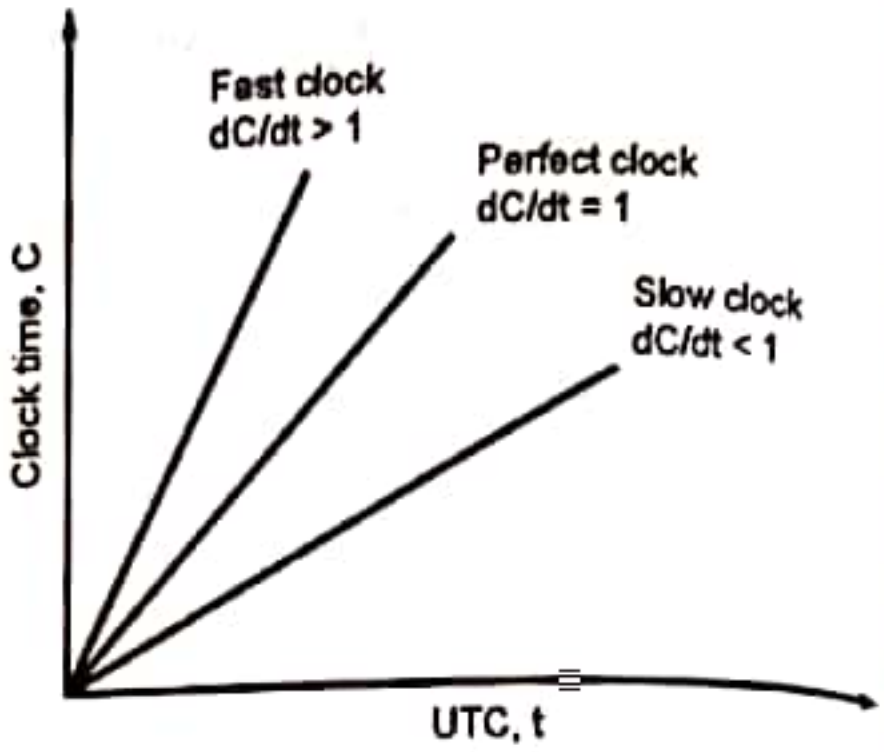


Fig. 3.2.2 : Cristian's Method

When the reply message arrives, the time taken by server's clock is in the range of $(t_1-t_0)/2 - 2t_{min}$. Therefore, the accuracy of clock time is $\pm (t_1-t_0)/2 - 2t_{min}$. There are some limitations as follows:

(i) There is a single time-server that might fail and thus synchronization temporarily unavailable.

(ii) There may be faulty time-server that reply with spurious time or an imposter time-server that replied with incorrect time.

Example:

- Send request at 5:08:15.100 (t_0)
- Receive response at 5:08:15.900 (t_1)
- Response contains 5:09:25.300 (t)
- Elapsed time is $t_1 - t_0$ [5:08:15.900 – 5:08:15.100 = 800ms)
- Assume timestamp was generated at 400ms ago
- Based on Cristian's algorithm, set time to t + elapsed time [5:09:25.30 + 400 = 5:09:25.700]

2. Active Time Server

Time-server periodically broadcasts its clock time as 'time=t'. Other nodes receive messages and readjust their local clock accordingly. Each node assumes message propagation time = t_a and readjust clock time = $t + t_a$. There are some limitations as follows:

(i) Due to communication link failure message may be delayed and clock readjusted to incorrect time.

(ii) Network should support broadcast facility.

3. Berkeley Algorithm

- Berkeley's Algorithm is a clock synchronization technique used in distributed systems.
- The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess a UTC server. This algorithm overcomes limitation of faulty clock and malicious interference in passive time-server and also overcomes limitation of active time-server algorithm.
- Time-server periodically sends a request message 'time=?' to all nodes in the system. Each node sends back its time value to the time-server.

- Time-server has an idea of message propagation to each node and readjust the clock values in reply message based on it. Time-server takes an average of other computer clock's value including its own clock value and readjusts its own clock accordingly.
- It avoids reading from unreliable clocks. For readjustment, time-server sends the factor by which other nodes require adjustment.
- The readjustment value can either be +ve or –ve.
- The following Fig. 3.2.3 illustrates how the master sends requests to slave nodes.

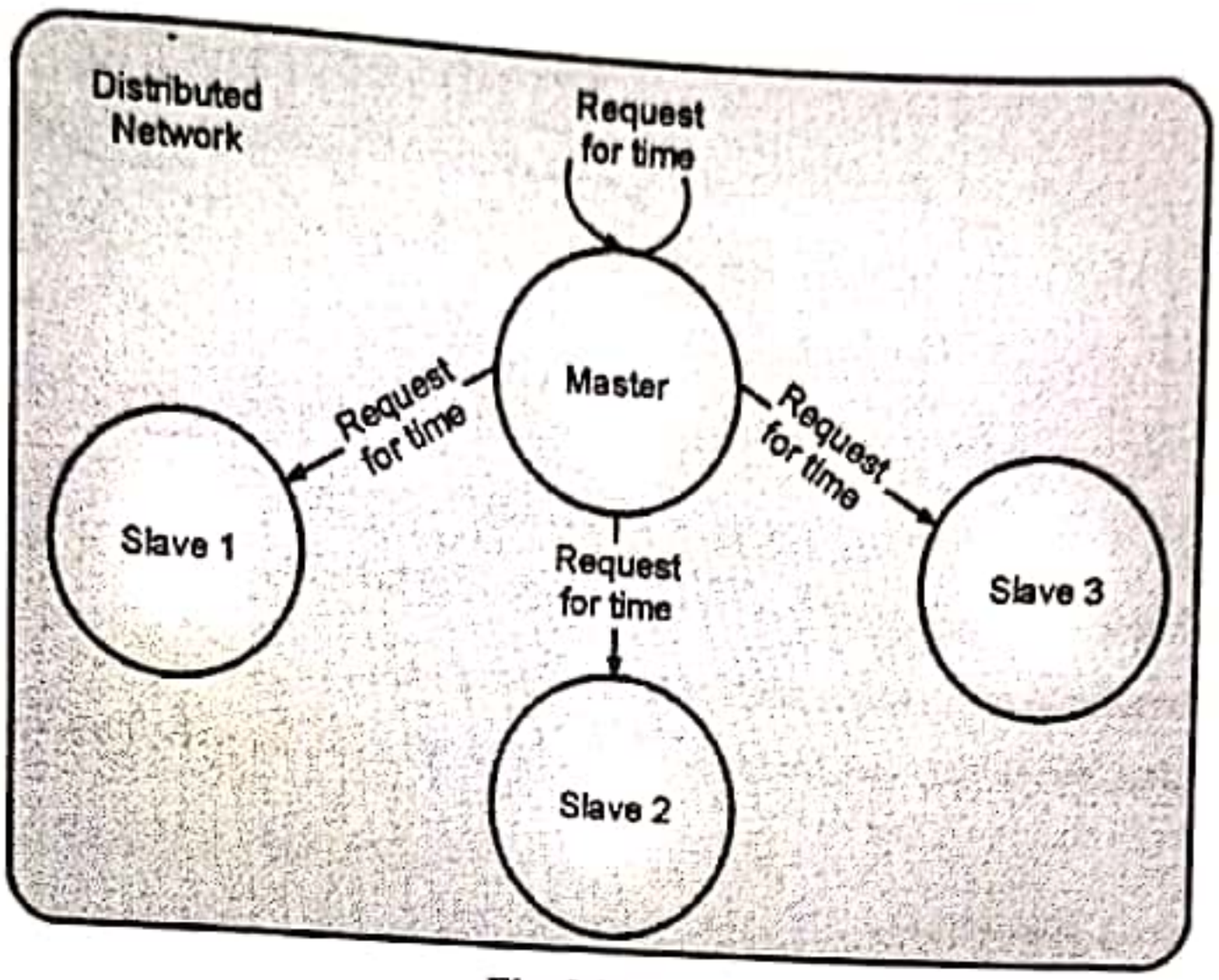


Fig. 3.2.3

- The following Fig. 3.2.4 illustrates how slave nodes send back time given by their system clock.

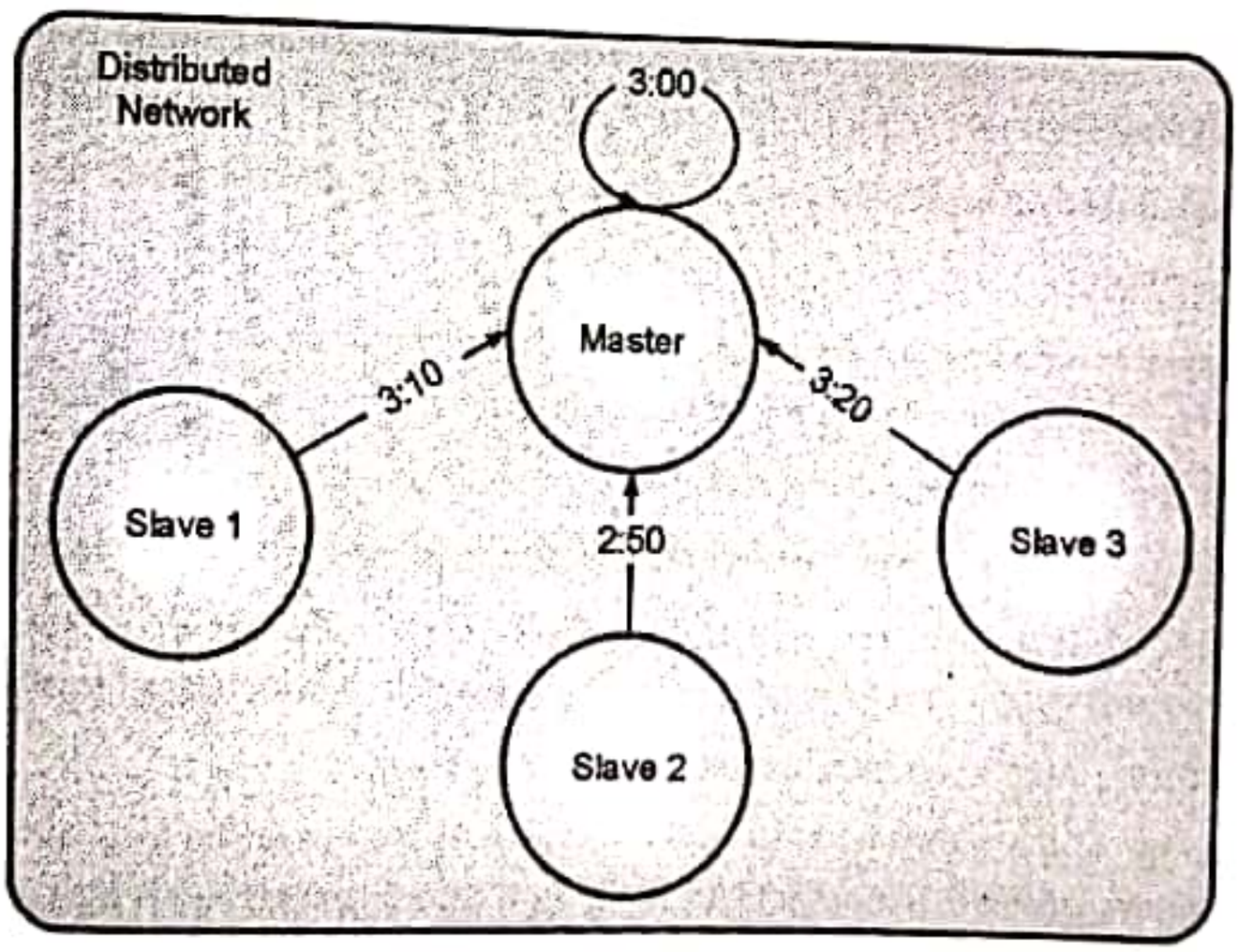


Fig. 3.2.4

- The following Fig. 3.2.5 illustrates the last step of Berkeley's algorithm.

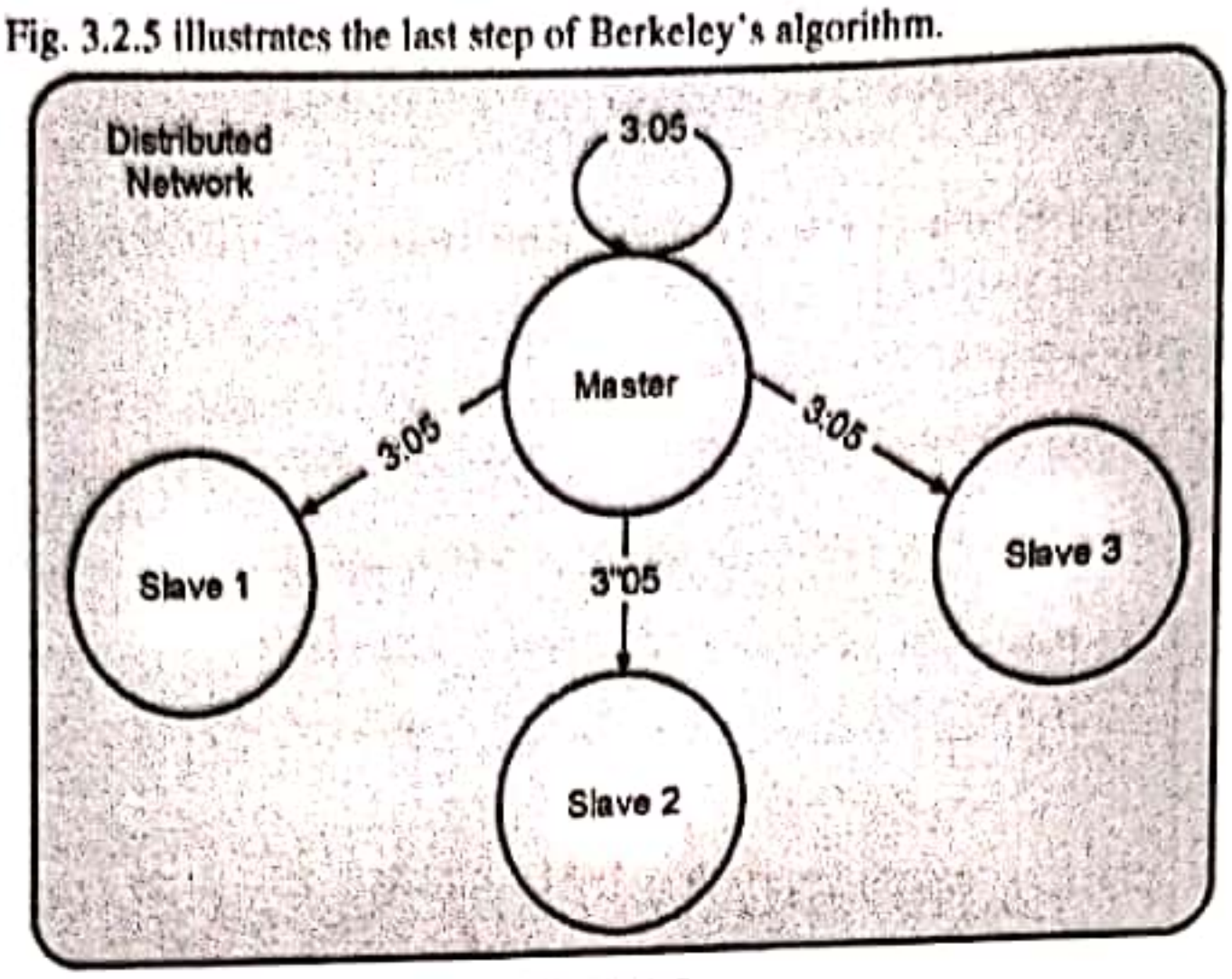


Fig. 3.2.5

- There are the following limitations:
  (i) Due to centralized system single point of failure may occur.
  (ii) A single time-server may not be capable of serving all time requests from scalability point of view. The readjustment value can either be +ve or –ve.

3.2.4(B) Distributed Algorithms

- There is no centralized or reference time-server.
- It performs clock synchronization based on the internal clock values of each node with the consideration of minimum clock skew value among clocks of different nodes in the system.
- Distributed clock synchronization algorithm overcomes issue of scalability and single point of failure as there is no common or global clock required.
- Processes make decisions based on local information and relevant information distributed across machines.

1. Global Averaging Algorithm

- Each node in the system broadcast its local clock time in the form of a special 'resync' message when its local time is $t_0 + IR$, where I is for interrupt time and R includes the number of nodes in the system, maximum drift rate, etc.
- After broadcasting, the clock process of the node waits for some time period t.
- During this period, it collects a 'resync' message from other nodes and records its receiving time based on its local clock time.
- At the end of the waiting period, it estimates the skew of its own clock with respect to all other nodes.
- To find the correct time, need to estimate the skew value as follows:
  (i) Take the average of the estimated skew and use it as a correction of the local clock. To limit the impact of faulty clock skews greater than threshold are set to zero before computing average of estimated skew.
  (ii) Each node limits the impact of faulty clock by discarding highest and lowest estimated skews and then calculates average of estimated skew.

- There are some limitations as follows:
  - (i) This method does not scale well.
  - (ii) Network should support broadcast facility.
  - (iii) Due to large number of messages, network traffic increases. Therefore, this algorithm may be useful for small networks only.

2. **Localized Averaging Algorithm**

- This algorithm overcomes the limitation of scalability of the global averaging algorithm.
- Nodes of the system are arranged logically in a specific pattern like a ring or grid.
- Periodically each node exchanges its local clock time with its neighbors and then sets its clock time to the average of its own clock time and the clock time of its neighbours.

3.2.4(C) Network Time Protocol (NTP)

- Network Time Protocol (NTP) is an internet protocol used to synchronize with computer clock time sources in a network. It belongs to and is one of the oldest parts of the TCP/IP suite.
- The term NTP applies to both the protocol and the client-server programs that run on computers.
- David Mills, professor at the University of Delaware, developed NTP in 1981. It is designed to be highly fault-tolerant and scalable, while supporting time synchronization.
- The following three steps are involved in the NTP time synchronization process:
  - (i) The NTP client initiates a time-request exchange with the NTP server.
  - (ii) The client is then able to calculate the link delay and its local offset and adjust its local clock to match the clock at the server's computer.
  - (iii) As a rule, six exchanges over a period of about five to 10 minutes are required to initially set the clock.
- Once synchronized, the client updates the clock about once every 10 minutes, usually requiring only a single message exchange, in addition to client-server synchronization. This transaction occurs via User Datagram Protocol (UDP) on port 123.
- NTP also supports broadcast synchronization of peer computer clocks. The distributed system may consist of several reference clocks.
- Each node of such network can exchange information either bidirectional or unidirectional.
- A client node can send request message to its directly connected time server node or nearly connected node so the propagation time from one node to another node forms hierarchical graph with reference clocks at the top. This hierarchical structure is known as strata synchronization subnet where each level is referred as strata. It is shown as below in Fig. 3.2.6.

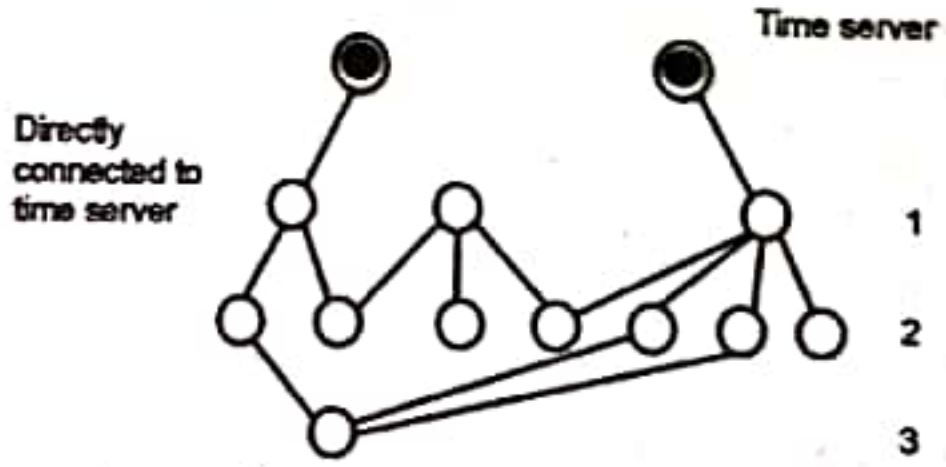


Fig. 3.2.6 : Strata Synchronization Subnet

- NTP servers synchronize with each other in one of the three modes:
  - (i) **Multicast Mode :** In this mode of synchronization, one or more servers periodically multicast their time to the other servers running in the network. Other servers set their time accordingly with the addition of small delay during transaction. This method is proposed for use in a high-speed Local Area Network (LAN).

(ii) **Procedural Call Mode :** It is similar to the process of Cristian's algorithm in which one server accepts reques from other computers and server replies with its current clock time. It gives more accurate time than multica mode. It can be used where multicast is not supported in hardware.

(iii) **Symmetric Mode :** In this mode of synchronization the pair of server exchanges messages. Servers supply tim information in LANs and higher levels of the synchronization subnet where the highest accuracy is to be achieved

- There are some limitations of NTP protocol as follows:

(i) NTP supports UNIX operating systems only.

(ii) For windows there are problems with time resolution, reference clock drives, authentication and name resolution

3.2.4(D) Simple Network Time Protocol (SNTP)

- This is a simplified way of clock synchronization method. SNTP is based on unicast mode of Network Time Protocol (NTP) and also operates in multicast and procedure call mode.
- It is recommended in network environment where server is root and client is leaf node.
- Client node send request message at time t_1 to server node and server send current time value in reply message at time t_3 as shown below in Fig. 3.2.7.
- Time offset $t = \dfrac{(t_2 - t_1) + (t_3 - t_4)}{2}$
- Current time = $t_4 + t$
- NTP protocol gives more accuracy in time than SNTP protocol. It is useful for simple applications where more accurate time is not necessary.
- Example: Suppose $t_1 = 1100$, $t_2 = 800$, $t_3 = 850$ and $t_4 = 1200$
- Time offset $t = \dfrac{(t_2 - t_1) + (t_3 - t_4)}{2}$

$$= \frac{(800 - 1100) + (850 - 1200)}{2} = \frac{(-300) + (-350)}{2}$$

$$= \frac{-650}{2} = -325$$

- Current time = $t_4 + t = 1200 - 325 = 875$

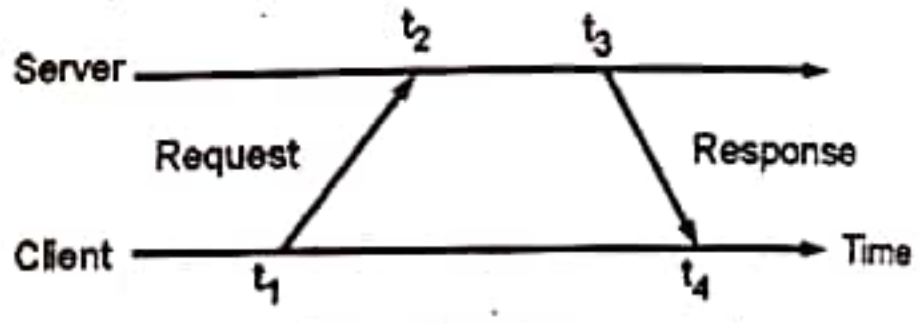


Fig. 3.2.7 : SNTP Synchronization

3.3 LOGICAL CLOCKS

UQ. What is a logical clock? Why are logical clocks required in distributed systems? How does Lamport synchronize logical clocks? Which events are said to be concurrent in Lamport's Timestamp ? **(MU - May 17)**

UQ. Describe any one method of Logical Clock Synchronization with the help of an example. **(MU - Dec. - 18, May 19)**

- Logical clocks refer to the implementation of a protocol on all machines in a distributed system so that the machines can maintain consistent sequencing of events over some virtual time span. In a distributed system, a logical clock is a technique for capturing temporal and causal relationships. Because distributed systems may lack a physically synchronous global clock, a logical clock permits global ordering on events from different processes in such systems.
- Example: If we go outside, we have a detailed plan of where we will go first, second, and so on. We don't start with second place and then move up to first. We always follow the previously planned procedure or arrangement. Similarly, we should do PC operations one by one in an ordered manner.

- Assume we have more than ten PCs in a distributed system, each doing its own work, but how can we get them to work together? This is where the LOGICAL CLOCK comes in.
- **Method 1**: To order events across process, try to sync clocks in one approach.
- This means that if one PC has a time 3:00 pm then every PC should have the same time which is quite not possible. Not every clock can sync at one time. Then we can't follow this method.
- **Method 2**: Another approach is to assign Timestamps to events.
- Suppose we assign the first place as 1, second place as 2, third place as 3 and so on. Then we always know that the first place will always come first and then so on. Similarly, if we give each PC their individual number than it will be organized in a way that 1st PC will complete its process first and then second and so on. But timestamps will only work as long as they obey causality.

- Causality is fully based on HAPPEN BEFORE RELATIONSHIP.
  1. Taking single PC only if 2 events A and B are occurring one by one then TS(A) < TS(B). If A has timestamp of 1, then B should have timestamp more than 1, then only happen before relationship occurs.
  2. Taking 2 PCs and event A in P1 (PC.1) and event B in P2 (PC.2) then also the condition will be TS(A) < TS(B). Taking example- suppose you are sending message to someone at 2:00:00 pm, and the other person is receiving it at 2:00:02 pm. Then, it's obvious that TS(sender) < TS(receiver).
- Properties Derived from Happen Before Relationship:
  1. **Transitive Relation :** If, TS(A) <TS(B) and TS(B) <TS(C), then TS(A) < TS(C)
  2. **Causally Ordered Relation :** a→b, this means that a is occurring before b and if there are any changes in a it will surely reflect on b.
  3. **Concurrent Event :** This means that not every process occurs one by one, some processes are made to happen simultaneously i.e., A ‖ B.

3.3.1 Lamport's Scalar Clock

- In a distributed system, clocks need not be synchronized absolutely.
- If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus it does not cause problem.
- It is not important that all processes agree on what the actual time is, but that they agree on the order in which events occur.
- Lamport clocks are a simple technique used for determining the order of events in a distributed system.
- Lamport clocks provide a partial ordering of events – specifically "happened-before" ordering.
- If there is no "happened-before" relationship, then the events are considered concurrent.
- Rules of Lamport's Logical Clocks:

(a) Happened Before Relation

- If a & b are events in the same process and a occurs before b, then a→b.
- If a & b belong to two different processes and a sends message to b, then a→b.
- If a→b and b→c, then a→c.

(b) Logical Clocks Concept

- Maintaining a common clock or set of perfectly synchronized clock is difficult.
- So, local timestamp is given for happened before relation.

- **Example**

1. Consider three processes 0, 1 & 2.
2. For process 0 the time ticked is 6, for 1 its 8 and for 2 its 10.
3. The clocks run at constant rate but different due to difference in the crystals.
4. At time 6, process 0 sends message A to process 1 and is received at time 16.
5. Time taken for transfer is 10.
6. Similarly process 1 sends message B to process 2 and is received at time 40.
7. Time taken for transfer is 16.
8. Now message C is sent from process 2 to process 1 it takes 16 ticks to reach and similarly message D takes 20 ticks to reach.
9. This value is certainly impossible and such a situation must be prevented.
10. Solution for this is Lamport happen before relation.
11. Since message C left at 64 so it must arrive at 65 or later and similarly, message D left at 69 so it must be arrived at 70 or later.
12. This is shown in Fig. 3.3.1.

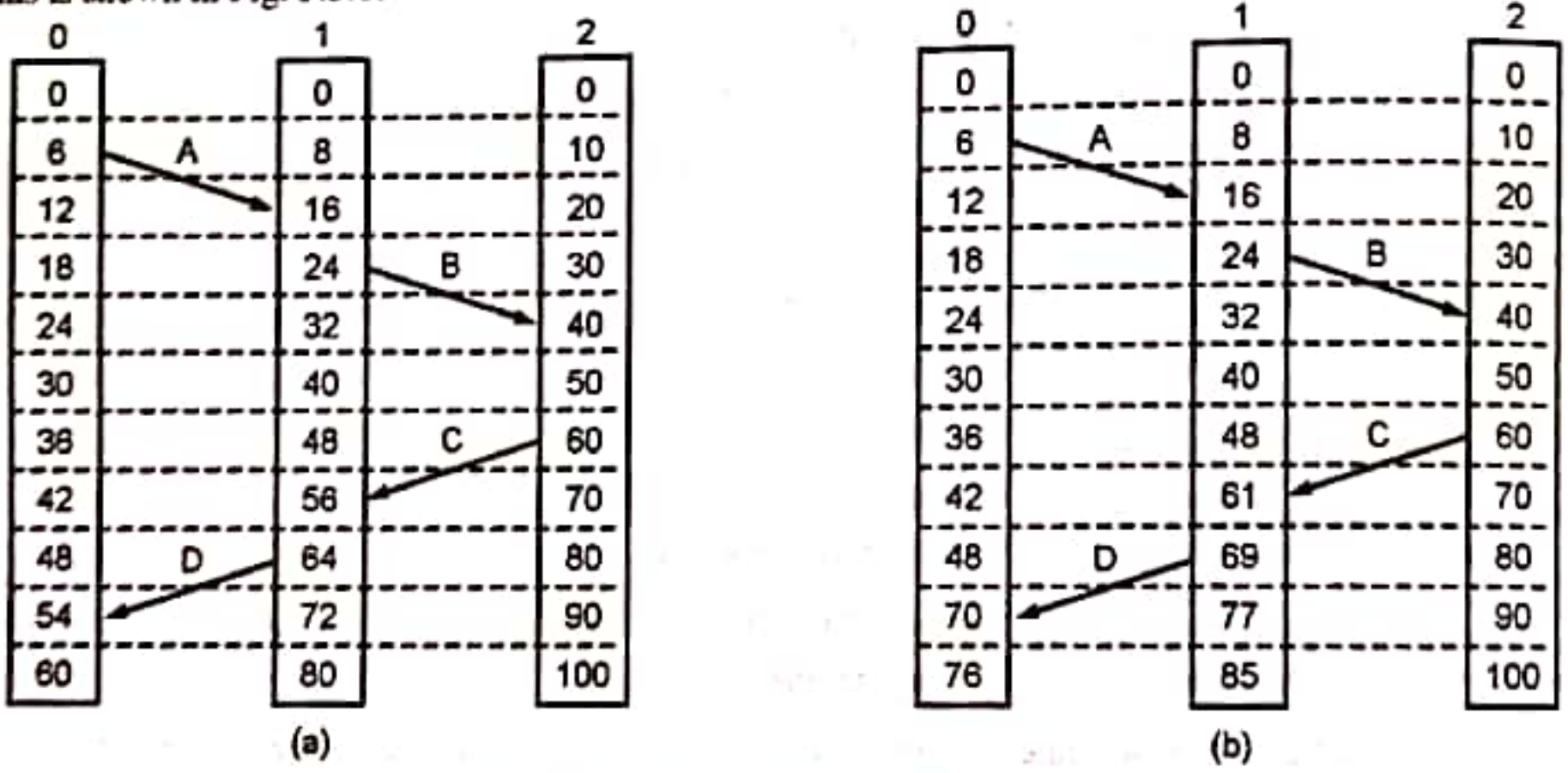


Fig. 3.3.1 : Event Ordering Example

3.3.2 Vector Timestamp Ordering

- **Vector Clock** is an algorithm that generates partial ordering of events and detects causality violations in a distributed system.
- These clocks expand on Scalar time to facilitate a causally consistent view of the distributed system, they detect whether a contributed event has caused another event in the distributed system.
- It essentially captures all the causal relationships.
- This algorithm helps us label every process with a vector(a list of integers) with an integer for each local clock of every process within the system.
- So, for N given processes, there will be vector/ array of size N.
- Working of Vector Clock Algorithm :

(i) Initially, all the clocks are set to zero.

(ii) Every time, an internal event occurs in a process, the value of the processes' logical clock in the vector is incremented by 1.

(iii) Also, every time a process sends a message, the value of the processes' logical clock in the vector is incremented by 1.

- Every time, a process receives a message, the value of the processes' logical clock in the vector is incremented by 1, and moreover, each element is updated by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).
- **Example** : Consider a process (P) with a vector size N for each process: the above set of rules mentioned are to be executed by the vector clock:

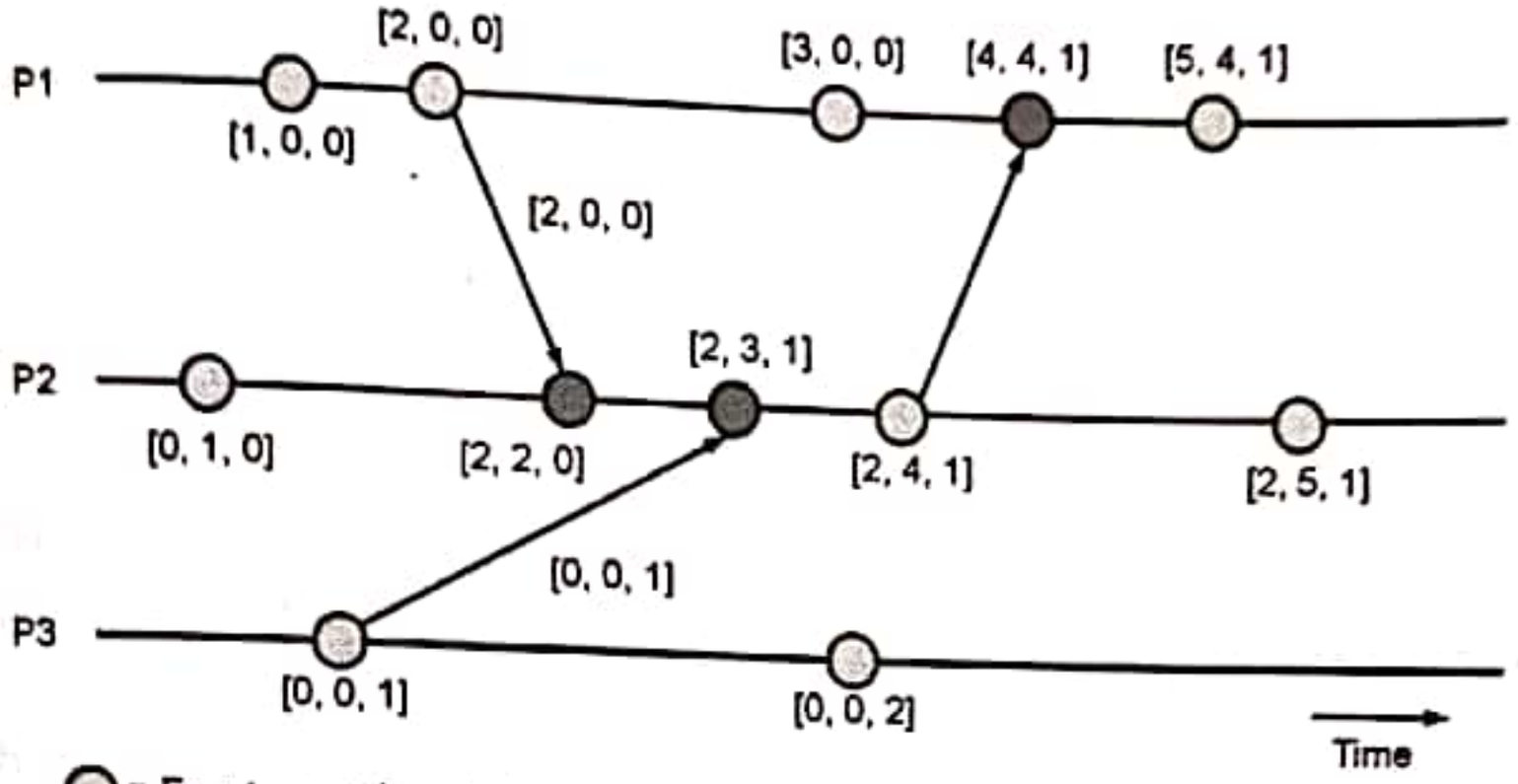


Fig. 3.3.2 : Vector Clock Mechanism

- The above example depicts the vector clocks mechanism in which the vector clocks are updated after execution of internal events, the arrows indicate how the values of vectors are sent in between the processes (P1, P2, P3).
- To sum up, Vector clocks algorithms are used in distributed systems to provide a causally consistent ordering of events but the entire Vector is sent to each process for every message sent, in order to keep the vector clocks in sync.

3.4 ELECTION ALGORITHMS

UQ. Write a note on Election Algorithm. (MU - Dec. 16, 19)

UQ. What is the requirement of Election algorithm in distributed systems? Describe any one Election algorithm in detail with example. (MU - May 17)

UQ. Describe any one Election algorithm in detail with example. (MU - Dec. 17)

UQ. Explain Bully Election Algorithm. (MU - May 18)

UQ. What is coordinator process? Explain algorithms used for selection of a coordinator. (MU - May 22)

- Distributed system is a collection of independent computers that do not share their memory. Each processor has its own memory, and they communicate via communication networks.
- Communication in networks is implemented in a process on one machine communicating with a process on another machine. Many algorithms used in distributed system require a coordinator that performs functions needed by other processes in the system.
- Election algorithms are designed to choose a coordinator.

- Election algorithms choose a process from group of processes to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor.
- Election algorithm basically determines where a new copy of coordinator should be restarted.
- Election algorithm assumes that every active process in the system has a unique priority number.
- The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then this number is sent to every active process in the distributed system. We have two election algorithms for two different configurations of distributed system.

3.4.1 Bully Algorithm

- This algorithm was proposed by Garcia-Molina. When the process notices that the coordinator is no longer responding to requests, it initiates an election.
- A process, P, holds an election as follows:

(i) P sends an ELECTION message to all processes with higher numbers.

(ii) If no one responds, P wins the election and becomes the coordinator.

(iii) If one of the higher-up's answers, it takes over. P's job is done.

(a) A process can get an ELECTION message at any time from one of its lower numbered colleagues.

(b) When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election unless it is already holding one.

(c) All processes give up except one that is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

(d) If a process that was previously down comes back up, it holds an election. If it happens to the highest numbered process currently running, it will win the election and take over the coordinator's job. Thus, the biggest guy in town always wins, hence the name "bully algorithm".

(e) Example:

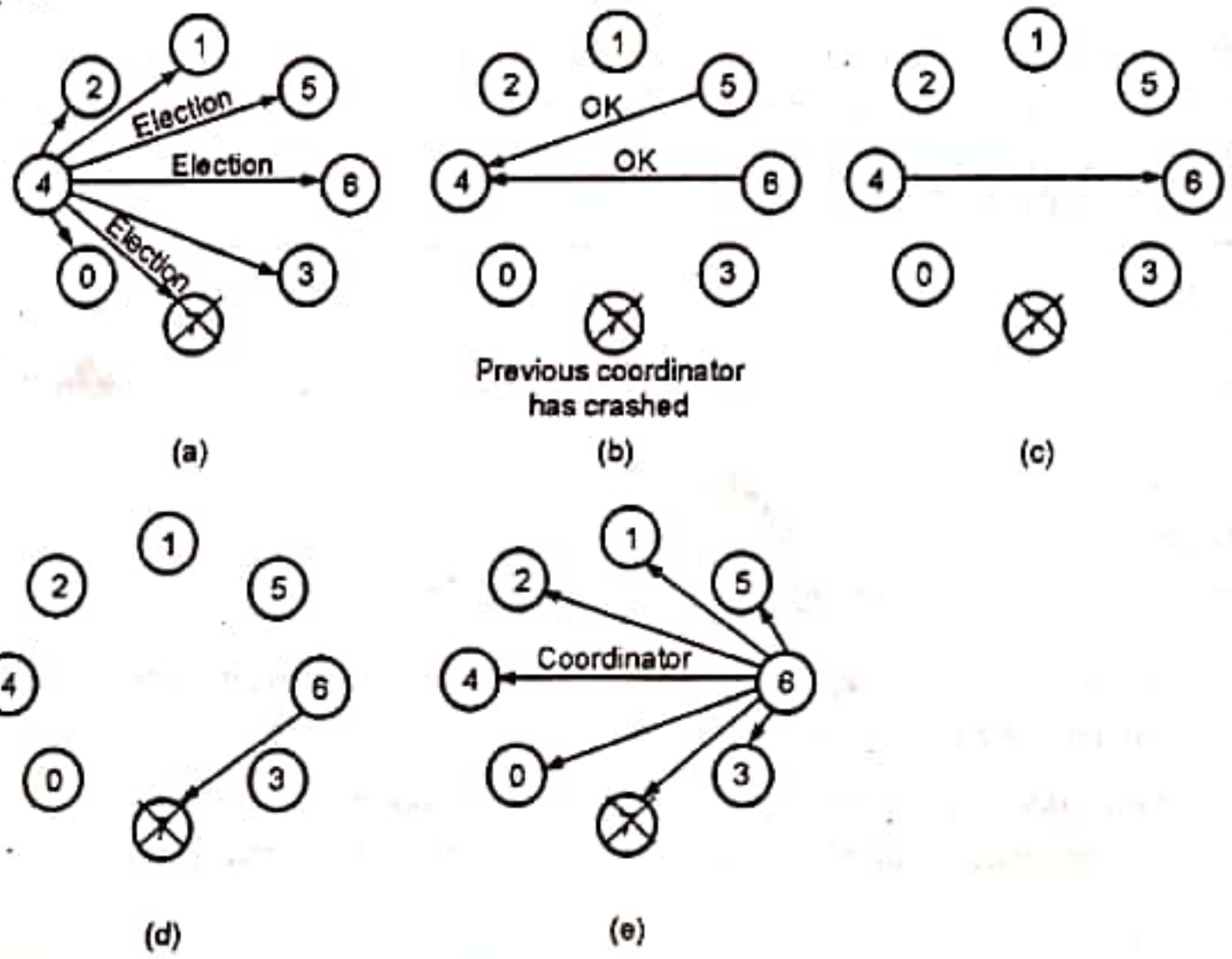


Fig. 3.4.1 : Election Algorithm

(i) In Fig. 3.4.1(a) a group of eight processes taken is numbered from 0 to 7. Assume that previously process 7 was the coordinator, but it has just crashed. Process 4 notices it first and sends ELECTION messages to all the processes higher than it that is 5, 6 and 7.

(ii) In Fig. 3.4.1(b) processes 5 and 6 both respond with OK. Upon getting the first of these responses, process4job is over. It knows that one of these will become the coordinator. It just sits back and waits for the winner.

(iii) In Fig. 3.4.1(c), both 5 and 6 hold elections by each sending messages to those processes higher than itself.

(iv) In Fig. 3.4.1(d), process 6 tells 5 that it will take over with an OK message. At this point 6knows that 7 is dead and that 6 it is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announce this by sending a COORDINATOR message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of is handled and the work can continue.

(v) If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

3.4.2 Ring Algorithm

This algorithm uses a ring for its election but does not use any token. In this algorithm it is assumed that the processes are physically or logically ordered so each processor knows its successor :

(i) When any process notices that a coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down the sender skips over the successor and goes to the next member along the ring until a process is located.

(ii) At each step the sender adds its own process number to the list in the message making itself a candidate to elected as coordinator.

(iii) The message gets back to the process that started it and recognizes this event as the message consists of its own process number.

(iv) At that point the message type is changed to COORDINATOR and circulated once again to inform everyone who the coordinator is and who are the new members.

(v) The coordinator is selected with the process having highest number.

(vi) When this message is circulated once it is removed, and normal work is preceded.

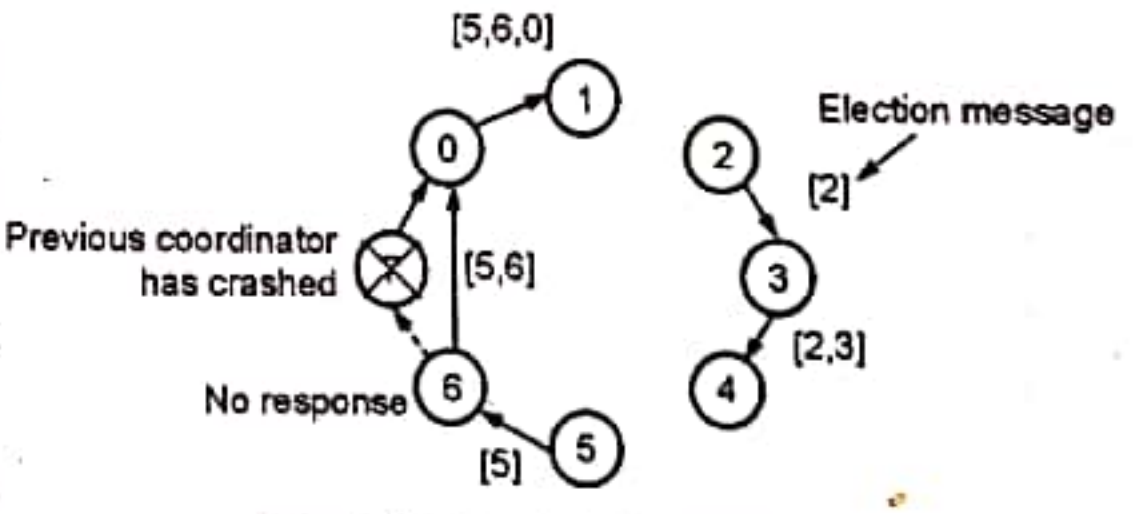


Fig. 3.4.2 : Ring Algorithm

3.5 MUTUAL EXCLUSION

UQ. Explain the distributed algorithms for mutual exclusion. What are the advantages and disadvantages of it over centralized algorithms? (MU - May 16)

UQ. Explain Lamport's Mutual Exclusion Algorithm. (MU - Dec. 17)

- Mutual exclusion is a concurrency control property which is introduced to prevent race conditions.

- It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e., only one process is allowed to execute the critical section at any given instance of time.
- **Mutual Exclusion in Single Computer System :** In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.
- **Mutual Exclusion in Distributed Systems :** In Distributed systems, we neither have shared memory nor a common physical clock and there for we cannot solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used. A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

3.5.1 Requirements of Mutual Exclusion Algorithm

Following are the requirements of mutual exclusion algorithms :

(i) **No Deadlock :** Two or more sites should not endlessly wait for any message that will never arrive.

(ii) **No Starvation :** Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other sites are repeatedly executing critical section.

(iii) **Fairness :** Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e., critical section execution requests should be executed in the order of their arrival in the system.

(iv) **Fault Tolerance :** In case of failure, it should be able to recognize it by itself to continue functioning without any disruption.

3.5.2 Classification of Mutual Exclusion Algorithms

Mutual exclusion algorithms can be classified as :

1. Centralized Algorithm
2. Distributed Algorithm
3. Token Ring Algorithm

(1) Centralized Algorithm

- One of the simplest approaches to achieve mutual exclusion is to use centralized algorithm.
- The centralized algorithm shown in Fig. 3.5.1 below works as follows.
- In centralized algorithm one process is elected as the coordinator which may be the machine with the highest network address.
- When a process wishes to enter a critical region, it sends a request message to the coordinator, indicating which critical region it wishes to enter and requesting permission to do so. If no other process is currently operating in that critical region, the coordinator responds by granting permission, as seen in Fig 3.5.1(a). The requesting procedure enters the critical region when the reply arrives.
- Suppose another process 2 shown in Fig 3.5.1(b), asks for permission to enter the same critical region. Now the coordinator knows that a different process is already in the critical region, so it cannot grant permission. The coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply, or it could send a reply saying 'permission denied.'
- When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access as shown in Fig. 3.5.1(c).

- The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked it unblocks and enters the critical region.
- If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. When it sees the grant, it can enter the critical region.

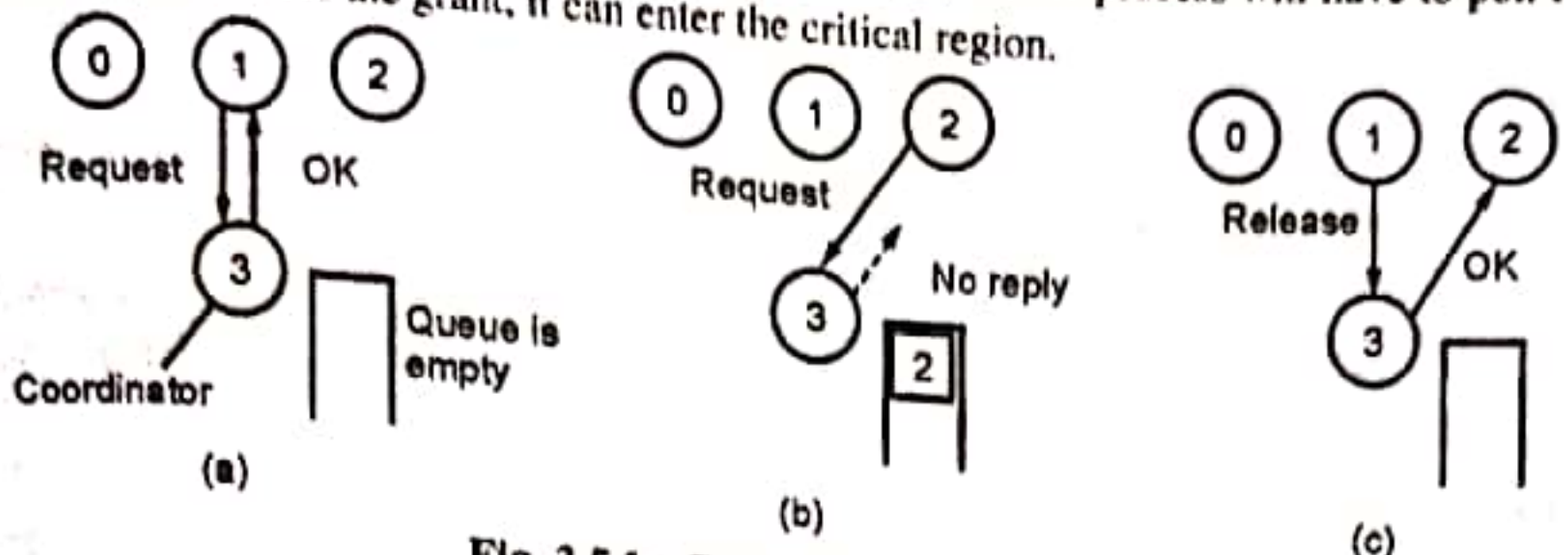


Fig. 3.5.1 : Centralized Algorithm

Advantages

(i) Algorithm guarantees mutual exclusion by letting one process at a time into each critical region.

(ii) It is also fair as requests are granted in the order in which they are received.

(iii) No process ever waits forever so no starvation.

(iv) Easy to implement so it requires only three messages per use of a critical region (request, grant, release).

(v) Used for more general resource allocation rather than just managing critical regions.

Disadvantages

(i) The coordinator is a single point of failure, the entire system may go down if it crashes.

(ii) If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since no message comes back.

(iii) In a large system a single coordinator can become a performance bottleneck.

(2) Distributed Algorithm (Lamport's Mutual Exclusion)

- In this scheme, there is no coordinator. Every process asks to other process for permission to enter critical section.
- The algorithm works as follows :

(i) When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time.

(ii) It then sends the message to all other processes, conceptually including it.

(iii) The sending of messages is assumed to be reliable that is, every message is acknowledged.

(iv) Reliable group communication if available can be used instead of individual messages.

(v) When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message.

(vi) Three cases have to be distinguished:

Case 1 : If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.

Case 2 : If the receiver is already in the critical region, it does not reply. Instead, it queues the request.

Case 3 : If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If

the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

(vii) After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission.

(viii) As soon as all the permissions are in, it may enter the critical region.

(ix) When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

(x) Let us try to understand how the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 3.5.2(a).

(a) Two processes want to enter the same critical region at the same moment.

(b) Process 0 has the lowest timestamp, so it wins.

(c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

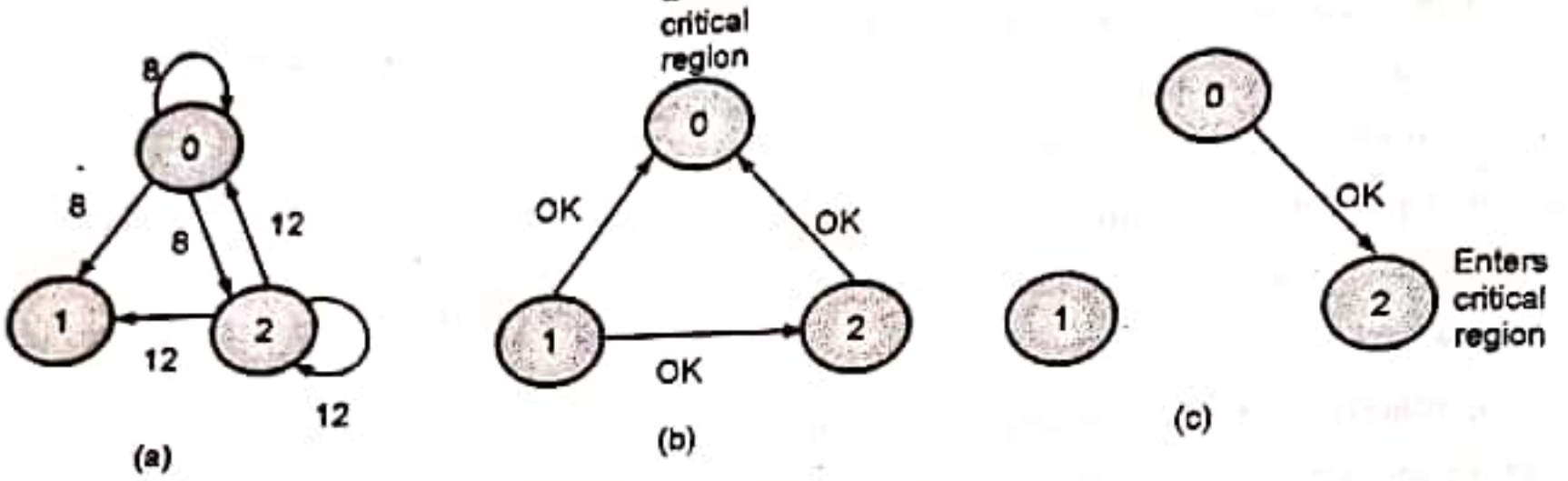


Fig. 3.5.2 : Distributed Algorithm

- An example will explain the algorithm more clearly.

(i) Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12.

(ii) Process 1 is not interested in entering the critical region, so it sends OK to both senders.

(iii) Processes 0 and 2 both see the conflict and compare timestamps.

(iv) Process 2 sees that it has lost, so it grants permission to 0 by sending OK.

(v) Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 3.5.2(b).

(vi) When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region, as shown in Fig. 3.5.2(c).

(vii) The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

☞ ***Advantages***

(i) Mutual exclusion is guaranteed without deadlock or starvation. (ii) No single point of failure

☞ ***Disadvantages***

(i) A group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing.

(ii) This algorithm is slower, more complicated, more expensive, and less robust.

(3) Token Ring

- In this algorithm it is assumed that all the processes in the system are organized in a logical ring. The Fig. 3.5.3 below describes the structure.

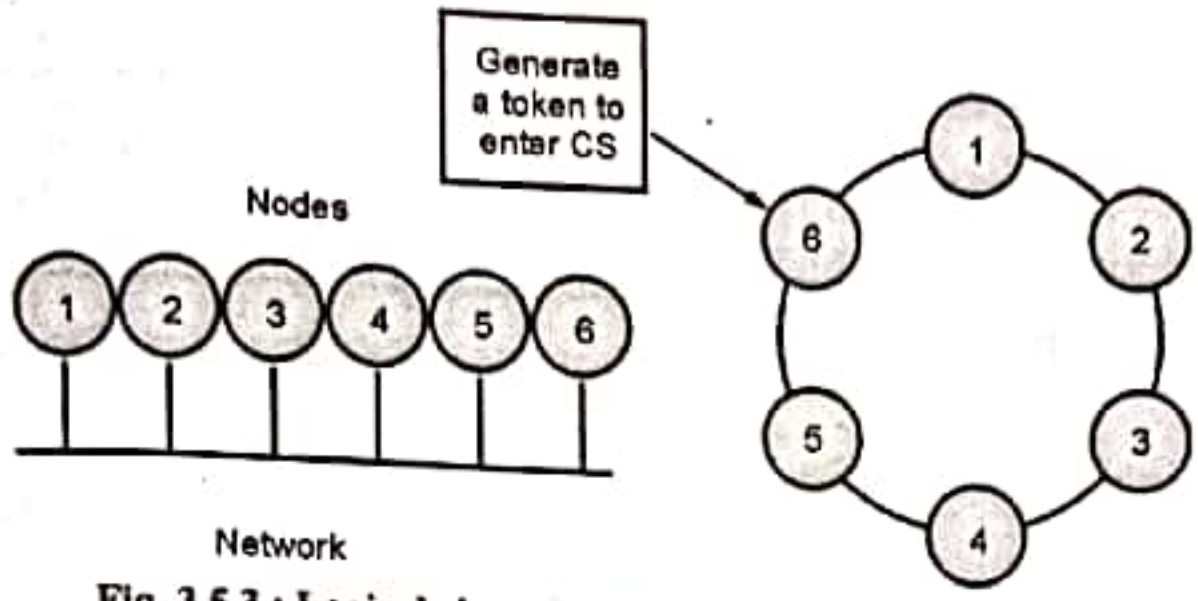


Fig. 3.5.3 : Logical ring of unordered node of the network

- The ring positions may be allocated in numerical order of network addresses and is unidirectional in the sense that all messages are passed only in clockwise or anti-clockwise direction.
- When a process sends a request message to current coordinator and does not receive a reply within a fixed timeout, it assumes the coordinator has crashed. It then initializes the ring and process P_i is given a token.
- The token circulates around the ring. It is passed from process k to k + 1 in point-to-point messages. When a process acquires the token from its neighbour it checks to see if it is attempting to enter a critical region. If so, the process enters the region does all the execution and leaves the region. After it has exited it passes the token along the ring. It is not permitted to enter a second critical region using the same token.
- If a process is handed the token by its neighbour and is not interested in entering a critical region it just passes along. When no processes want to enter any critical regions, the token just circulates at high speed around the ring.
- Only one process has the token at any instant so only one process can actually be in a critical region. Since the token circulates among the process in a well-defined order, starvation cannot occur.
- Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

☞ **Advantages**

(i) Only one process has the token at any instant, so only one process can be in a critical section, thereby avoiding the problem of race condition.

(ii) Since thetoken circulates among processes in awell-defined order, starvation cannot occur.

☞ **Disadvantages**

(i) If the token is lost, it must be regenerated. But the detection of lost token is difficult. If the token is not received for a long time, it might not be lost but is in use.

(ii) If a process fails, the algorithm also fails, but recovery is easier than in the other cases. If a process receiving the token is required to confirm receipt, a dead process will be recognized when its neighbour tries and fails to deliver it the token. The dead process can then be removed from the group, and the token holder can pass the token on to the next member in line.

3.5.3 Comparison of Mutual Exclusion Algorithms

Table 3.5.1 : Comparison of Mutual Exclusion Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n-1)	2(n-1)	Crash of any process
Token Ring	1 to ∞	0 to (n-1)	Lost token, process crash

3.6 NON-TOKEN BASED ALGORITHMS

UQ. Explain Ricart-Agrawala algorithm for mutual exclusion. (MU - May 18)
UQ. Discuss Ricart-Agrawal's Algorithm and justify how this algorithm optimized the message overhead in achieving mutual exclusion. (MU - Dec. 18)
UQ. Justify how this algorithm optimized the message overhead in achieving mutual exclusion. (MU - May 19)
UQ. Write a note on: Lamport's Algorithm. (MU - Dec. 19)

- A specific node communicates with a group of nodes to determine who should enter the critical section next in a non-token-based mutual exclusion method.
- In non-token-based mutual exclusion algorithms, requests to access the critical section are ranked based on timestamps.
- Simultaneous requests are likewise managed by using the request's timestamp. The Lamport's logical clock is employed, and requests for critical sections with lower timestamp values take precedence over those with higher timestamp values.
- Different non-token based mutual exclusion algorithms are discussed in this section.

3.6.1 Lamport's Algorithm

- Lamport's Distributed Mutual Exclusion Algorithm is a permission-based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems.
- In permission-based timestamp is used to order critical section requests and to resolve any conflict between requests.
- In Lamport's Algorithm critical section requests are executed in the increasing order of timestamps i.e., a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.
- In this algorithm:
  (i) Three types of messages (**REQUEST, REPLY** and **RELEASE**) are used, and communication channels are assumed to follow FIFO order.
  (ii) A site sends a **REQUEST** message to all other site to get their permission to enter critical section.
  (iii) A site sends a **REPLY** message to requesting site to give its permission to enter the critical section.
  (iv) A site sends a **RELEASE** message to all other site upon exiting the critical section.
  (v) Every site S_i, keeps a queue to store critical section requests ordered by their timestamps. **request_queue**$_i$ denotes the queue of site S_i.
  (vi) A timestamp is given to each critical section request using Lamport's logical clock.
  (vii) Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

Algorithm

To enter Critical section

- (a) When a site S_i wants to enter the critical section, it sends a request message **Request(TS_i, i)** to all other sites and places the request on **request_queue$_i$**. Here, TS_i denotes the timestamp of Site S_i
- (b) When a site S_j receives the request message **REQUEST(TS_i, i)** from site S_i, it returns a timestamped REPLY message to site S_i and places the request of site S_i on **request_queue$_j$**

To execute the critical section

- (a) A site S_i can enter the critical section if it has received the message with timestamp larger than (TS_i, i) from all other sites and its own request is at the top of **request_queue$_i$**

To release the critical section

- (a) When a site S_i exits the critical section, it removes its own request from the top of its request queue and sends a timestamped **RELEASE** message to all other sites
- (b) When a site S_j receives the timestamped **RELEASE** message from site S_i, it removes the request of S_i from its request queue

Message Complexity

Lamport's Algorithm requires invocation of 3(N – 1) messages per critical section execution. These 3(N – 1) messages involves

- (N – 1) request messages
- (N – 1) reply messages
- (N – 1) release messages

Drawbacks of Lamport's Algorithm

- **Unreliable approach:** failure of any one of the processes will halt the progress of entire system.
- **High message complexity:** Algorithm requires 3(N-1) messages per critical section invocation.

Performance

- Synchronization delay is equal to maximum message transmission time
- It requires 3(N – 1) messages per CS execution.
- Algorithm can be optimized to 2(N – 1) messages by omitting the **REPLY** message in some situations.

3.6.2 Ricart–Agrawala Algorithm

- Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala.
- This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm.
- Like Lamport's Algorithm, it also follows permission-based approach to ensure mutual exclusion.
- In this algorithm:
  - (i) Two type of messages (**REQUEST** and **REPLY**) are used and communication channels are assumed to follow FIFO order.
  - (ii) A site send a **REQUEST** message to all other site to get their permission to enter critical section.
  - (iii) A site send a **REPLY** message to other site to give its permission to enter the critical section.
  - (iv) A timestamp is given to each critical section request using Lamport's logical clock.
  - (v) Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

Algorithm

To enter Critical section

(a) When a site S_i wants to enter the critical section, it sends a timestamped **REQUEST** message to all other sites.

(b) When a site S_j receives a **REQUEST** message from site S_i, it sends a **REPLY** message to site S_i if and only if

- Site S_j is neither requesting nor currently executing the critical section.
- In case Site S_i is requesting, the timestamp of Site S_i's request is smaller than its own request.

Otherwise, the request is deferred by site S_j.

- **To execute the critical section**

  Site S_i enters the critical section if it has received the **REPLY** message from all other sites.

- **To release the critical section**

  Upon exiting site S_i sends **REPLY** message to all the deferred requests.

Message Complexity

Ricart–Agrawala algorithm requires invocation of 2(N – 1) messages per critical section execution. These 2(N – 1) messages involves

- (N – 1) request messages
- (N – 1) reply messages

Drawbacks of Ricart–Agrawala algorithm:

Unreliable approach : Failure of any one of node in the system can halt the progress of the system. In this situation, the process will starve forever. The problem of failure of node can be solved by detecting failure after some timeout.

Performance

- Synchronization delay is equal to maximum message transmission time
- It requires 2(N – 1) messages per Critical section execution

3.6.3 Maekawa's Algorithm

- Maekawa's Algorithm is quorum-based approach to ensure mutual exclusion in distributed systems.
- As we know, in permission-based algorithms like Lamport's Algorithm, Ricart-Agrawala Algorithm etc. a site reques permission from every other site but in quorum-based approach,
- A site does not request permission from every other site but from a subset of sites which is called **quorum.**
- In this algorithm:
  (i) Three types of messages (**REQUEST, REPLY** and **RELEASE**) are used.
  (ii) A site sends a **REQUEST** message to all other site in its request set or quorum to get their permission to enter critical section.
  (iii) A site sends a **REPLY** message to requesting site to give its permission to enter the critical section.
  (iv) A site sends a **RELEASE** message to all other site in its request set or quorum upon exiting the critical section.

The construction of request set or Quorum

A request set or Quorum in Maekawa's algorithm must satisfy the following properties:

(1) $\forall i\ \forall j : i \neq j,\ 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset$ i.e., there is at least one common site between the request sets of any two sites.

(2) $\forall i : 1 \leq i \leq N :: S_i \in R_i$

(3) $\forall i : 1 \leq i \leq N :: |R_i| = K$

(4) Any site S_i is contained in exactly K sets.

(5) $N = K(K - 1) + 1$ and $|R_i| = \sqrt{N}$

Algorithm

- **To enter Critical section**

(a) When a site S_i wants to enter the critical section, it sends a request message **REQUEST(i)** to all other sites in the request set R_i.

(b) When a site S_j receives the request message **REQUEST(i)** from site S_i, it returns a **REPLY** message to site S_i if it has not sent a **REPLY** message to the site from the time it received the last **RELEASE** message. Otherwise, it queues up the request.

- **To execute the critical section**

(a) A site S_i can enter the critical section if it has received the **REPLY** message from all the site in request set R_i

- **To release the critical section**

(a) When a site S_i exits the critical section, it sends **RELEASE(i)** message to all other sites in request set R_i

(b) When a site S_j receives the **RELEASE(i)** message from site S_i, it sends **REPLY** message to the next site waiting in the queue and deletes that entry from the queue

(c) In case queue is empty, site S_j update its status to show that it has not sent any **REPLY** message since the receipt of the last **RELEASE** message

Message Complexity

Maekawa's Algorithm requires invocation of $3\sqrt{N}$ messages per critical section execution as the size of a request set is $\sqrt{N}$. These $3\sqrt{N}$ messages involves.

- $\sqrt{N}$ request messages
- $\sqrt{N}$ reply messages
- $\sqrt{N}$ release messages

Drawbacks of Maekawa's Algorithm

This algorithm is deadlock prone because a site is exclusively locked by other sites and requests are not prioritized by their timestamp.

Performance

- Synchronization delay is equal to twice the message propagation delay time
- It requires $3\sqrt{n}$ messages per critical section execution.

3.7 TOKEN BASED ALGORITHMS

UQ Write a Suzuki-Kasami's broadcast algorithm. Explain with suitable example. (MU - May 16)

UQ Write short note on: Raymond Tree Based Algorithm. (MU - Dec. 16)

UQ Discuss Raymond's Tree based algorithm of token based distributed mutual exclusion. (MU - May 17, Dec. 19)

UQ Differentiate between Token-based algorithm and Non-Token based algorithm. Explain in detail Raymond's Tree Based Algorithm. (MU - May 22)

- Token-based algorithms make use of tokens, and this unique token is shared by all processes.

- The process with the token can enter the critical section.
- There are various token-based algorithms depending on how the process searches for the token.
- Instead of timestamps, this algorithm uses sequence numbers. Each token request has a sequence number, and each process's sequence number progresses independently.
- When a process request for the token is made, the token's sequence number is incremented. This aids in distinguishing between old and new token requests. Because only the process that acquires the token enters the critical area, the algorithm ensures mutual exclusion.

3.7.1 Suzuki-Kasami's Broadcast Algorithm

- Suzuki–Kasami algorithm is a token-based algorithm for achieving mutual exclusion in distributed systems.
- This is modification of Ricart–Agrawala algorithm, a permission based (Non-token based) algorithm which uses REQUEST and REPLY messages to ensure mutual exclusion.
- In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token. Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.
- Each request for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.

Data structure and Notations

- An array of integers RN[1...N]

  A site S_i keeps $RN_i[1...N]$, where $RN_i[j]$ is the largest sequence number received so far through REQUEST message from site S_j.
- An array of integer LN[1...N]

  This array is used by the token. LN[j] is the sequence number of the request that is recently executed by site S_j.
- A queue Q

  This data structure is used by the token to keep record of ID of sites waiting for the token.

Algorithm

- ***To enter Critical section***

  (a) When a site S_i wants to enter the critical section and it does not have the token then it increments its sequence number $RN_i[i]$ and sends a request message **REQUEST(i, sn)** *to all other sites in order to request the token. Here sn is update sequence number of $RN_i[i]$*

  (b) When a site S_j receives the request message **REQUEST(i, sn)** *from site S_i, it sets $RN_j[i]$ to maximum of $RN_j[i]$ and sn i.e $RN_j[i] = \max(RN_j[i], sn)$.*

  (c) After updating $RN_j[i]$, Site S_j sends the token to site S_i if it has token and $RN_j[i] = LN[i] + 1$

- ***To execute the critical section***

  (a) Site S_i executes the critical section if it has acquired the token.

- ***To release the critical section***

  After finishing the execution, Site S_i exits the critical section and does following:

  (a) sets $LN[i] = RN_i[i]$ to indicate that its critical section request $RN_i[i]$ has been executed

  (b) For every site S_j, whose ID is not present in the token queue Q, it appends its ID to Q if $RN_i[j] = LN[j] + 1$ to indicate that site S_j has an outstanding request.

(c) After above updation, if the Queue **Q** is non-empty, it pops a site ID from the **Q** and sends the token to site indicated by popped ID.

(d) If the queue **Q** is empty, it keeps the token

Message Complexity

The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution. This N messages involves :

- (N – 1) request messages
- 1 reply message

Drawbacks of Suzuki–Kasami Algorithm

Non-symmetric Algorithm: A site retains the token even if it does not have requested for critical section. According to definition of symmetric algorithm "No site possesses the right to access its critical section when it has not been requested."

Performance

- Synchronization delay is 0 and no message is needed if the site holds the idle token at the time of its request.
- In case site does not holds the idle token, the maximum synchronization delay is equal to maximum message transmission time and a maximum of N message is required per critical section invocation.

Example

Consider 5 sites as shown in Fig. 3.7.1 below.

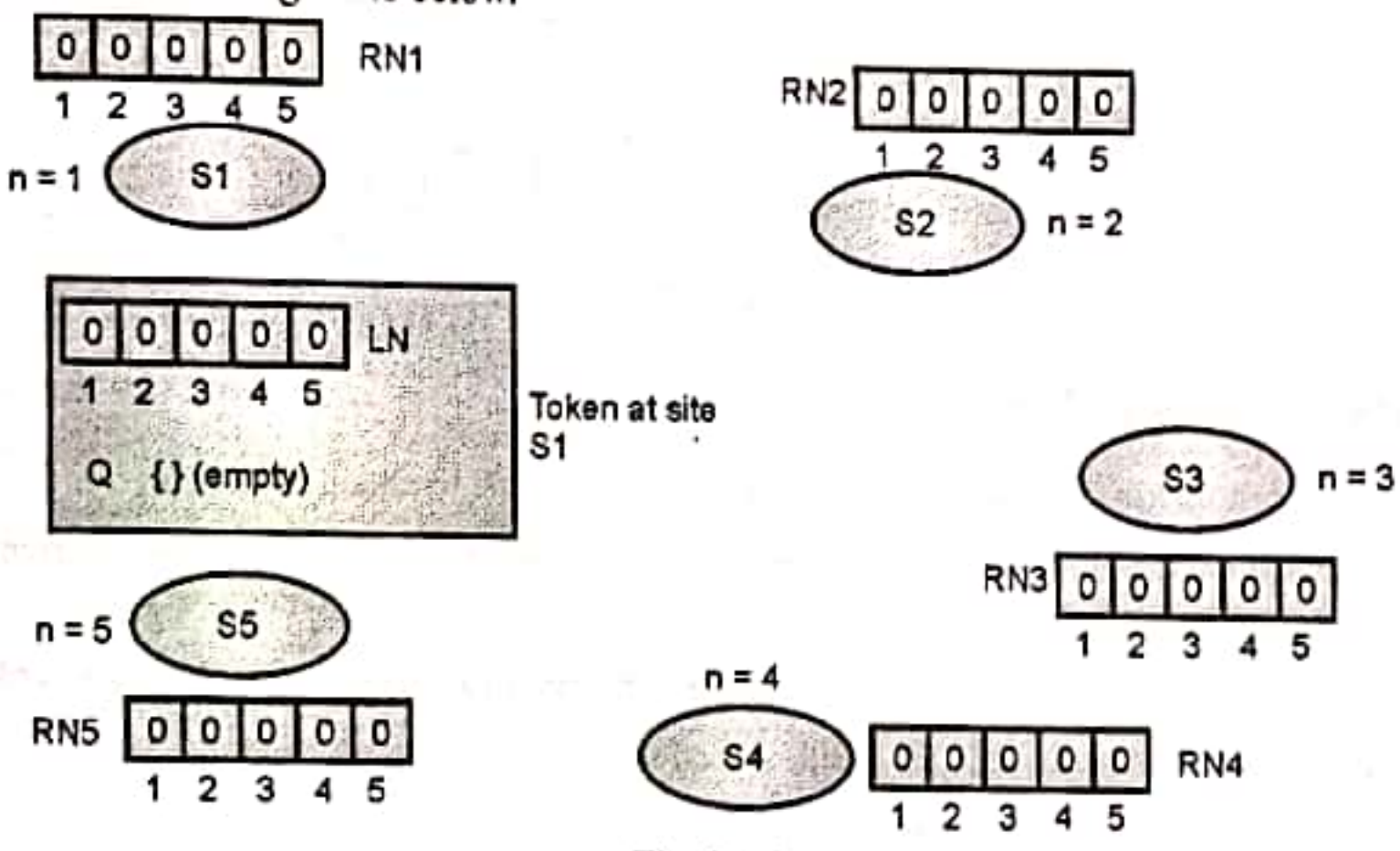


Fig. 3.7.1

Suppose Site S1 wants to enter critical section.

S1 won't execute the request part of the algorithm.

S1 enters the critical section as it has token. When S1 releases the critical section, it performs following tasks:

(i) LN[1] = RN[1] = 0 i.e., same as before

(ii) If during execution of critical section by S1, any request from other site (say S2)have come, then site's ID will now be entered into the queue if RN[2] = LN[2] + 1.

So, if site holding (not received from other) the token enters the critical section, then overall state of the system won't change, it remains same.

Now, suppose Site S2 wants to enter critical section.

S2 does not have a token, so it send REQUEST(2,1) message to every other site.

Sites S1, S3, S4, and S5 updates their RN as follows:

0	1	0	0	0
1	2	3	4	5

It may be possible that S2's request came at S1 when S1 is under critical section execution. So, at this point S1 will not pass the token to S2. It will finish its critical section and then add S2 into queue because at that point $RN_1[2] = 1$ which is LN[2]+1.

Now, as the queue is not empty, so S1 will pass token to S2 by removing S2 entry from token queue.

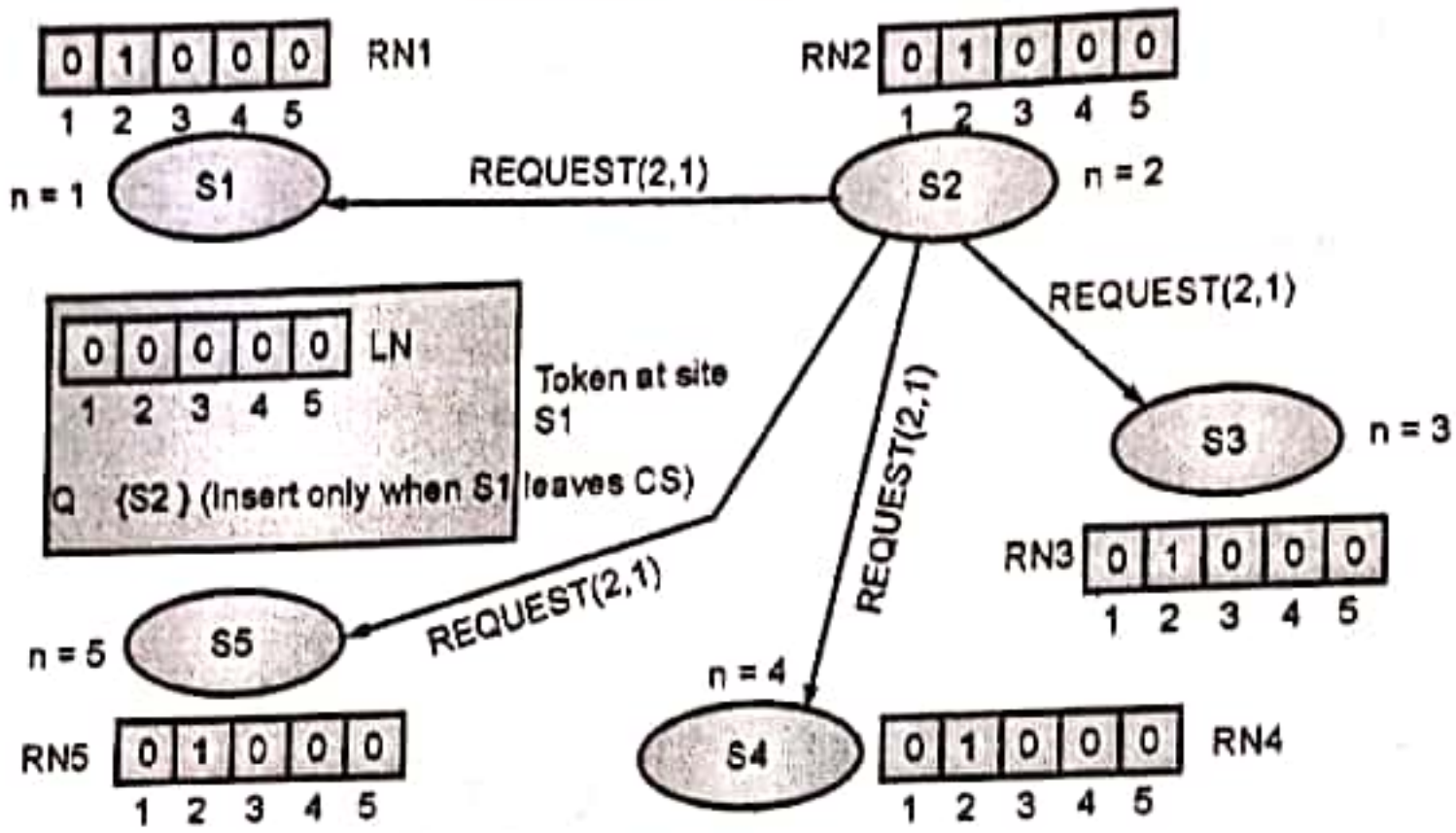


Fig. 3.7.2

3.7.2 Singhal's Heuristic Algorithm

- Suzuki-Kasami's algorithm has the limitation of having N messages just to obtain the token.
- Singhal's heuristic method tries to reduce this number by having knowledge of each process's current state and sending the token request accordingly.
- The process could be in any of the following states (shown in hierarchical order; state 1 is higher in hierarchy):
  1. E = Possess the token and is executing the critical section
  2. H = Holding the token and not executing
  3. R = Requesting the token
  4. N = Neutral, none of the above
- The algorithm selects the processes which have more probability of having the token and the requesting process then sends the request to only the selected ones rather than broadcasting thereby reducing the number of messages. It is called a heuristic algorithm because processes are heuristically selected for sending token request messages according to their current states.
- One of the design requirements of this algorithm is that a requesting process sends a REQUEST to a process that either holds the token or is going to obtain the token. Otherwise, there is a potential for deadlock or starvation.

Data Structures

Let there be N processes.

A process P_i has following two arrays:

1. Array holding the state of the process $State_i[1...N]$
2. Array $R_i[1...N]$ maintaining the highest number of requests received from each process.

The token has the following two arrays:

1. T_State[1...N], maintaining the status of the processes.
2. T_R[1...N], number of critical regions executed by each process.

Since the heuristic selects every process that is requesting the critical region according to the local information, every process is free to initialize its local array, $State_1[]$, according to its own will.

Select k from 1...N

Array Initialization

for each process P_i for i = 1...N do

$State_i[j] = N$; for j = N...k /*All processes >=k are set to neutral state = N */

$State_i[j] = R$; for j = k-1 ...1 /* All processes < k are assigned the state R */

Requesting Process

$R_i[j] = 0$; for j = 1...N do /*Initially there is no request received by any process*/

/* Let P_1 hold the token initially*/

$State_1[1] = H$ /*P1 has the state hold and it has the token*/

T_State[j] = N; for j = 1...N /*The token is not being used by any; the array is in state N*/

T_R[j] = 0; for j = 1...N /*There is no process that has executed the critical section*/

For example, if k is chosen to be 2, then the process having IDs 1 and above ($i \geq k$) are in neutral state, process P_1 ($i<k$) would be the only process in R state. Thus, if process P_2 now wishes to enter the critical section, the REQUEST message needs to be sent only to 1 as there is no need to send REQUEST for the permission grant to others as they never requested to enter the critical section in the first place.

Algorithm

Following is the heuristic algorithm, which is quite simple to understand.

- **To enter critical section**

  Set $State_i[j] = R$; /*requesting*/

  $n = R_i[j] = R_i[j]+1$;

  REQUEST(i,n) message to all other processes P_j for which $State_i[j] = R$

  /*This reduces the number of request messages sent in comparison to Suzuki-Kasami algorithm*/

- **P_j on receiving the request message from P_i; REQUEST(I,n)**

  (i) P_j sets $R_j[i] = \max(R_j[i],n)$ /* to take care of outdated requests*/

  (ii) If $State_i[j] = n$, then set $State_j[j] = R$ /*update the state of Pi in Pj to the requesting state*/

  (iii) If $State_j[j] = R$, if $State_j[j] != R$, set $State_j[j] = R$, send REQUEST(j, $R_j[j]$) to P_i/*If P_j is requesting, let P_i know about this request*/

  (iv) If $State_j[j] = E$, then set $State_j[j] = R$

  (v) If $State_j[j] = H$, then set $State_j[j] = R$, T_State[i] = R, T_R[i] = n, $State_j[j] = N$ and send the token to P_i.

- **On the execution of the critical section: Pi**
  - (i) Once the token is received, set $State_i[i] = E$;
  - (ii) Process P_i executes the critical section after it has received the token.
- **On the release of the critical section: Pi**
  - (i) Set $State_j[j] = N$, T_State[i] = N
  - (ii) for all j = 1...N do

    if $State_j[j] > T\_State[j]$ /*according to hierarchical order*/

    then

    $T\_State[j] = State_i[j]$;

    $T\_R[j] = Ri[j]$ /*update token information from local information*/
  - (iii) If (for all j; if $State_i[j] = N$), then set $State_i[i] = H$

    Else, if $State_i[j] = R$, send the token to j

Performance Parameters

1. The synchronization delay in the algorithm is T.
2. The number of REQUEST is (N-1)/2 and REPLY is (N-1)/2. So, the message complexity is (N-1) in low load condition.
3. In heavy load condition, message complexity is 3(N-1)/2.

3.7.3 Raymond's Tree-based Algorithm

- Raymond's tree based algorithm is lock based algorithm for mutual exclusion in a distributed system in which a site is allowed to enter the critical section if it has the token.
- In this algorithm, all sites are arranged as a directed tree such that the edges of the tree are assigned direction towards the site that holds the token.
- Site which holds the token is also called root of the tree.

Data structure and Notations:

- Every site S_i keeps a FIFO queue, called **request_q**

  This queue stores the requests of all neighbouring sites that have sent a request for the token to site S_i but have not yet been sent token. A non-empty request_q at any site indicates that the site has sent a REQUEST message to the root node.

  Every site S_i has a local variable, called holder. This variable points to an immediate neighbour node on a directed path to the root node.

Algorithm

- **To enter Critical section:**
  - (a) When a site S_i wants to enter the critical section it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its request_q is empty.
  - (b) After sending REQUEST message it add its request to its request_q.
  - (c) When a site S_j on the path to the root receives the REQUEST message of site Si, it places the REQUEST in its request_q and sends the REQUEST message along the directed path to the root, if it has not sent any REQUEST message for previously received REQUEST message.
  - (d) When the root site S_r (having token) receives the REQUEST message, it sends the token to the requesting site and sets its holder variable to point at that site.

(e) On receiving the token, Site S_j deletes the top entry from its request_q and sends the token to the site indicated by deleted entry. holder variable of Site S_j is set to point at that site.

(f) After deleting the topmost entry of the request_q, if it is still non-empty Site S_j sends a REQUEST message to the site indicated by holder variable in order to get token back.

To execute the critical section

- (a) Site Si executes the critical section if it has received the token and its own entry is at the top of its request_q.

To release the critical section

- After finishing the execution of the critical section, site Si does the following:

(a) If its request_q is non-empty, then it deletes the topmost entry from its request_q and then it sends the token to that site indicated by deleted entry and also its holder variable is set to point at that site.

(b) After performing above action, if the request_q is still non-empty, then site S_i sends a REQUEST message to the site pointed by holder variable in order to get token back.

Message Complexity

- In the worst case, the algorithm requires 2 x (Longest path length of the tree) message invocation per critical section entry.
- If all nodes are arranged in a straight line, then the longest path length will be N – 1 and thus the algorithm will require 2 x (N -1) message invocation for critical section entry.
- However, if all nodes generate equal number of REQUEST messages for the privilege, the algorithm will require approximately 2 x (N / 3) messages per critical section entry.

Drawbacks of Raymond's tree based algorithm:

Can cause starvation: Raymond's algorithm uses greedy strategy as a site can executes the critical section on receiving the token even when its request is not on the top of the request queue. This affects the fairness of the algorithm and thus can cause in starvation.

Performance

- Synchronization delay is (T x logN)/ 2, because the average distance between two sites to successively execute the critical section is (logN)/2. Here, T is maximum message transmission time.
- In heavy load conditions, the synchronization delay become T because a site executes the critical section every time the token is transferred.
- The message complexity of this algorithm is O(logN) as the average distance between any two nodes in a tree with N nodes is logN.
- Deadlock is impossible.

3.8 COMPARATIVE PERFORMANCE ANALYSIS OF MUTUAL EXCLUSION ALGORITHMS

The performance of mutual exclusion algorithms is measured by the following metrics:

1. **Synchronization Delay (SD):** It is the time interval between critical section exit and new entry by any process.
2. **System Throughput(ST):** It is the rate at which requests for the critical section gets executed.
3. **Message Complexity (MC):** It is the number of messages required per critical section execution by a process.
4. **Response Time (RT):** It is the time interval from a request send to its critical section execution.

Also, a mutual exclusion algorithm should satisfy the following properties:

1. **Safety Property**: At most one process may execute in the critical section at a time.
2. **Liveness Property**: A process requesting entry to the critical section is eventually granted to it. There should not be deadlock and starvation.
3. **Fairness Property**: Each process should get a fair chance to execute a critical section.

Table 3.8.1 : Performance Analysis of Mutual Exclusion Algorithms

Algorithm	Type	Messages Complexity (High Load)	Messages Complexity (Low Load)	Synchronization Delay (SD)	System Throughput (ST)	Correctness Properties
Lamport's Algorithm	Non-token based	3(N-1)	3(N-1)	T	1/(T+E)	Safety, Fairness and Liveness
Ricart-Agrawala Algorithm	Non-token based	2(N-1)	2(N-1)	T	1/(T+E)	Safety, Fairness and Liveness
Maekawa's Algorithm (N is quorum size)	Token based, Quorum based	$5\sqrt{N}$	$3\sqrt{N}$	2T	1/(2T+E)	Safety, Fairness, but no Liveness
Suzuki-Kasami's Algorithm	Token based, broadcast	0 (If the requesting site holds the token)	N	0 or T	1/(T+E)	Safety, Liveness, Unfair SD and Message complexity high if N is large
Singhal's Dynamic Algorithm	Token based, heuristic	3(N-1)/2	(N-1)	T	1/(T+E)	Safety, Fairness and Liveness
Raymond's Tree Based Algorithm (k is depth)	Token based, tree based	4	logN	T(logN)/2	1/(T+E)	Safety, Fairness and Liveness

3.8.1 Comparison of Token based and Non-token based Mutual Exclusion Algorithms

Table 3.8.2 : Token based Vs Non-token based Mutual Exclusion Algorithms

Sr.No.	Token Based Algorithms	Non-Token Based Algorithms
1.	In the Token-based algorithm, a unique token is shared among all the sites in Distributed Computing Systems.	In Non-Token based algorithm, there is no token even not any concept of sharing token for access.
2.	Here, a site is allowed to enter the Critical Section if it possesses the token.	Here, two or more successive rounds of messages are exchanged between sites to determine which site is to enter the Critical Section next.
3.	The token-based algorithm uses the sequences to order the request for the Critical Section and to resolve the conflict for the simultaneous requests for the System.	Non-Token based algorithm uses the timestamp (another concept) to order the request for the Critical Section and to resolve the conflict for the simultaneous requests for the System.
4.	The token-based algorithm produces less message traffic as compared to Non-Token based Algorithm.	Non-Token based Algorithm produces more message traffic as compared to the Token-based Algorithm.

Sr.No.	Token Based Algorithms	Non-Token Based Algorithms
5.	They are free from deadlock (i.e. here there are no two or more processes are in the queue in order to wait for messages that will actually can't come) because of the existence of unique token in the distributed system.	They are not free from the deadlock problem as they are based on timestamp only.
6.	Here, it is ensured that requests are executed exactly in the order as they are made in.	Here there is no surety of execution order.
7.	Token-based algorithms are more scalable as they can free your server from storing session state and also, they contain all the necessary information which they need for authentication.	Non-Token based algorithms are less scalable than the Token-based algorithms because server is not free from its tasks.
8.	Here the access control is quite Fine-grained because here inside the token roles, permissions and resources can be easily specifying for the user.	Here the access control is not so fine as there is no token which can specify roles, permission, and resources for the user.
9.	Token-based algorithms make authentication quite easy.	Non-Token based algorithms can't make authentication easy.
10.	Examples of Token-Based Algorithms are: (i) Singhal's Heuristic Algorithm (ii) Raymonds Tree Based Algorithm (iii) Suzuki-Kasami Algorithm	Examples of Non-Token Based Algorithms are: (i) Lamport's Algorithm (ii) Ricart-Agarwala Algorithm (iii) Maekawa's Algorithm

3.9 DEADLOCK

- A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource occupied by some other process.
- In distributed environment, if the sequence of resource allocation to processes is not controlled, a deadlock may occur.
- In principle, deadlocks in distributed systems are similar to deadlocks in centralized systems. Therefore, the description of deadlocks presented above holds good both for centralized and distributed systems.
- However, handling of deadlocks in distributed systems is more complex than in centralized systems because the resources, the processes, and other relevant information are scattered on different nodes of the system.
- Three commonly used strategies to handle deadlocks are as follows:

(I) **Avoidance :** Resources are carefully allocated to avoid deadlocks.

(II) **Prevention :** Constraints are imposed on the ways in which processes request resources in order to prevent deadlocks.

(III) **Detection and recovery :** Deadlocks are allowed to occur, and a detection algorithm is used to detect them. After a deadlock is detected, it is resolved by certain means.

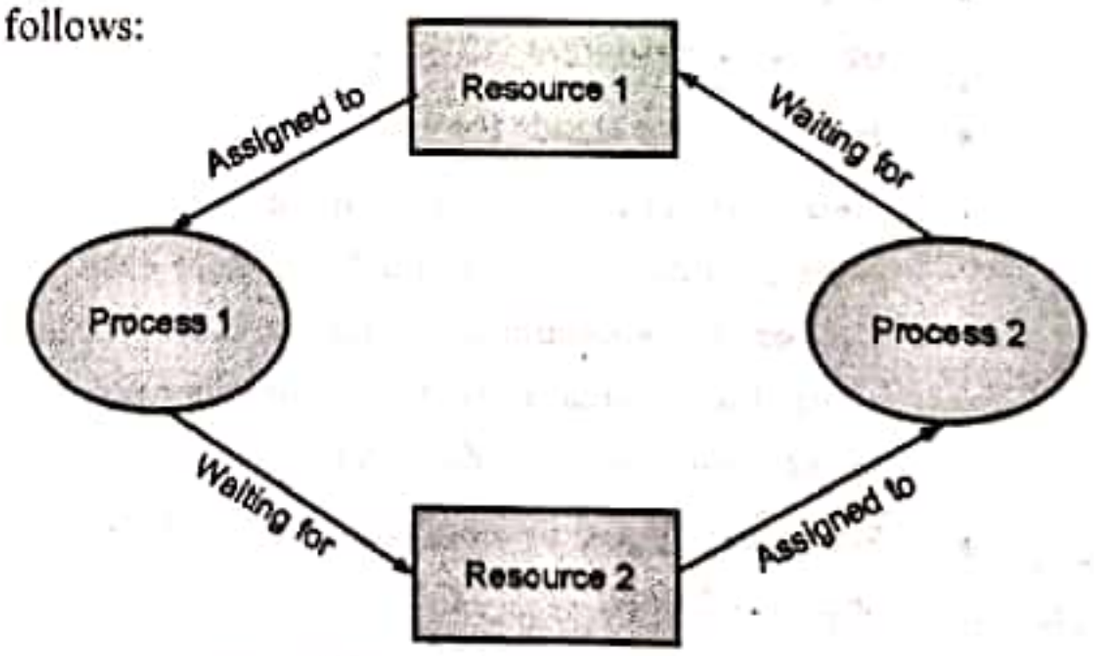


Fig. 3.9.1 : Deadlock Scenario

3.9.1 Types of Distributed Deadlock

There are two types of deadlocks in distributed system:

1. **Resource Deadlock :** A resource deadlock occurs when two or more processes wait permanently for resources held by each other.
   - A process that requires certain resources for its execution, and cannot proceed until it has acquired all those resources.
   - It will only proceed to its execution when it has acquired all required resources.
   - It can also be represented using **AND** condition as the process will execute only if it has all the required resources.
   - Example: Process 1 has R1, R2, and requests resources R3. It will not execute if any one of them is missing. It will proceed only when it acquires all requested resources i.e. R1, R2, and R3.

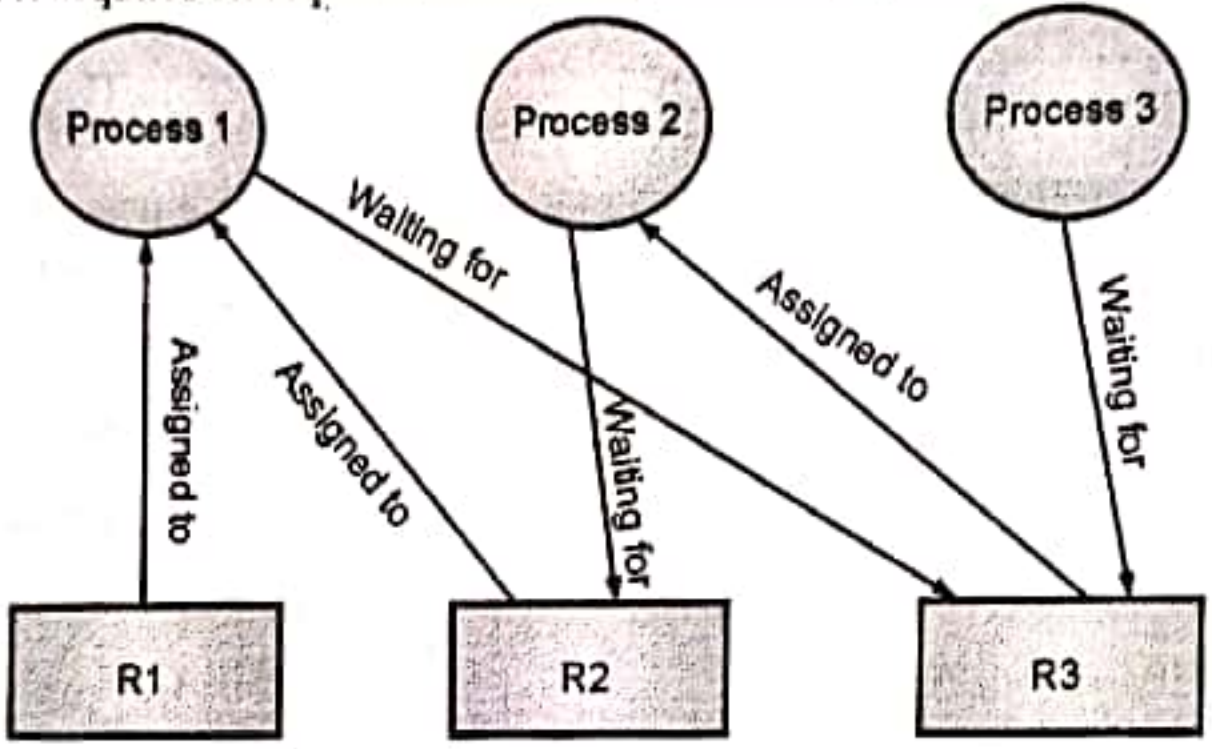


Fig. 3.9.2 : Resource Deadlock

2. **Communication Deadlock :** On the other hand, a communication deadlock occurs among a set of processes when they are blocked waiting for messages from other processes in the set in order to start execution but there are no messages in transit between them. When there are no messages in transit between any pair of processes in the set, none of the processes will ever receive a message. This implies that all processes in the set are deadlocked. Communication deadlocks can be easily modelled by using wait-for-graphs (WFGs) to indicate which processes are waiting to receive messages from which other processes. Hence, the detection of communication deadlocks can be done in the same manner as that for systems having only one unit of each resource type.
   - In Communication Model, a Process requires resources for its execution and proceeds when it has acquired at least one of the resources it has requested for.
   - Here resource stands for a process to communicate with.
   - Here, a Process waits for communicating with another process in a set of processes. In a situation where each process in a set, is waiting to communicate with another process which itself is waiting to communicate with some other process, this situation is called communication deadlock.
   - For 2 processes to communicate, each one should be in the unblocked state.
   - It can be represented using **OR** conditions as it requires at least one of the resources to continue its Process.

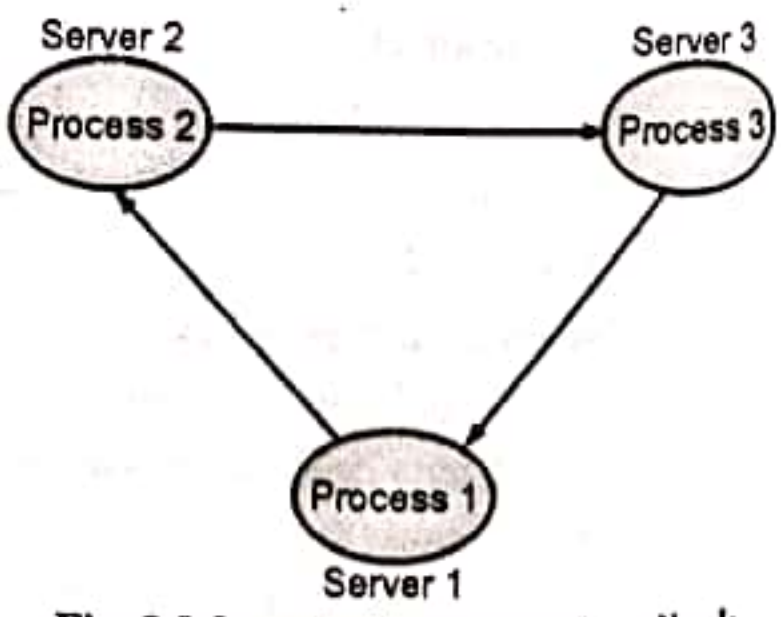


Fig. 3.9.3 : communication Deadlock

- Example: In a Distributed System network, Process 1 is trying to communicate with Process 2, Process 2 is trying to communicate with Process 3 and Process 3 is trying to communicate with Process 1. In this situation, none of the processes will get unblocked and a communication deadlock occurs.

3.9.2 Deadlock Detection

- In a distributed system, deadlock cannot be prevented nor avoided because the system is too vast.
- As a result, only deadlock detection is possible.
- The following are required for distributed system deadlock detection techniques:
  1. **Progress :** The method may detect all the deadlocks in the system.
  2. **Safety :** The approach must be capable of detecting all system deadlocks.
- Various approaches to detect the deadlock in the distributed system are as follows:
  1. **Centralized Approach :** Only one resource is responsible for detecting deadlock in the centralized method, and it is simple and easy to use. Still, the disadvantages include excessive workload on a single node and single-point failure (i.e., the entire system is dependent on one node, and if that node fails, the entire system crashes), making the system less reliable.
  2. **Hierarchical Approach :** In a distributed system, it is the integration of both centralized and distributed approaches to deadlock detection. In this strategy, a single node handles a set of selected nodes or clusters of nodes that are in charge of deadlock detection.
  3. **Distributed Approach :** In the distributed technique, various nodes work to detect deadlocks. There is no single point of failure as the workload is equally spread among all nodes. It also helps to increase the speed of deadlock detection.

3.9.2(A) Centralized Deadlock Detection Approach

- This is the technique used in distributed database system to handle deadlock detection.
- According to this approach, the system maintains one ***Global wait-for graph*** in a single chosen site, which is named as ***deadlock-detection coordinator***.
- The Global wait-for graph is updated during the following conditions:
  (i) Whenever a new edge is inserted or removed in the local wait-for graphs of any sites.
  (ii) Periodically
  (iii) Whenever the coordinator invokes the detection algorithm.
- When the deadlock-detection coordinator starts the deadlock-detection algorithm, it searches for cycles. If the coordinator finds a cycle, then the following will happen:
  (i) The coordinator selects a victim transaction that need to be rolled back.
  (ii) The coordinator informs about the victim transaction to all the sites in the distributed database.
  (iii) The sites rollback the transaction.
- This approach (centralized detection approach) may lead to unnecessary rollbacks due to one of the following: (the main cause is the communication delay.)
  1. False cycles
  2. Individual transaction rollback during a deadlock and a victim is chosen – for example; let us assume that a deadlock occurred in a distributed database. Then the coordinator chooses one victim transaction and informs the sites about the victim to rollback. At the same time, because of some other reasons, a transaction Ti rollback itself. Now the whole system performed unnecessary rollbacks.

3.9.2(B) Chandy-Misra-Hass Algorithm

- Chandy-Misra-Haas's distributed deadlock detection algorithm is an edge chasing algorithm to detect deadlock in distributed systems.
- In edge chasing algorithm, a special message called *probe* is used in deadlock detection.
- A *probe* is a triplet *(i, j, k)* which denotes that process P_i has initiated the deadlock detection and the message is being sent by the home site of process P_j to the home site of process P_k.
- The probe message circulates along the edges of WFG to detect a cycle.
- When a blocked process receives the probe message, it forwards the probe message along its outgoing edges in WFG.
- A process P_i declares the deadlock if probe messages initiated by process P_i returns to itself.
- **Other terminologies used in the algorithm**
  1. **Dependent process :** A process P_i is said to be dependent on some other process P_j, if there exists a sequence of processes P_i, P_{i1}, P_{i2}, P_{i3}..., P_{im}, P_j such that in the sequence, each process except P_j is blocked and each process except P_i holds a resource for which previous process in the sequence is waiting.
  2. **Locally dependent process:** A process P_i is said to be locally dependent on some other process P_j if the process P_i is *dependent* on process P_j and both are at same site.

Data structures

A Boolean array, **dependent$_i$**.

Initially, **dependent$_i$[j]** is false for all value of i and j.

dependent$_i$[j] is true if process P_j is dependent on process P_i.

Algorithm

Process of sending probe:

1. If process P_i is locally dependent on itself then declare a deadlock.
2. Else for all P_j and P_k check following condition:
   (a) Process P_i is locally dependent on process P_j
   (b) Process P_j is waiting on process P_k
   (c) Process P_j and process P_k are on different sites.

   If all the above conditions are true, send probe (i, j, k) to the home site of process P_k.

On the receipt of probe (i, j, k) at home site of process P_k:

1. Process P_k checks the following conditions:
   (a) Process P_k is blocked.
   (b) dependent$_k$[i] is *false*.
   (c) Process P_k has not replied to all requests of process P_j

   If all the above conditions are found to be true then:
1. Set dependent$_k$[i] to true.
2. Now, If k == i then, declare the P_i is deadlocked.
3. Else for all P_m and P_n check following conditions:
   (a) Process P_k is locally dependent on process P_m and (b) Process P_m is waiting upon process P_n and
   (c) Process P_m and process P_n are on different sites.
4. Send probe (i, m, n) to the home site of process P_n if above conditions satisfy.

Thus, the *probe* message travels along the edges of transaction wait-for (TWF) graph and when the *probe* message returns to its initiating process then it is said that deadlock has been detected.

Performance

- Algorithm requires at most exchange of $m(n-1)/2$ messages to detect deadlock. Here, m is number of processes and n is the number of sites.
- The delay in detecting the deadlock is $O(n)$.

☞ **Advantages:**

- There is no need for special data structure. A probe message, which is very small and involves only 3 integers and a two dimensional Boolean array *dependent* is used in the deadlock detection process.
- At each site, only a little computation is required, and overhead is also low
- Unlike other deadlock detection algorithm, there is no need to construct any graph or pass nor to pass graph information to other sites in this algorithm.
- Algorithm does not report any false deadlock (also called phantom deadlock).

☞ **Disadvantages**

The main disadvantage of a distributed detection algorithms is that all sites may not be aware of the processes involved in the deadlock; this makes resolution difficult. Also, proof of correction of the algorithm is difficult.

Descriptive Questions

Q. 1 Explain why clock synchronization is required in distributed system.

Q. 2 What are Physical Clocks? Explain any one Physical Clock Synchronization Algorithm.

Q. 3 Explain Berkeley Algorithm with suitable example.

Q. 4 Explain Cristian's Algorithm for clock synchronization.

Q. 5 What is a logical clock? Why are logical clocks required in distributed systems? How does Lamport synchronize logical clocks? Which events are said to be concurrent in Lamport's Timestamp?

Q. 6 Describe any one method of Logical Clock Synchronization with the help of an example.

Q. 7 What are the requirements of mutual exclusion algorithms?

Q. 8 Explain Lamport's Mutual Exclusion Algorithm.

Q. 9 Describe Ricart-Agrawala's algorithm for mutual exclusion.

Q. 10 Explain Maekawa's Mutual Exclusion Algorithm.

Q. 11 Explain Suzuki-Kasami's Token based Mutual Exclusion Algorithm.

Q. 12 Discuss Singhal's Heuristic Based Mutual Exclusion Algorithm.

Q. 13 Explain Raymond's Tree Based Mutual Exclusion Algorithm.

Q. 14 Compare Token based and Non-Token based algorithms.

Q. 15 Explain performance analysis of different mutual exclusion algorithms.

Chapter Ends...

❑❑❑

MODULE 4

CHAPTER 4

Resource and Process Management

4.1 INTRODUCTION

UQ. State the desirable features of global scheduling algorithms. (MU - Dec. 16)
UQ. Explain desirable features of global scheduling algorithm. (MU - May 18, 22)
UQ. Enlist and discuss the desirable features of global scheduling algorithm. (MU - Dec. 19)

- A group of resources linked together through a network make up distributed systems.
- To satisfy their resource needs, processes migrate. It is up to the resource manager to manage how resources are allocated to various processes.
- There are two types of resources: logical (shared files) and physical (CPU).
- Processors are assigned to run on the available resources through the scheduling mechanism.
- In a distributed computing system, the scheduling of various modules on specific processing nodes may be preceded by appropriate task-specific module allocation to various processing nodes, and only then can the proper execution characteristic be attained.
- Within the Distributed Computing System's operating system, work schedule, or task allocation, becomes the most crucial and significant operation.

4.1.1 Desirable Features of Global Scheduling Algorithms

The desirable Features of a Global Scheduling algorithm are as follows:

(1) General Purpose	(2) Efficiency
(3) Dynamic	(4) Fairness
(5) Transparency	(6) Scalability
(7) Fault Tolerance	(8) Quick Decision-Making Ability
(9) Balanced System Performance	(10) Stability

(1) General Purpose

- The sorts of applications that can be run should not be subject to many restrictions or assumptions in a scheduling strategy.
- Effective performance should be provided for interactive jobs, distributed and parallel applications, as well as batch jobs that are not interactive.
- Although this attribute is simple, it might be challenging to attain.
- The requirements given to the scheduler may conflict because different job types demand different qualities.
- A trade-off might need to be made in order to attain the overall goal.

(2) Efficiency

It has two meanings: first, it should maximize the efficiency of planned tasks; second, the overhead associated with scheduling should be kept to a minimum so as not to negate the advantages.

(3) Dynamic

The algorithms that are used to determine where to perform a job should adapt to changes in load and achieve maximum utilization of the resources at their disposal.

(4) Fairness

- When there is a high demand for resources, sharing them among users presents new difficulties in ensuring that each user gets his or her fair share.
- This issue might worsen in a distributed system when one person ends up utilizing pretty much the entire system.
- Fairness can be attained on a single node using a variety of pre-established mechanisms.

(5) Transparency

- The host(s) on which a job performs shouldn't have an impact on how it behaves or the way it turns out.
- There should be no distinction between local and remote execution.
- A user shouldn't have to invest any effort to decide where to perform a task or to start remote execution; they shouldn't even be aware that remote processing is underway.
- Furthermore, the applications should not undergo any significant changes or modifications.
- The application programs should not have to be altered in order to run on the system.

(6) Scalability

- As new nodes are added, a scheduling system should be able to scale well.
- A scheduling strategy is said to have poor scalability if it selects the node with the lowest workload after first polling all the nodes about their workload.
- This will only function properly if there aren't too many nodes in the system.
- This is because the inquirer receives a deluge of responses practically instantaneously, and as the number of nodes (N) rises, it takes too long to process the response messages in order to choose a node.
- Additionally, network capacity is greatly consumed due to this activity.
- To choose a node, a straightforward strategy is to investigate only M out of N nodes.

(7) Fault Tolerance

- The failure of one or more system nodes should not hinder an effective scheduling algorithm.
- The algorithm should also be capable of functioning appropriately if a group of nodes is split into two or more groups because of link failures.
- Algorithms that implement a decentralized decision-making capability that only consider the active nodes in the system have a superior fault tolerance ability.

(8) Quick Decision-Making Ability

It is preferred to use heuristic strategies over exhaustive (optimal) solution methods since they require less computational work (and therefore take less time) while still producing close to optimal results.

(9) Balanced System Performance

- It is preferable to choose algorithms that deliver close to optimal system performance with a minimum amount of global state information gathering overhead (such as CPU load).
- This is because overhead increases as more data on the global state are gathered.
- This is because the cost of collecting and processing the extra information reduces the usability of the information due to both the aging of the data being collected and the low scheduling frequency.

(10) Stability

- Processor thrashing, or the wasteful transfer of processes, must be avoided.

- For instance, suppose nodes n1 and n2 see that node n3 is idle and decide to transfer some of their work to n3 without being aware of the other node's decision to offload.
- Now, if n3 experiences overload as a result of this, it may once more begin transferring its processes to other nodes.
- This is brought on by the fact that each node's scheduling choices are made independently of those made by other nodes.

4.2 TASK ASSIGNMENT APPROACH

- As the name implies, the task assignment approach is based on the division of the original process into several smaller tasks.
- To increase performance and efficiency, these tasks should be assigned to the most suitable processors.
- However, this approach has a significant drawback because it requires prior knowledge of the characteristics of all the involved processes.
- Additionally, it disregards the dynamically shifting state of the system.
- As it is based on the division of tasks in a system, this approach's primary objective is to redistribute the tasks of a single process in the best manner feasible.
- To do that, the right approach for its implementation must be determined.
- For the Task Assignment Approach, the following presumptions are made:
  (i) A process has already been divided up into sections called tasks.
  (ii) This division occurs along natural boundaries (such as a method), so that each task can have consistency and data transfers among the jobs are limited.
  (iii) The amount of processing required by each task and the speed of each CPU are known.
  (iv) The cost of executing each task on every node is known.
  (v) The IPC cost between every pair of tasks is already known.
  (vi) For tasks assigned to the same node, the IPC cost is 0.
  (vii) If two tasks communicate *'n'* times and the typical inter-task communication duration is *'t'*, then the total IPC (Inter Process Communication) cost for the two tasks is *'n * t'*.
  (viii) Relationships between the tasks in terms of precedence are known.
  (ix) It is not feasible to reassign tasks.
- The purpose of this technique is to divide a process's tasks among the available nodes of a distributed system to accomplish the following objectives:
  - Minimization of IPC costs.
  - Quick turnaround time for the entire procedure.
  - The high degree of parallelism.
  - Optimal use of the available system resources.
- These objectives however usually contradict each other due to the basic requirements needed to achieve them.
- For instance, in order to reduce IPC costs, it is feasible to assign all of a process's tasks to a single node.
- However, efficient utilization of system resources implies an equal distribution of the tasks among all the nodes.
- Likewise, a high degree of parallelism and a quick turnaround time encourage parallel task execution, whereas the precedence relationship i.e., the order in which the tasks are executed limits their parallel execution.

- In the case of m tasks and q nodes, there are m^q possible ways to assign the tasks to nodes. In practice, however, the actual number of possible assignments of tasks to nodes may be less than m^q due to the restriction that certain tasks cannot be assigned to certain nodes due to their specific requirements.

4.3 LOAD BALANCING APPROACH

UQ. Explain the Load Balancing approach in a distributed system. (MU - Dec. 16)
UQ. Differentiate between Job scheduling and load balancing. Discuss the issues in designing load-balancing algorithms. (MU - May 17)
UQ. Write a short note on Load Balancing Techniques. (MU - Dec. 18)
UQ. Explain the load-balancing approach. Explain static and dynamic load balancing algorithms. (MU - Dec. 19)
UQ. Discuss the different issues and steps involved in developing a good load-balancing algorithm. (MU - May 22)

- The function of a load balancer is to distribute network or application traffic among several available servers by acting as a reverse proxy.
- The method of distributing load units (i.e., tasks and assignments) around the organization that is connected to the distributed system is called load adjusting. Thus, a load balancer operates as the "traffic cop" in front of your servers, distributing client requests across all servers equipped to handle them in a way that maximizes speed and capacity utilization and makes sure that no server is overworked, which can result in performance degradation.
- The load balancer routes traffic to the active servers in case one server goes offline. The load balancer initiates requests to a new server when it is added to the server group.
- In this way, a load balancer accomplishes the following tasks:

(i) Efficiently divides client requests or network strain among several servers.
(ii) Ensures high availability and dependability by directing requests to only active servers.
(iii) Provides the flexibility to add or remove servers as needed.

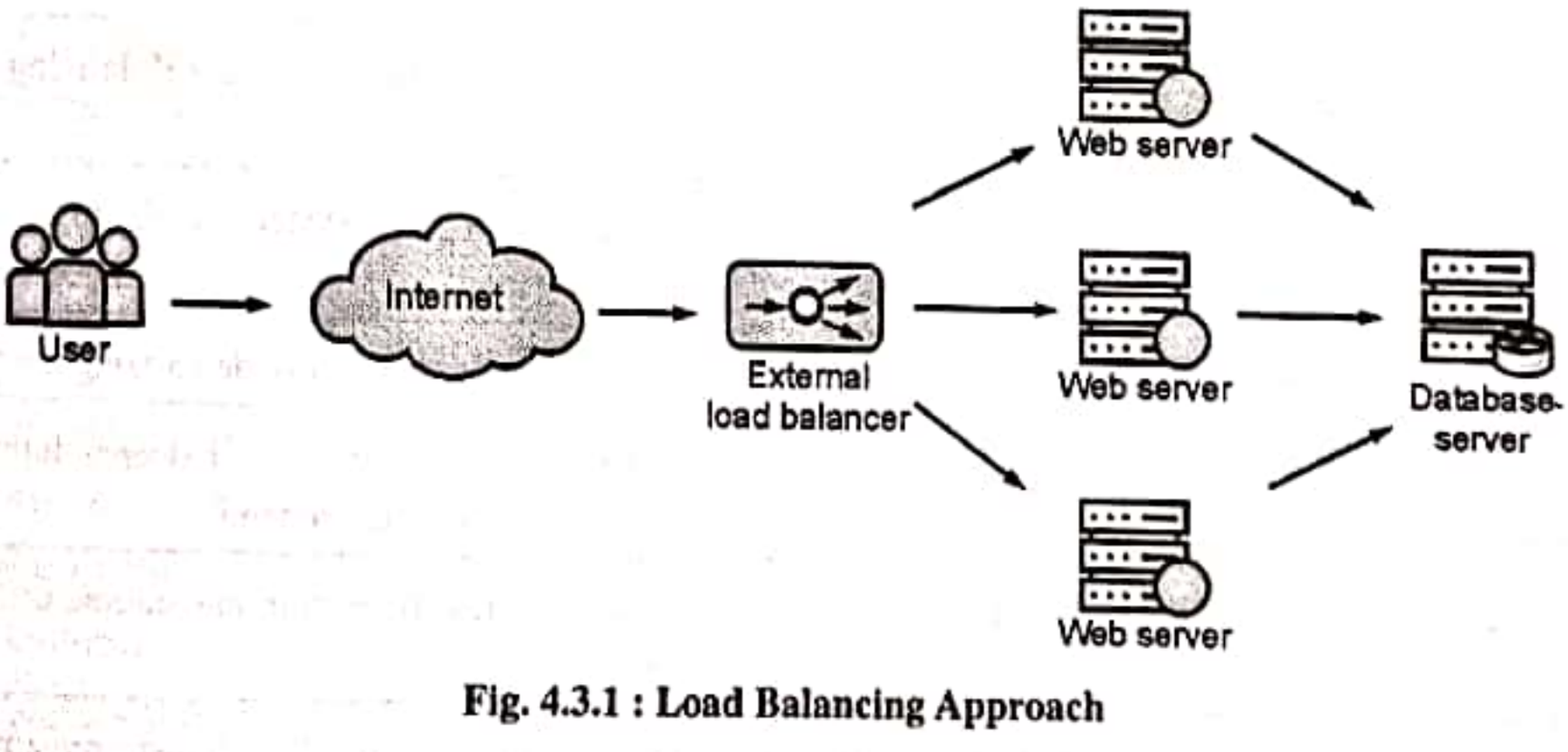


Fig. 4.3.1 : Load Balancing Approach

4.3.1 Purpose of Load Balancing in Distributed Systems

(1) **Security :** Almost no modifications to your application are required for a load balancer to supply safety to your website.
(2) **Protection from threats :** The Web Application Firewall (WAF) in the load balancer defends your website from new threats to apps.
(3) **Authenticate User Access :** To prevent unauthorized access, the load balancer may request a username and secret key before granting access to your site.

(4) **Protect against DDoS attacks :** Using a load balancer that can identify and filter out DDoS traffic before it reaches your website can be useful in protecting your website from DDoS attacks.

(5) **Performance :** Burden balancers can forward traffic for a better client experience and lessen the load on your web servers.

(6) **SSL Offload :** By using SSL (Secure Sockets Layer) to secure traffic on the load balancer, we can free up more resources for our online application by removing the need for upward traffic from web servers.

(7) **Traffic Compression :** By packing site traffic, a load balancer can significantly improve your visitors' experience on your site.

4.3.2 Classification of Load Balancing Algorithms

Fig. 4.3.2 below depicts the classification of load balancing algorithms.

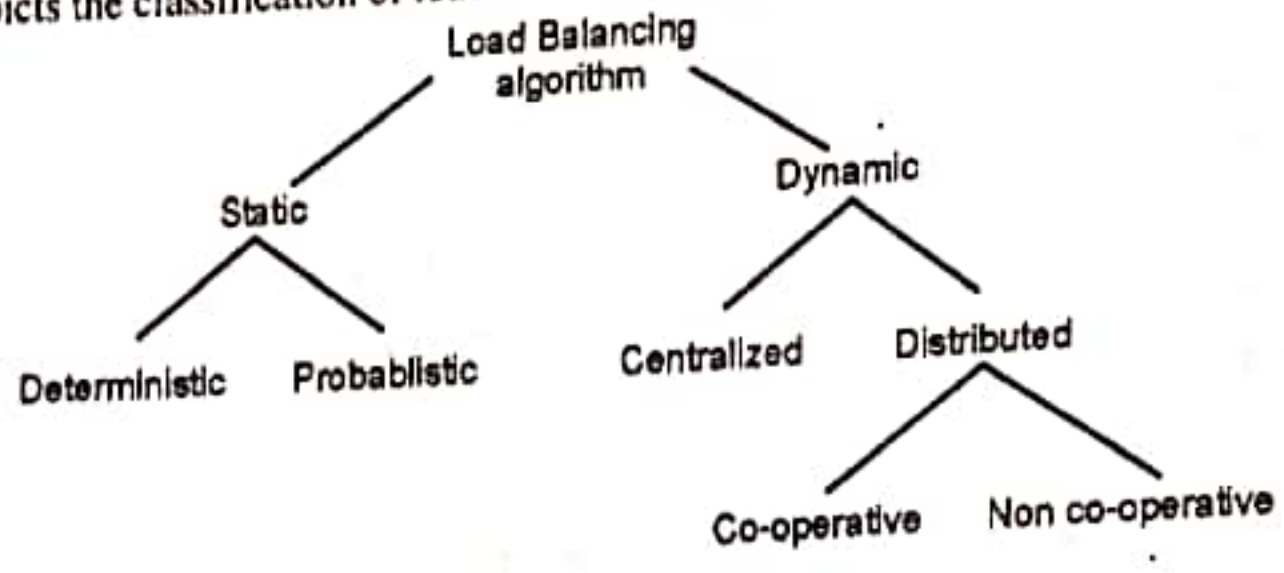


Fig. 4.3.2 : Classification of Load Balancing Algorithms

(1) Static vs Dynamic

Sr. No.	Static Load Balancing	Dynamic Load Balancing
1.	Designed for the system with low fluctuation in incoming load.	Designed for the system with high fluctuation in incoming load.
2.	Traffic is equally divided among the servers.	Traffic is dynamically divided among the servers.
3.	It requires deeper information about available system resources.	It does not necessarily need deeper information about system resources beforehand.
4.	It does not require real-time communication with the servers.	It requires real-time communication actively with the servers.
5.	The allocated load cannot be retransferred to other servers during runtime.	The allocated load can be retransferred among servers to reduce the under-utilization of resources.
6.	Example: Round Robin algorithm for load balancing.	Example: Least Connection algorithm for load balancing.

(2) Deterministic versus Probabilistic

Sr. No.	Deterministic Load Balancing	Probabilistic Load Balancing
1.	Deterministic algorithms use information about the properties of the nodes and the characteristic of processes to be scheduled.	Probabilistic algorithms use the information of static attributes of the system (e.g., number of nodes, processing capability, topology) to formulate simple process placement rules.
2.	A deterministic approach is difficult to optimize.	The probabilistic approach has poor performance.

(3) Centralized versus Distributed

Sr. No.	Centralized Load Balancing	Distributed Load Balancing
1.	Centralized approach collects information to server node and makes assignment decision.	Distributed approach contains entities to make decisions on a predefined set of nodes.
2.	Centralized algorithms can make efficient decisions, have lower fault-tolerance.	Distributed algorithms avoid the bottleneck of collecting state information and react faster.

(4) Cooperative versus Non-cooperative

Sr. No.	Cooperative Load Balancing	Non-cooperative Load Balancing
1.	In Cooperative algorithms distributed entities cooperate with each other.	In Non-cooperative algorithms entities act as autonomous ones and make scheduling decisions independently from other entities.
2.	Cooperative algorithms are more complex and involve a larger overhead.	Non-cooperative algorithms are simpler.
3.	The stability of Cooperative algorithms is better	The stability of non-cooperative algorithms is poor.

4.3.3 Issues in Load Balancing Algorithms

The following are the issues related to load balancing algorithms :

(1) **The load estimation policy :** Chooses how to estimate a node's workload.

(2) **Process transfer policy :** Specifies whether a process should be run locally or remotely.

(3) **State information exchange policy :** Identifies the best way for nodes to share load information.

(4) **Location Policy :** Determines which node should be assigned to the transferable process.

(5) **Priority assignment policy :** Defines the order in which local and remote processes should be executed.

(6) **Migration restricting policy :** Limits the number of times a process can migrate.

(A) Load Estimation Policies

Policy I

- It is vital to determine how to measure the burden of a specific node in order to balance the workload across all of the system's nodes.
- The following are some quantifiable parameters (with time and node dependent factors):
  - Total number of processes on the node
  - Resource demands of these processes

- Instruction mixes of these processes
- Architecture and speed of the node's processor

- Numerous load-balancing techniques maximise efficiency by using the total number of processes.

Policy II

- The genuine load may occasionally change significantly depending on the remaining service time, which can be determined in a variety of ways:
  - **The memory-less** technique makes the assumption that each process has the same predicted remaining service time, regardless of how much time has already been utilised.
  - **Past repeats** make the assumption that the amount of service time left is equal to the amount of service time already consumed.
  - According to the **distribution approach**, the linked process's remaining service time is the anticipated remaining time adjusted for the time already consumed if the distribution service times are known.

Policy III

- Modern systems cannot employ any of the aforementioned techniques because of daemons and programmes that run on a regular basis.
- Measuring the CPU utilisation of the nodes would be a suitable technique for use as the load estimation policy in these systems.
- The number of CPU cycles that are actually executed for each unit of real time is known as central processing unit usage.
- It can be determined by setting up a timer to check the CPU's idle/busy condition on a regular basis.

(B) Threshold Policy

- The threshold policy is typically used by algorithms to determine whether a node is lightly burdened or highly burdened.
- A threshold value is a value that limits a node's workload and can be established by:
  - **Static policy :** a threshold value that is predetermined for each node based on processing power.
  - **Dynamic policy :** threshold value is determined by a predetermined constant and the average workload
- When a node's threshold value is below the threshold, it accepts the processes for execution; whereas when it is above the threshold, it seeks to transfer the processes to a node that is not overloaded.

(C) Location Policies

- **Threshold method**

  This policy chooses a random node, determines whether the node can receive the process, and then transfers the process. Another node is chosen at random if the first rejects. This continues until the probe limit is reached.

- **Shortest method**
  - Randomly selected L unique nodes are polled to ascertain their load. Unless its workload value prevents it from accepting the process, the process is shifted to the node with the smallest value.
  - A straightforward enhancement is to stop probing if a node has zero load.

(D) Priority Assignment Policy

- **Selfish :** Priority is given to local processes over remote processes. Worst response time performance among the three policies.
- **Altruistic :** Remote processes are prioritized over local ones. among the three policies, response times performance is the best.

- **Intermediate :** Local processes are given precedence over remote processes when the number of local processes is more than or equal to the number of remote processes. Otherwise, local processes are prioritized below remote ones.

(E) Migration Limiting Policy

This policy determines the maximum number of migrations that a process may undergo.

- **Uncontrolled :** A process may be migrated indefinitely. A distant process arriving at a node is considered the same as a process coming from a node.
- **Controlled :** These policies prevent the volatility of uncontrolled ones.
  - Uses the migration count parameter to set a cap on the number of times a process can migrate.
  - Migration policy is unchangeable; with the migration count always equal to 1.
  - To respond to dynamically changing situations, the migration count for extended execution processes must be more than 1.

4.4 LOAD-SHARING APPROACH

UQ Compare Load-sharing to task assignment and Load balancing strategies for the scheduling process in a distributed system. (MU - May 16)

The following issues with the load-balancing approach prompted the load-sharing approach:

- A load-balancing technique that attempts to balance the workload on all nodes is not an appropriate object because gathering exact state information generates significant overhead.
- Load balancing is impossible because the number of processes in a node is constantly changing and there is always a temporal imbalance among the nodes.

4.4.1 Basic Idea

- It is both necessary and sufficient to keep nodes from becoming idle while other nodes have more than two processes running.
- Load-sharing is much simpler than load-balancing because it only tries to ensure that no node is idle when there are several other heavily loaded nodes.
- The load-balancing algorithms' priority assignment and migration limiting policies are the same.

(A) Load Estimation Policies

- Since load-sharing algorithms merely try to avoid idle nodes, just knowing whether a node is busy or idle is sufficient.
- As a result, these algorithms typically use the most basic load estimation policy of counting the total number of processes.
- In modern systems, where multiple processes can run indefinitely on a single idle node, algorithms measure CPU utilization to estimate a node's load.

(B) Process Transfer Policies

- Load sharing algorithms typically employ an all-or-nothing strategy.
- This strategy employs a fixed threshold value of 1 for all nodes.
- When a node has no processes, it becomes a receiver node; when it has more than one process, it becomes a sender node.

- To avoid wastage of processing power on nodes with zero process load, load-sharing algorithms use a threshold value of 2 rather than 1.
- When CPU utilization is used as the load estimation policy, the process transfer policy should be the double-threshold policy.

(C) Location Policies

- The location policy determines whether the process's sender or receiver node takes the initiative to search for a suitable node in the system, and it can be one of the following.
- **Sender-initiated location policy**: The sender node decides where to send the process. Heavily loaded nodes look for light nodes.
- **Receiver initiated location policy**: The receiver node decides where to get the process. Nodes that are lightly loaded look for nodes that are heavily loaded.

(i) Sender-initiated location policy

- When a node becomes overloaded, it either broadcasts or randomly probes the other nodes one by one to locate a node capable of receiving remote processes.
- When broadcasting, the appropriate node is identified as soon as the response arrives.

(ii) Receiver-initiated location policy

- When a node becomes overloaded, it either broadcasts or randomly probes the other nodes to indicate its willingness to receive remote processes.
- Since scheduling decisions are typically made at process departure epochs, receiver-initiated policies necessitate a pre-emptive process migration facility.
- Both policies provide significant performance advantages over the case where no load-sharing is attempted.
- At low to moderate system loads, a sender-initiated policy is preferable.
- At high system loads, a policy initiated by the receiver is preferable.
- Because of the pre-emptive transfer of processes, sender-initiated policies outperform receiver-initiated policies.

(D) State Information Exchange Policies

- Load-sharing algorithms do not require nodes to exchange state information on a regular basis, but they must be aware of the state of other nodes when they are either underloaded or overloaded.
- When there is a state change, the following two methods are used:

(i) Broadcast when state changes

- When a node becomes overloaded or underloaded in a sender-initiated/receiver-initiated location policy, it broadcasts a State Information Request.
- When a receiver-initiated policy with a fixed threshold value of 1 is used, it is referred to as a broadcast-when-idle policy.

(ii) Poll when state changes

- Polling mechanisms are used in large networks.
- The polling mechanism asks different nodes for state information at random until an appropriate one is found, or the probe limit is reached.
- It is called poll-when-idle policy when the receiver-initiated policy is used with a fixed threshold value of 1.

4.5 PROCESS MANAGEMENT

- A process is a program under execution. A process is a 'active' entity, as opposed to a program, which is a 'passive' entity.
- When run multiple times, a single program may generate a large number of processes. For example, when we open a .exe or binary file multiple times, multiple instances begin.
- ***Process management includes operations intended to define a process, set roles, assess the performance of the process, and find areas for improvement.***
- Without a CPU executing the program's instructions, nothing happens.
- Process requires computer resources to complete its task.
- There could be multiple processes running concurrently in the system that need the same resource. As a result, the operating system must efficiently and conveniently handle all processes and resources.
- To keep the system consistent, some resources would need to be used by only one process at a time. Otherwise, the system might become inconsistent, and a deadlock may occur.

4.5.1 Process Migration

- Process migration is the act of moving a significant portion of a process' state from one computer to another.
- ***The act of moving a process between two machines is referred to as process migration.***
- The process executes on the target machine.
- Moving processes from heavily loaded to lightly loaded systems is the goal of migration.
- The performance and communication are improved as a result.
- To cut down on communications costs, processes that interact frequently can be moved to the same node.
- Process migration is intended to improve resource utilization throughout the entire system.
- A process may benefit from special hardware or software features.
- Process migration offers localised data access, failure resilience, dynamic load distribution, and simplicity of system administration.

4.5.2 Issues In Process Migration

Process migration must address the following issues:

- Who initiates the migrating process?
- What occurs during a migration?
- Which part of the procedure has been migrated?
- What happens to unfinished signals and messages?

4.5.3 Threads

- The component of a process that can be scheduled for execution is a thread.
- A process' virtual address space and system resources are shared by all its threads.
- Each thread has its own set of structures that the system will use to store the thread context until it is scheduled, along with exception handlers, a scheduling priority, thread local storage, a special thread identification, and other features.
- The kernel stack, a thread environment block, a user stack, and the thread's set of machine registers are all included in the thread context, which is located in the process's address space.

- The security context that a thread has access to can be utilised to pretend to be a client.
- Threads are frequently referred to as "lightweight processes" since they have some characteristics with processes.
- Threads enable multiple executions of streams within a process.
- A process can be single threaded or multi- threaded as shown in Fig. 4.5.1.

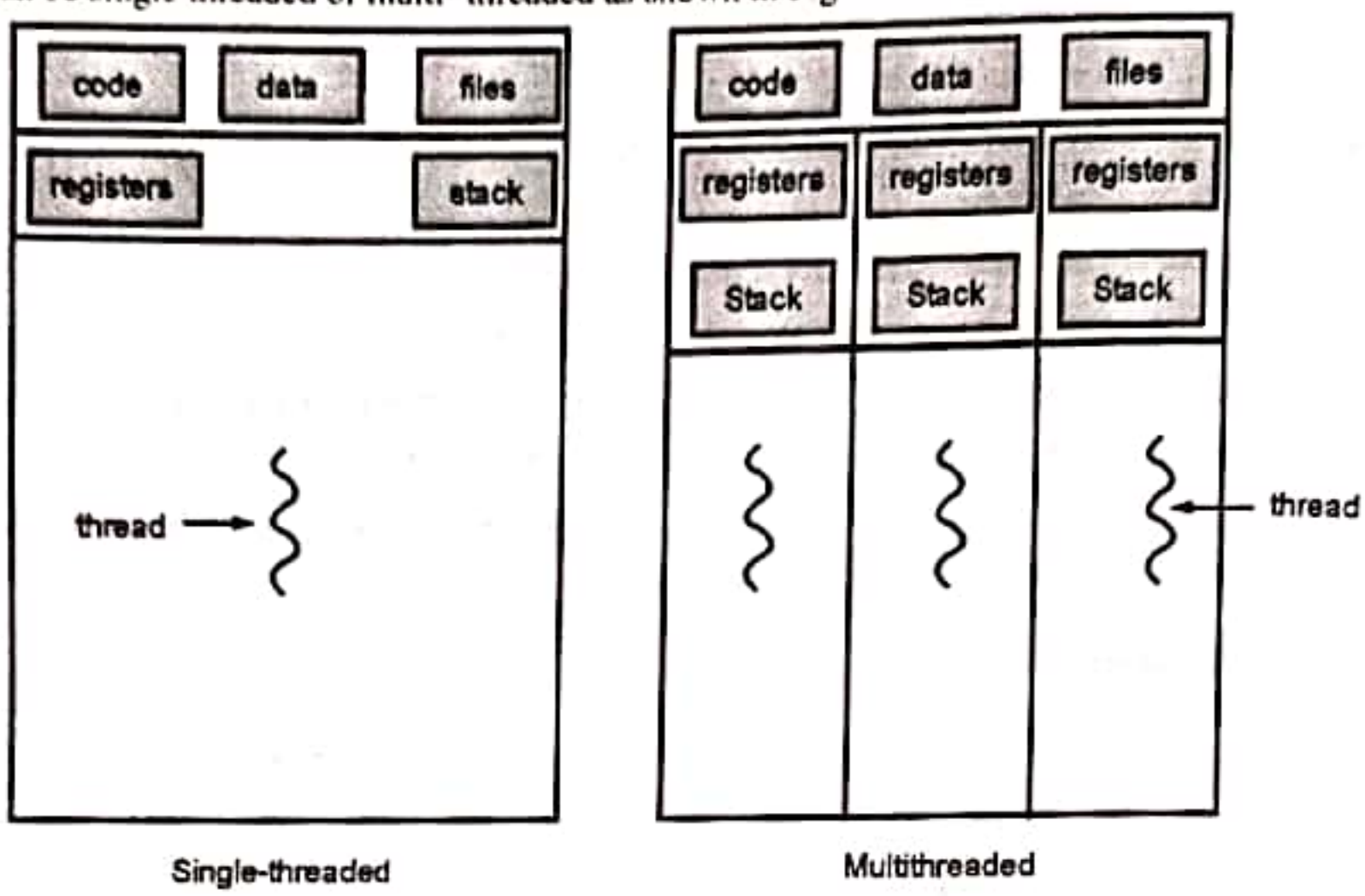


Fig. 4.5.1 : Types of threads

- With threads, multiprocessor architectures can be used more efficiently, and shared resources are available more effectively and quickly.

4.5.3.1 Difference between Process and Thread

Sr. No.	Process	Thread
1.	A Process simply means any program in execution.	Thread simply means a segment of a process.
2.	The process consumes more resources	Thread consumes fewer resources.
3.	The process requires more time for creation.	Thread requires comparatively less time for creation than process.
4.	The process is a heavyweight process	Thread is known as a lightweight process
5.	The process takes more time to terminate	The thread takes less time to terminate.
6.	Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
7.	The process takes more time for context switching.	The thread takes less time for context switching.
8.	Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.
9.	For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.

4.5.3.2 Types of Threads

Fig. 4.5.2 depicts different types of threads.

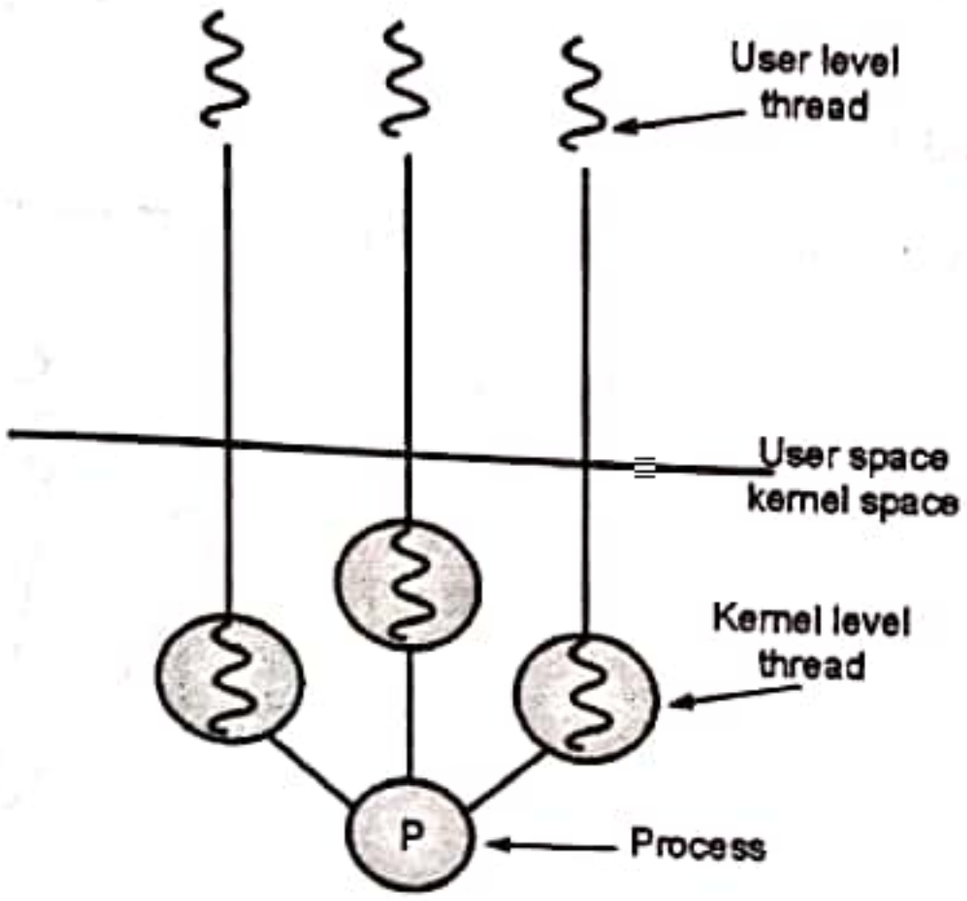


Fig. 4.5.2 : Types of Threads

(1) User Level Thread

- User-level threads are implemented and managed by the user and the kernel is not aware of it.
- User-level threads are implemented using user-level libraries and the OS does not recognize these threads.
- User-level thread is faster to create and manage compared to kernel-level thread.
- Context switching in user-level threads is faster.
- If one user-level thread performs a blocking operation, then the entire process gets blocked.
- Example: POSIX threads, Java threads, etc.

(2) Kernel-level Thread

- Kernel-level threads are implemented and managed by the OS.
- Kernel-level threads are implemented using system calls and Kernel level threads are recognized by the OS.
- Kernel-level threads are slower to create and manage compared to user-level threads.
- Context switching in a kernel-level thread is slower.
- Even if one kernel-level thread performs a blocking operation, it does not affect other threads.
- Example: Window Solaris.

4.5.3.3 Advantages of Threads

The following are the advantages of threads :

(1) Threads improve the overall performance of a program.

(2) Threads increase the responsiveness of the program.

(3) Context Switching time in threads is faster.

(4) Threads share the same memory and resources within a process.

(5) Communication is faster in threads.

(6) Threads provide concurrency within a process.

(7) Enhanced throughput of the system.

(8) Since different threads can run parallelly, threading enables the utilization of the multiprocessor architecture to a greater extent and increases efficiency.

4.5.4 Multithreading Models

Based on how user and kernel threads map onto one another, there are essentially three different multithreading architectures.

(1) Many to one (2) One to One (3) Many to many

4.5.4(A) Many to One Model

- Numerous user-level threads are mapped to a single kernel thread in this paradigm.
- Examples are GNU Portable Threads and Solaris Green Threads.

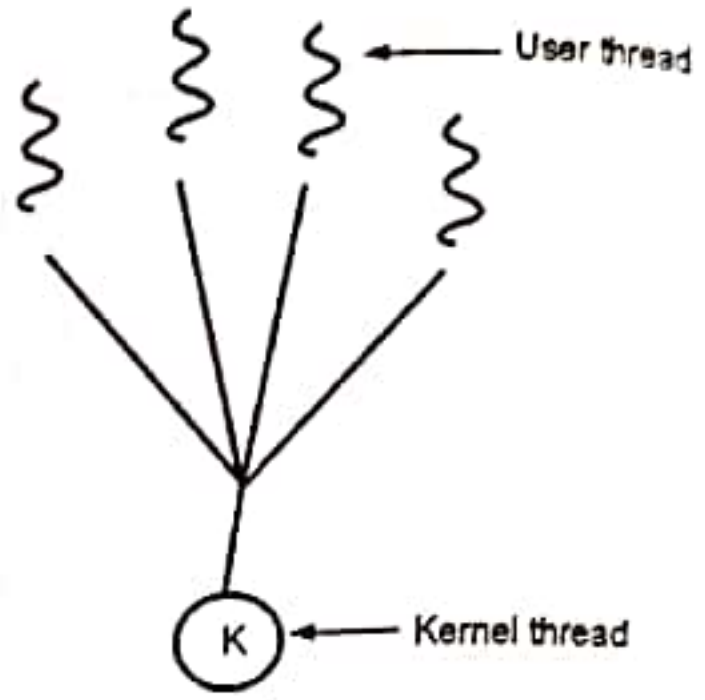


Fig. 4.5.3 : Many to One Multithreading Model

4.5.4(B) One to One Model

- Each user-level thread corresponds to a kernel thread in this approach.
- Drawback of this model is that generating a user thread necessitates making a kernel thread.
- The threads in Linux, Solaris 9 and later, Windows NT/XP/2000, are examples of One to One model.

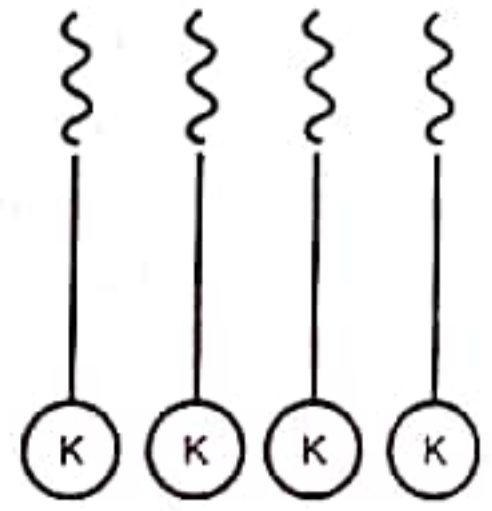


Fig. 4.5.4 : One to One Multithreading Model

4.5.4(C) Many to Many Model

- This architecture enables the mapping of numerous user level threads to numerous kernel threads.
- Additionally, it enables the operating system to generate enough kernel threads.
- Examples include Windows NT/2000 with the Thread Fiber package and Solaris prior to version 9.

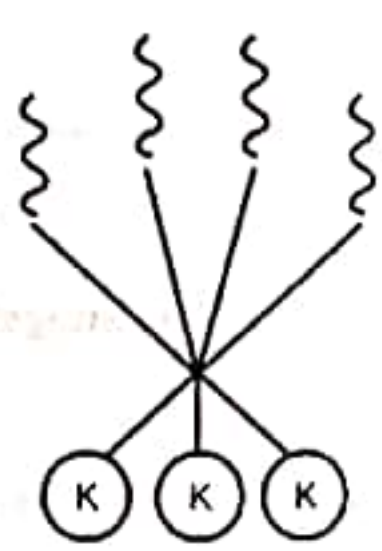


Fig. 4.5.5 : Many to Many Multithreading Model

4.5.4(D) Two Level Model

- This is like the Many-to-Many concept, with the exception that it enables the binding of a user thread to a kernel thread.
- Examples include the threads in Solaris 8 and earlier, HP-UX, IRIX, and Tru64 UNIX.

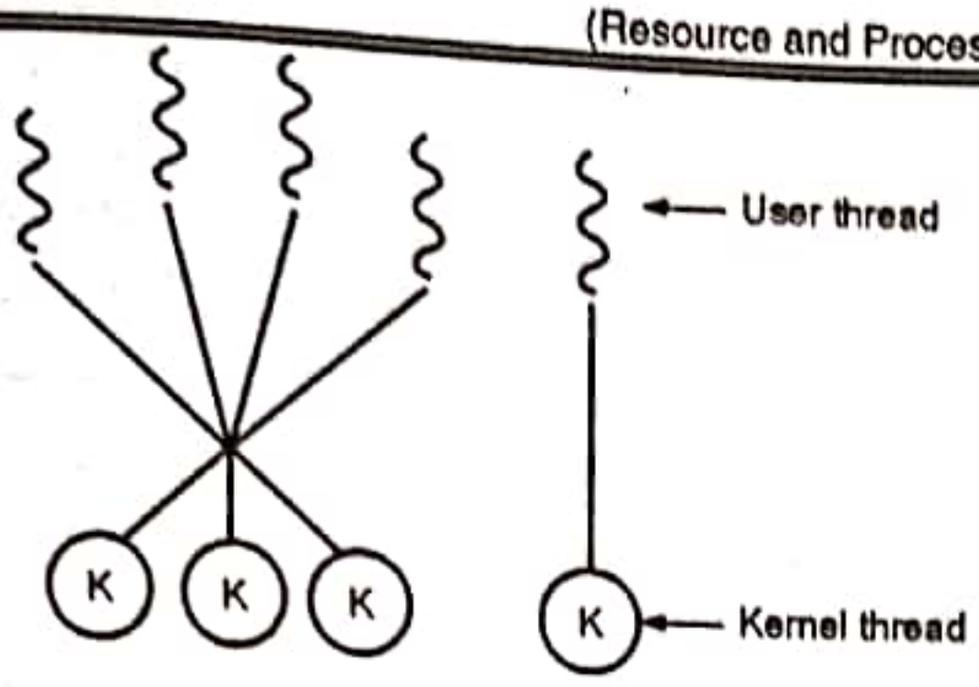


Fig. 4.5.6 : Two Level Multithreading Model

4.5.4(E) Issues in Multithreading

Below we have mentioned a few issues related to multithreading :

(1) **Thread Cancellation :** Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is Asynchronous cancellation, which terminates the target thread immediately. The other is Deferred cancellation allows the target thread to periodically check if it should be cancelled.

(2) **Signal Handling :** Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all or a single thread.

(3) **fork() System Call :** fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in the Multithreaded process is, if one thread forks, will the entire process be copied or not?

(4) **Security Issues :** Yes, there can be security issues because of the extensive sharing of resources between multiple threads.

4.6 VIRTUALIZATION

- Virtualization refers to the creation of a virtual resource such as a server, desktop, operating system, file, storage, or network.
- The main goal of virtualization is to manage workloads by radically transforming traditional computing to make it more scalable.
- Virtualization can be applied to a wide range of system layers, including operating system-level virtualization, hardware-level virtualization, and server virtualization.
- Virtualization is the most effective way to reduce IT expenses and boost efficiency large to mid-size and small organizations.
- Virtualization lets you run multiple operating systems and applications on a single server, consolidate hardware to get vastly higher productivity from fewer servers and simplify the management, maintenance, and deployment of new applications.

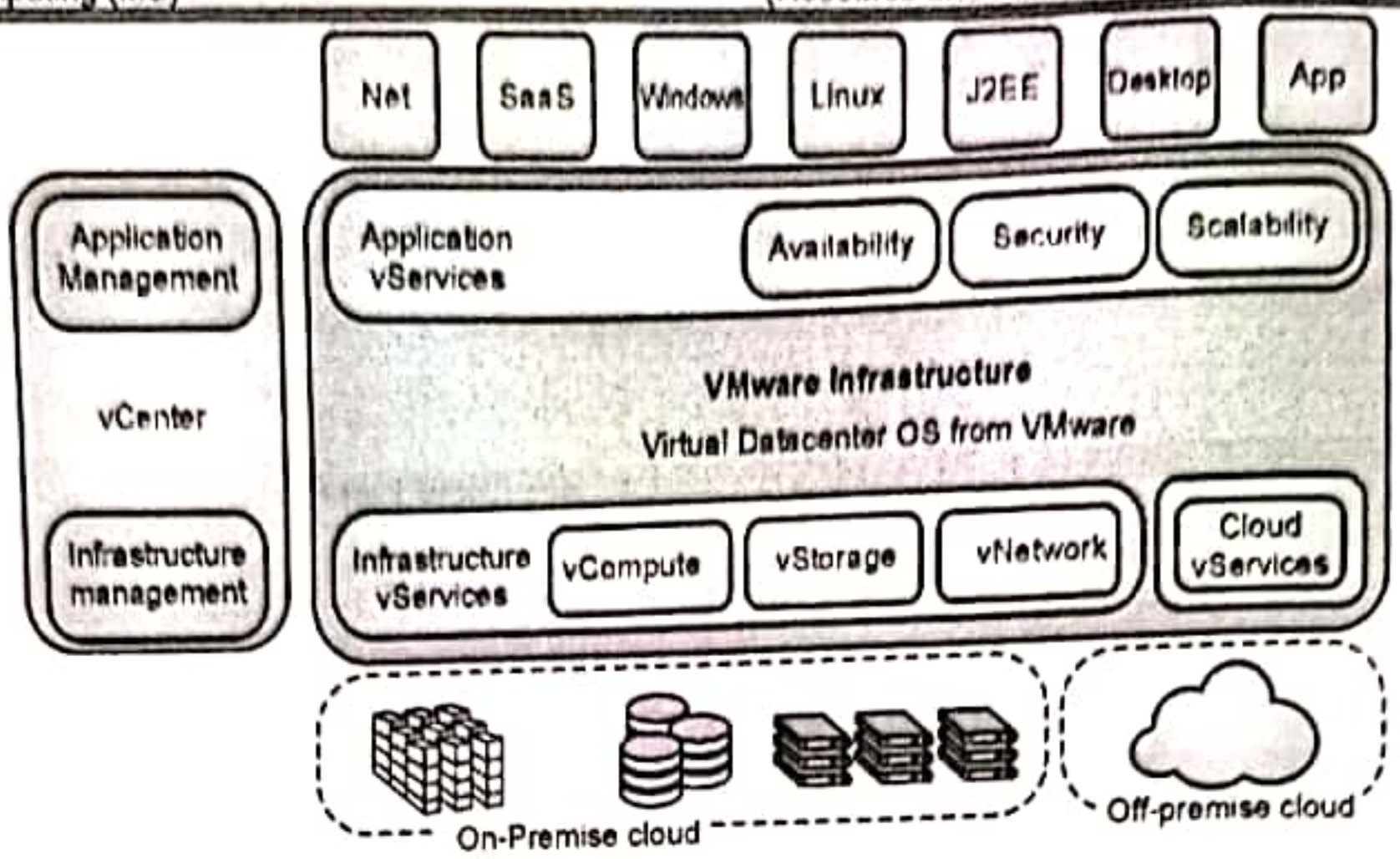


Fig. 4.6.1 : Virtualization Concept

- The term virtualization is now associated with several computing technologies including the following :
  - **(i) Storage Virtualization :** It is the amalgamation of multiple network storage devices into what appears to be a single storage unit.
  - **(ii) Server Virtualization :** It is the partitioning of a physical server into smaller virtual servers.
  - **(iii) OS Virtualization :** Multiple instances of an operating system are contained within a single device.
  - **(iv) Network Virtualization :** It is using network resources through a logical segmentation of a single physical network.
  - **(v) Application Virtualization :** An application is installed on a server but can be accessed and used as if it were installed on a user's local device.

4.6.1 Benefits of Virtualization

The following are the advantages of virtualization :

- A system VM provides a sandbox that isolates one system environment from other environments.
- Virtualization helps isolate the effects of a failure to the VM where the failure occurred.
- A single hardware platform can support multiple operating systems concurrently.
- A virtualized system can be (dynamically or statically) re-configured for changing needs.

4.6.2 Virtualization in Distributed System

- Distributed virtualization means the transparent sharing of resources by many clients.
- The clients are neither aware of each other nor of various available resources.
- There are various levels at which virtualization can be implemented to increase the performance of a distributed network.
- Each level provides a unique direction to create a more reliable, robust, cost-effective, and efficient distributed system.
- The levels are:
  - **(A) Application Virtualization :** Various organizations share access to applications as remote services. It leads to a reduction in the cost of software and infrastructure. The main concern for this level is to maintain security across multiple users.

(B) **Utility Computing :** Resources from various connected data centers are made available to be used as and when necessary. A major concern on this level is the proper implementation of resource management.

(C) **Grids :**

- **Computational Grids :** Transparent sharing of computational server resources among multiple groups across or within an organization.
- **Transactional Grids :** To support high-performance transactional applications, the distributed hardware and software resources can be shared. Main issue implementation lacks standardization.
- **Data Grids :** Data servers can be shared between various groups for easier access to distributed data. The main issue is the maintenance and consistency of the metadata.

(D) **Virtual servers :** Virtual Servers can use multiple operating systems and applications giving themthe functionality of multiple servers in one server.

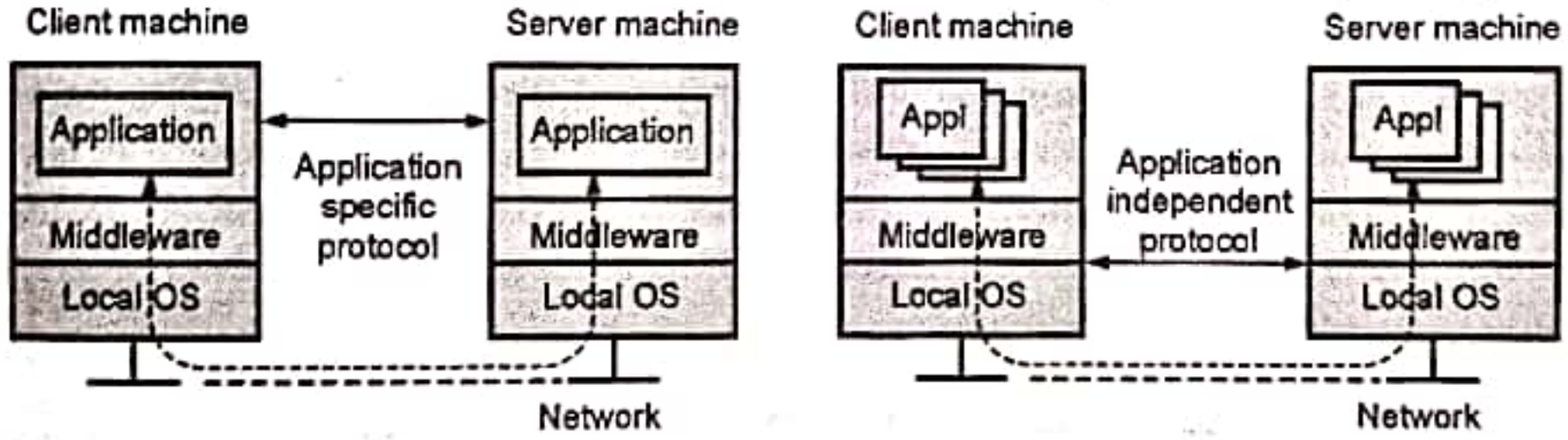
(E) **Virtual machine :** These are the nodes running their own CPU and OS in an independent environment for the execution of various applications on a portable platform.

(F) **Storage grids and utilities :** Storage virtualization creates one virtual environment of multiple storage devices which makes backup, migrating data, and storage expansion for multiple clients very efficient.

- In conclusion, Virtualization in distributed computing brings flexibility, scalability, portability, and cost advantage to an organization.
- There are some issues, challenges and concern which are to be considered and ways must be explored to overcome the serious issues like security, privacy, network management, data consistency, user and group management, maintenance and functionality of data when access is shared among different groups and organizations.

4.7 CLIENTS

- Client machines play an important role in allowing users to interact with remote servers. This interaction can be provided in about two ways.
- To begin, each remote service will have a separate counterpart on the client machine that can communicate with the service across the network. An agenda running on a user's PDA that has to synchronize with a remote, possibly shared agenda is a common example.
- Synchronization will be handled via an application-level protocol in this case, as shown in Fig. 4.7.1(a).



(a) A networked application with its own protocol

(b) A general solution to allow access to remote applications

Fig. 4.7.1

- A second solution is to provide direct access to remote services by only offering a convenient user interface. This means that the client machine is used only as a terminal with no need for local storage.
- This leads to an application neutral solution as shown in Fig. 4.7.1(b).

4.7.1 Thin-Client Network Computing

- Applications manipulate a display by using the display instructions provided by Windows X. These commands are often transmitted over the network and executed by the X kernel. Applications designed for X should, by definition, isolate application logic from user-interface commands.
- Unfortunately, this is not always the case. It turns out that most of the application logic and user interaction are tightly connected, which means that an application will send several queries to the X kernel and will wait for a response before proceeding.
- When operating over a wide-area network with long latencies, this synchronous behaviour may have a negative impact on performance.
- There are several solutions to this problem.
  (i) One is to re-engineer the implementation of the X protocol. An important part of this work concentrates on bandwidth reduction by compressing X messages.
  (ii) Messages are considered to consist of a fixed part, which is treated as an identifier, and a variable part. In many cases, multiple messages will have the same identifier in which case they will often contain similar data. This property can be used to send only the differences between messages having the same identifier.
  (iii) Both the sending and receiving sides keep a local cache in which the items can be searched using the message identification. When a message is sent, it first checks the local cache. If it is discovered, it signifies that a prior message with the same identity but perhaps different data was delivered. Differential encoding is employed in this example to communicate only the differences between the two. The message is also searched up in the local cache at the receiving end, following which decoding through the differences can take place. Standard compression techniques are applied in the cache miss, which generally results in a factor four improvement in bandwidth. Overall, this approach has shown bandwidth reductions of up to a factor 1000, allowing X to run through 9600 kbps lines.
  (iv) An important side effect of caching messages is that the sender and receiver have shared information about the present condition of the display. For example, the application can get geometric information about numerous objects by simply asking lookups in the local cache. Having this shared information alone minimizes the number of messages required to keep the application and display synchronized.
  (v) To avoid bandwidth availability becoming a concern, this approach necessitates sophisticated compression techniques. Consider presenting a video stream at 30 frames per second on a screen with a resolution of 320 x 240. A screen of this size is prevalent in many PDAs. If each pixel is represented by 24 bits, we would require a bandwidth of around 53 Mbps without compression. Compression is definitely required in this instance, and various strategies are now in use. It should be noted, however, that compression necessitates decompression at the receiver, which might be computationally expensive without hardware support. Hardware support can be given, however this boosts the price of the device.

4.7.2 Client-Side Software for Distribution Transparency

- User interfaces are only one component of client software.
- In many circumstances, parts of a client-server application's processing and data level are also executed on the client side. Embedded client software, such as that found in automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, and other devices, belongs to a distinct class.
- In many circumstances, the user interface is a minor component of the client software, compared to the local processing and communication capabilities.

- Client software includes components for achieving distribution transparency. In addition to the user interface and other application-related software.
- A client should ideally be unaware that it is connecting with remote processes. In contrast, distribution is frequently less transparent to servers for efficiency and correctness concerns.
- Access transparency is typically achieved by creating a client stub from an interface definition of what the server must give. The stub provides the same interface as the server, buthides any changes in machine architectures as well as actual communication.
- There are different ways to handle location, migration, and relocation transparency. In this case, the client's middleware can hide the server's current geographical location from the user, and also transparently rebind to the server if necessary. At worst, the client's application may notice a temporary loss of performance.
- Similarly, many distributed systems use client-side methods to enable replication transparency. Consider a distributed system with replicated servers. Replication can be accomplished by forwarding a request to each replica, as shown in Fig. 4.7.2. Client-side software can transparently collect all responses and give them to the client application as a single response.
- Finally, consider failure transparency. Client middleware is commonly used to mask communication problems with a server. Client middleware, for example, can be configured to repeatedly attempt to connect to a server, or to try another server after multiple attempts. There are even cases where the client middleware returns data stored during a prior session, like Web browsers do when they fail to connect to a server.
- Concurrency transparency can be handled by specific intermediary servers, such as transaction monitors, and requires less client software support. Similarly, persistent transparency is frequently handled entirely at the server.

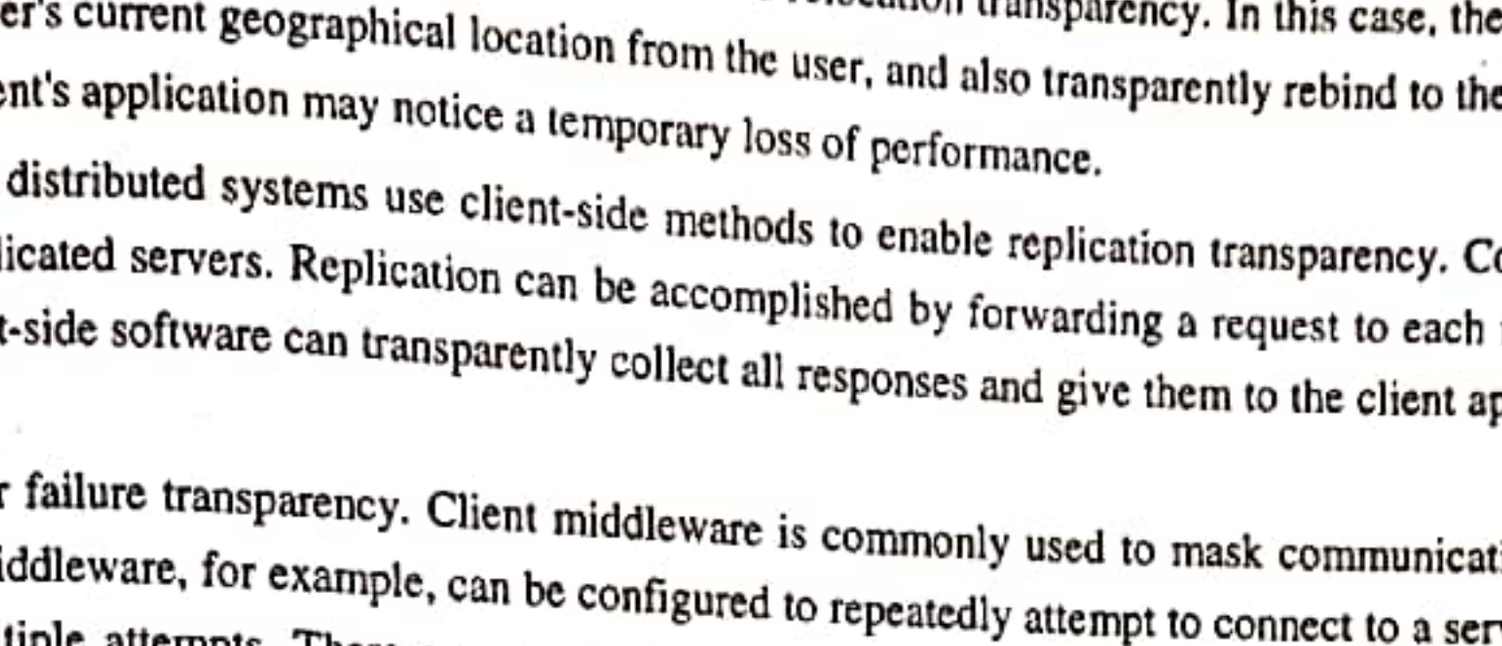


Fig. 4.7.2 : Transparent replication of a server using a client-side solution

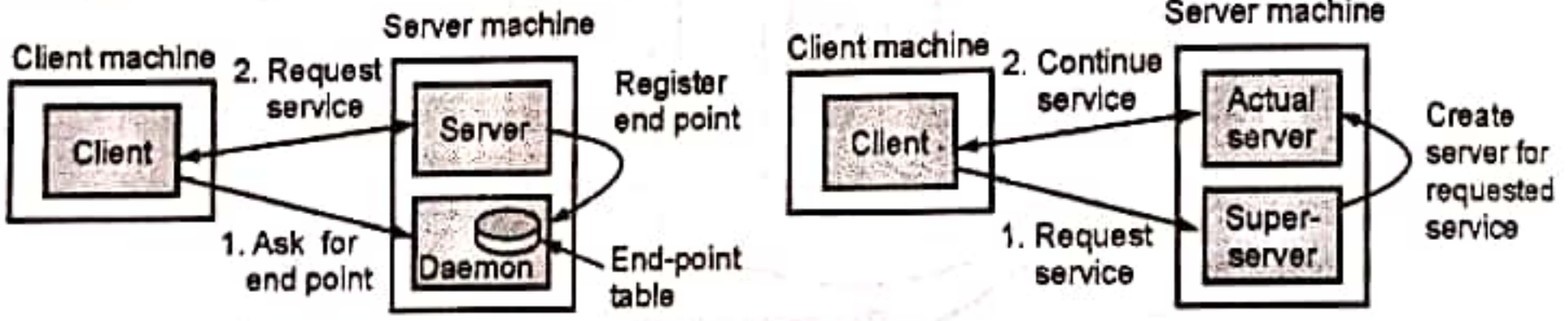
4.8 SERVERS

- Let us now take a closer look at server organization. In the following section, we will first focus on a variety of generic server design issues, followed by a discussion of server clusters.

4.8.1 General Design Issues

- A server is a process that performs a certain service on behalf of a group of clients.
- In essence, each server is structured in the same way and performs the following functionality :
  (i) It waits for an incoming request from a client
  (ii) Then it verifies that the request is handled before waiting for the next incoming request.

- Following are the ways to organize servers :
  - (i) **Iterative server** : The server itself handles the request and, if necessary, returns a response to the requesting client.
  - (ii) **Concurrent server** : The server does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request.
  - (iii) **Multithreaded server** : This is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request. This approach is followed in many UNIX systems.
- The thread or process that handles the request is responsible for returning a response to the requesting client.
- Another issue is how clients communicate with a server. Clients always submit requests to an endpoint, also known as a port, on the machine where the server is running. Each server listens to a different endpoint. How do clients know when a service is complete?
- One approach is to globally assign endpoints for well-known services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80.
- These endpoints have been assigned by the Internet Assigned Numbers Authority (IANA).
- The client just has to find the network address of the machine where the server is executing with assigned endpoints. There are many services that do not require a preassigned endpoint. For example, a time-of-day server may use an endpoint that is dynamically assigned to it by its local operating system.
- In that case, a client will first have to look up the endpoint. One solution is to have a special daemon running on each machine that runs servers.
- The daemon keeps track of the current endpoint of each service implemented by a co-located server. The daemon itself listens to a well-known endpoint.
- A client will first contact the daemon, request the endpoint, and then contact the specific server, as shown in Fig. 4.8.1(a).



(a) Client-to-server binding using a daemon **(b) Client-to-server binding using a superserver**

Fig. 4.8.1

- Endpoints are frequently associated with a specific service. However, installing each service on a separate server could be a waste of resources. For example, on a typical UNIX system, many servers run concurrently, with the majority of them passively waiting for a client request. Instead of keeping track of so many passive processes, it is typically more effective to have a single superserver listening to each end point connected with a certain service, as seen in Fig. 4.8.1(b).

4.8.2 Server Clusters

- A server cluster is just a group of machines connected by a network, each of which runs one or more servers.
- Server clusters are those in which the machines are linked via a local-area network, providing high bandwidth and low latency.

- In most cases, a server cluster is logically organized into three tiers, as shown in Fig. 4.8.2.

(1) The first tier consists of a (logical) switch that routes client requests. A switch of this type can vary greatly. For example, transport-layer switches receive incoming TCP connection requests and route them to one of the cluster's servers.

(2) Many server clusters, like any multitiered client-server architecture, include servers dedicated to application processing. These are typically servers running on high-performance hardware dedicated to providing compute power in cluster computing. However, in the case of enterprise server clusters, applications may only need to run on very low-end servers because compute power is not the constraint, but access to storage is.

(3) The third tier is made up of data-processing servers, specifically file and database servers. Again, depending on how the server cluster is used, these servers may be running specialized machines with high-speed disk access and massive server-side data caches.

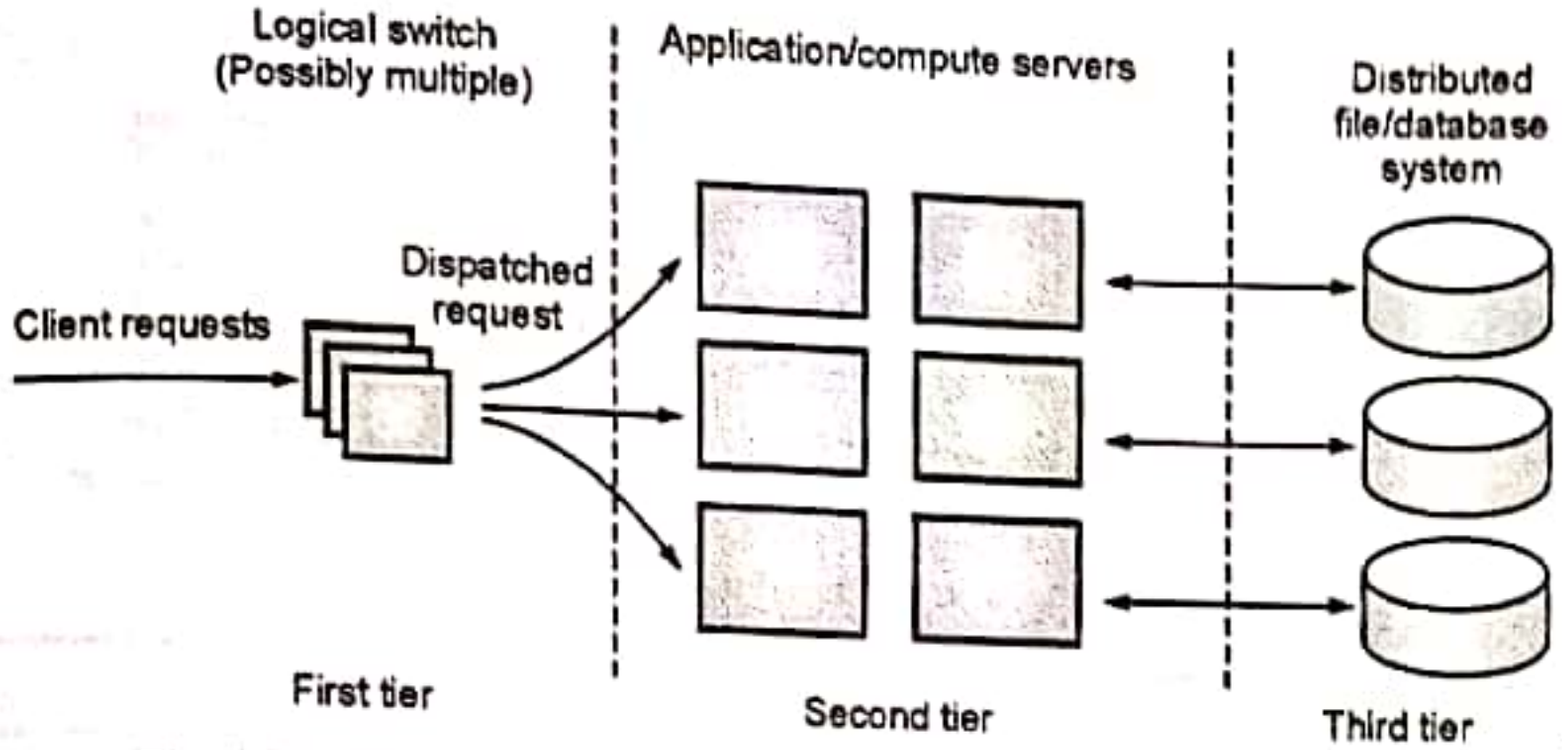


Fig. 4.8.2 : The general organization of a three-tiered server cluster

4.8.3 Distributed Servers

- The server clusters presented thus far are mostly statically configured. In these clusters, there is frequently a separate administration system that maintains track of available servers and provides this information to other machines as needed, such as the switch.
- As previously stated, most server clusters have a single access point. When that point fails, the cluster becomes unavailable. To overcome this potential issue, various access points with publicly accessible addresses can be supplied. For example, the Domain Name System (DNS) can output many addresses that all belong to the same host name.
- If one of the addresses fails, customers must try again. Furthermore, this does not address the issue of the need for static access points.
- Stability, such as a long-lived access point, is a desirable quality for both clients and servers. On the other hand, having a great degree of flexibility in configuring a server cluster, including the switch, is also desired.
- This discovery has resulted in the construction of a distributed server that is effectively nothing more than a possibly dynamically changing set of machines with possibly varied access points, but which looks to the outside world as a single, powerful system.
- The main concept behind a distributed server is that clients benefit from a reliable, high-performance, and stable server. These characteristics are frequently offered by high-end mainframes, some of which have a renowned mean time between failure of more than 40 years.
- However, by transparently grouping simpler computers into a cluster and not relying on the availability of a single machine, it may be feasible to attain a higher level of stability than by each component separately. As in the case of a

collaborative distributed system, such a cluster might be dynamically configured from end-user machines.

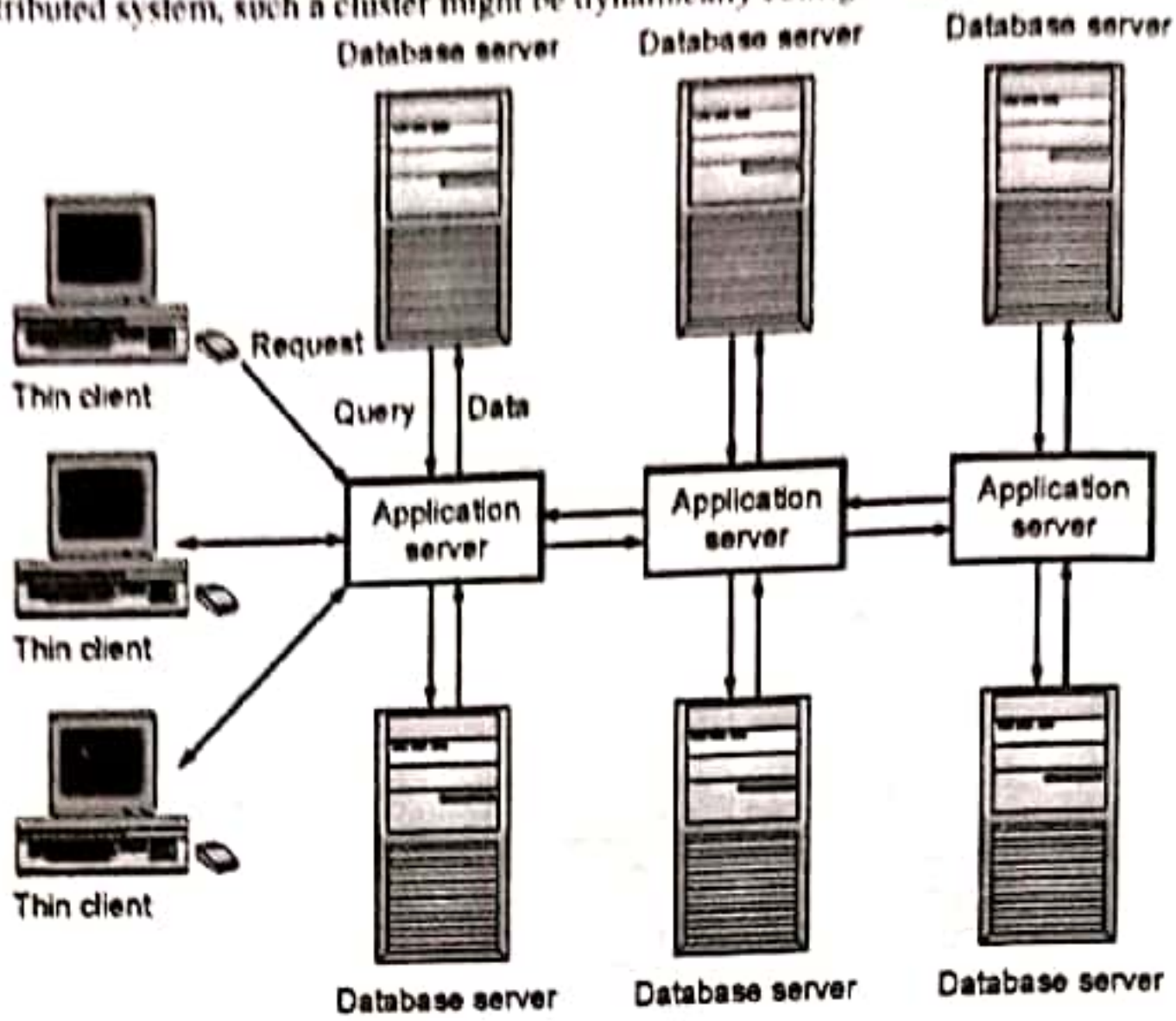


Fig. 4.8.3 : Distributed Servers

4.9 CODE MIGRATION

UQ. Write a short note on Code Migration. **(MU - Dec. 16)**

UQ. Describe code migration issues in detail. **(MU - May 17, Dec. 19)**

UQ. What is the need for Code Migration? Explain the role of Process to resource and resource to Machine binding in Code Migration. **(MU - Dec. 18, May 22)**

- So far, we've mostly focused on distributed systems where communication is confined to data passing. However, there are instances when passing programs, even while they are being run, simplifies the architecture of a distributed system.
- In this section, we'll go through exactly what code migration is. We begin by discussing various techniques for code migration, followed by a discussion of how to deal with the local resources that a migrating program uses.
- Migrating code on heterogeneous systems is a particularly difficult challenge, which is also explored.

4.9.1 Approaches to Code Migration

Let's first analyze why code migration might be advantageous before looking at the various types of migration.

(i) Traditionally, moving a complete process from one machine to another was the method used for code migration in distributed systems.

(ii) There had better be a strong reason for moving a running process to another machine because doing so is an expensive and difficult procedure. It has always been for performance. The underlying principle is that by moving operations from machines that are substantially loaded to machines that are lightly loaded, total system performance can be enhanced. Although other performance metrics are also utilized, the load is frequently stated in terms of CPU queue length or CPU utilization.

(iii) Because of the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration frequently relies on qualitative reasoning rather than mathematical models. Think about a client-server system where the server controls a large database, for instance. If a client program needs to do numerous database operations involving huge amounts of data, it could be preferable to ship some of the client applications to the server and transmit only the results over the network.

(iv) It is sometimes possible to avoid having to send a significant number of small messages across the network by processing the form on the client side and sending only the completed form to the server. As a result, the server spends less time on form processing and communication, and the client experiences better performance.

(v) Support for code migration can also boost performance by using parallelism, but without the usual complexities of parallel programming. Web-based information retrieval is a classic example. It is relatively straightforward to implement a search query as a little mobile program that hops from site to site. By creating multiple copies of the program and delivering them to separate locations, we may be able to get a linear speedup compared to using a single instance of the program.

(vi) In addition to enhancing performance, there are additional reasons to facilitate code migration. The most essential characteristic is flexibility. The conventional method for developing distributed applications entails dividing the application into parts and determining in advance where each part will be run. This strategy, for instance, has led to the development of several multitiered client-server programs. However, if code can move between different machines, it becomes possible to dynamically configure distributed systems.

(vii) In some cases, we need to download certain program to achieve the client server configuration. Here, code migration plays an important role as it should be compatible with both ends. Moreover, the machine should support the migrated codes.

4.9.2 Models for Code Migration

Although the name code migration suggests that we merely transfer code between machines, it actually encompasses a considerably broader scope. Communication in distributed systems has traditionally focused on the exchange of data between processes. In the broadest sense, code migration is the process of moving programs between machines with the purpose of executing them on the target. In certain instances, such as process migration, the status of a program's execution, pending signals, and other aspects of the environment must also be relocated.

- To gain a deeper knowledge of the various models for code migration, we employ the methodology outlined by Fugetta et al. A process comprises three stages within this framework.

  (1) The **code segment** is the section containing the collection of instructions that comprise the currently executing program.

  (2) The **resource segment** provides references to external resources required by the process, such as files, printers, devices, and so on.

  (3) An **execution segment** is used to store the current execution state of a process, which is comprised of private data, the stack, and the program counter.

- The various alternatives for code migration are summarized in Fig. 4.9.1.

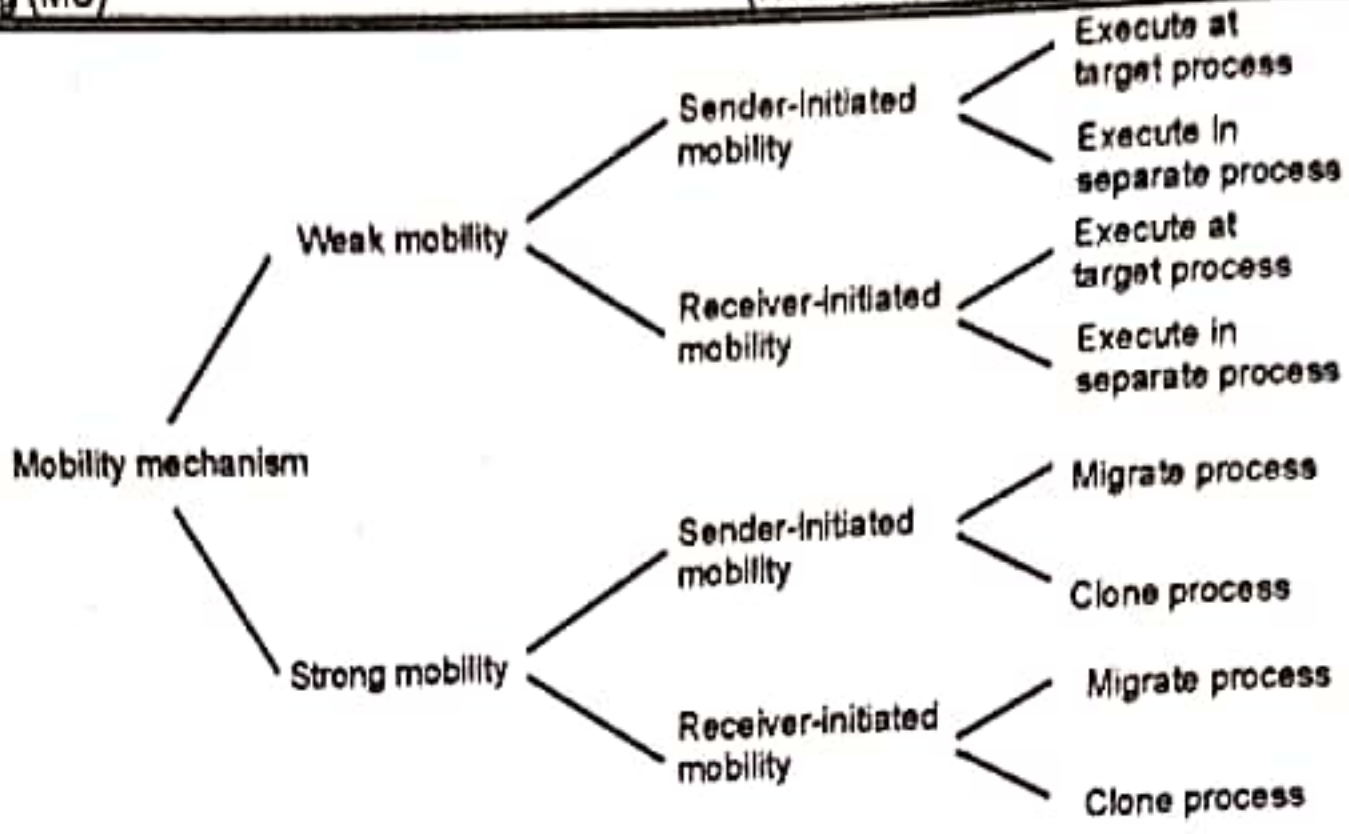


Fig. 4.9.1 : Alternatives for code migration

(i) **Weak Mobility :** In the weak mobility model, only the code segment and maybe initialization data can be transferred. Weak mobility is characterized by the fact that a transferred program is always restarted from its initial state. This is the case with Java applets, for example. This method has the benefit of being simple. Weak mobility needs merely that the target computer can execute the code, which boils down to the code being portable.

(ii) **Strong Mobility :** In systems with strong mobility, the execution segment can also be transferred, in contrast to systems with weak mobility. Strong mobility is characterized by the ability to pause a running process, move it to a different computer, and then resume execution from where it left off. Clearly, strong mobility is far more effective than weak mobility, but it is also considerably more difficult to implement.

(iii) **Sender-Initiated Mobility :** In sender-initiated migration, migration is begun on the computer where the code resides or is presently executing. Sender-initiated migration is typically performed when programs are uploaded to a compute server. Another example would be sending a search program over the Internet to a Web database server for that server to run the queries.

(iv) **Receiver-Initiated Mobility :** In receiver-initiated migration, the target machine initiates the migration of code. Java applets are one illustration of this methodology. Sender-initiated migration is typically more difficult to implement than receiver-initiated migration. Frequently, code migration occurs between a client and a server, with the client initiating the transfer. To securely upload code to a server, as is the case with sender-initiated migration, the client must typically have already registered and authenticated with that server.

4.9.3 Migration and Local Resources

- So far, just the code and execution segment migration has been considered. The resource section necessitates special consideration. What frequently makes code migration difficult is that the resource segment cannot always be transported along with the other segments without being altered.
- Assume a process has a reference to a certain TCP port through which it communicates with other (remote) processes. This type of reference is kept in its resource part.
- When the process transfers to a new location, it must surrender the port and request a new one at the new site. In other circumstances, moving a reference is not an issue. For example, an absolute URL reference to a file will stay valid regardless of the computer where the process that stores the URL lives.
- We identify three types of **process-to-resource bindings** to better understand the impact of code migration on the resource segment.

(1) The ***strongest binding*** occurs when a process refers to a resource by its identification. In that situation, the procedure requires only the referred resource and nothing else. When a process uses a URL to refer to a specific Web site or when it refers to an FTP server by using that server's Internet address, this is an example of binding by identifier. In the same way, references to local communication endpoints result in a binding by identifier.

(2) When simply the value of a resource is required, a ***weaker form*** of process-to-resource binding is used. In such an example, the process would not be affected if another resource provided the identical value. When software relies on standard libraries, such as those for programming in C or Java, this is an example of binding by value. Such libraries should always be available locally, but their actual position in the local file system may differ between locations. Not the specific files, but their content is critical for the effective execution of the operation.

(3) Finally, the ***weakest type of binding*** occurs when a process states that it just requires a specific type of resource. This type of binding is demonstrated by references to local devices like s monitors, printers, and so on.

- When migrating code, we frequently need to alter resource references but cannot change the type of process-to-resource binding. If and how a reference should be modified is determined by whether the resource can be moved to the target machine with the code.
- We must evaluate resource-to-machine bindings and identify the following instances :

(1) ***Unattached resources*** are simply transferred across workstations and are often (data) files related exclusively with the software being migrated.

(2) Moving or copying a ***fastened resource***, on the other hand, may be doable, but at a great expense. Local databases and full Web sites are common instances of fastened resources. Although such resources are not, in theory, dependent on their current computer, moving them to another environment is frequently impossible.

(3) Finally, ***fixed resources*** are inextricably linked to a particular system or environment and cannot be relocated. Local devices are frequently used as fixed resources. A local communication endpoint is another example of a fixed resource.

- Combining three different types of process-to-resource bindings and three different types of resource-to-machine bindings yields nine different possibilities to consider while migrating code. Fig. 4.9.2 depicts these nine combinations.

		Resource-to-machine binding		
		Unattached	**Fastened**	**Fixed**
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV, GR)	GR (or CP)	GR
	By type	RB (or MV, CP)	RB (or GR, CP)	RB (or GR)

Fig. 4.9.2 : Process-resource and resource-machine binding

Here, GR Establish a global systemwide reference

MV Move the resource

CP Copy the value of the resource

RB Rebind process to locally available resource

- Consider the possibilities when a process is bound to a resource via an identifier.
- When the resource is unattached, it is usually advisable to move it with the code.
- When the resource is shared by multiple processes, one alternative is to create a global reference, which can span machine boundaries. A URL is an example of such a reference.

- When a resource is fastened or fixed, it is ideal to construct a global reference.
- It is essential to understand that establishing a global reference may involve more than merely using URLs, and that using such a reference can be prohibitively expensive at times.
- The biggest disadvantage of this strategy is that communication with the migrated process may fail if the source machine fails. The alternate method is to modify the global reference of all processes that communicated with the migrating process and deliver messages to the new communication endpoint on the destination machine.

4.10 MIGRATION IN HETEROGENEOUS SYSTEMS

- So far, we've assumed implicitly that the migrated code can be simply executed on the target machine.
- When working with homogenous systems, this assumption is appropriate.
- In general, distributed systems are built on a diverse collection of platforms, each with its own operating system and machine architecture.
- In such systems, migration requires that each platform be supported, which means that the code segment can be executed on each platform, possibly after recompiling the original source.
- We must also guarantee that the execution section is accurately represented on each platform.
- When dealing with only weak mobility, problems can be mitigated to some extent. In that situation, there is virtually no runtime information that needs to be communicated between machines, therefore compiling the source code but generating multiple code segments, one for each probable target platform, suffices.
- The transfer of the execution segment is the key difficulty that must be solved in the case of strong mobility. The issue is that this segment is heavily reliant on the platform on which the process is run. In fact, only when the target machine has the same architecture and is running the exact same operating system as the source machine is it possible to migrate the execution segment without modifying it.
- Fig. 4.10.1 depicts a solution for procedural languages such as C and Java that works as follows. Code migration is limited to specific points in a program's execution. Migration can occur only when the next subroutine is executed. In C, a subroutine is a function, in Java, a method, and so on. The runtime system keeps its own copy of the program stack, but this copy is machine-independent. This copy is referred to as the migration stack. When a subroutine is invoked or when execution returns from a subroutine, the migration stack is updated.
- The runtime system marshals the data that has been pushed onto the stack since the last call when a function is called. These values indicate local variable values as well as parameter values for the freshly called subroutine. The marshalled data, together with an identification for the called procedure, are subsequently pushed onto the migration stack. In addition, the address where execution should proceed when the caller returns from the subroutine is pushed onto the migration stack in the form of a jump label.

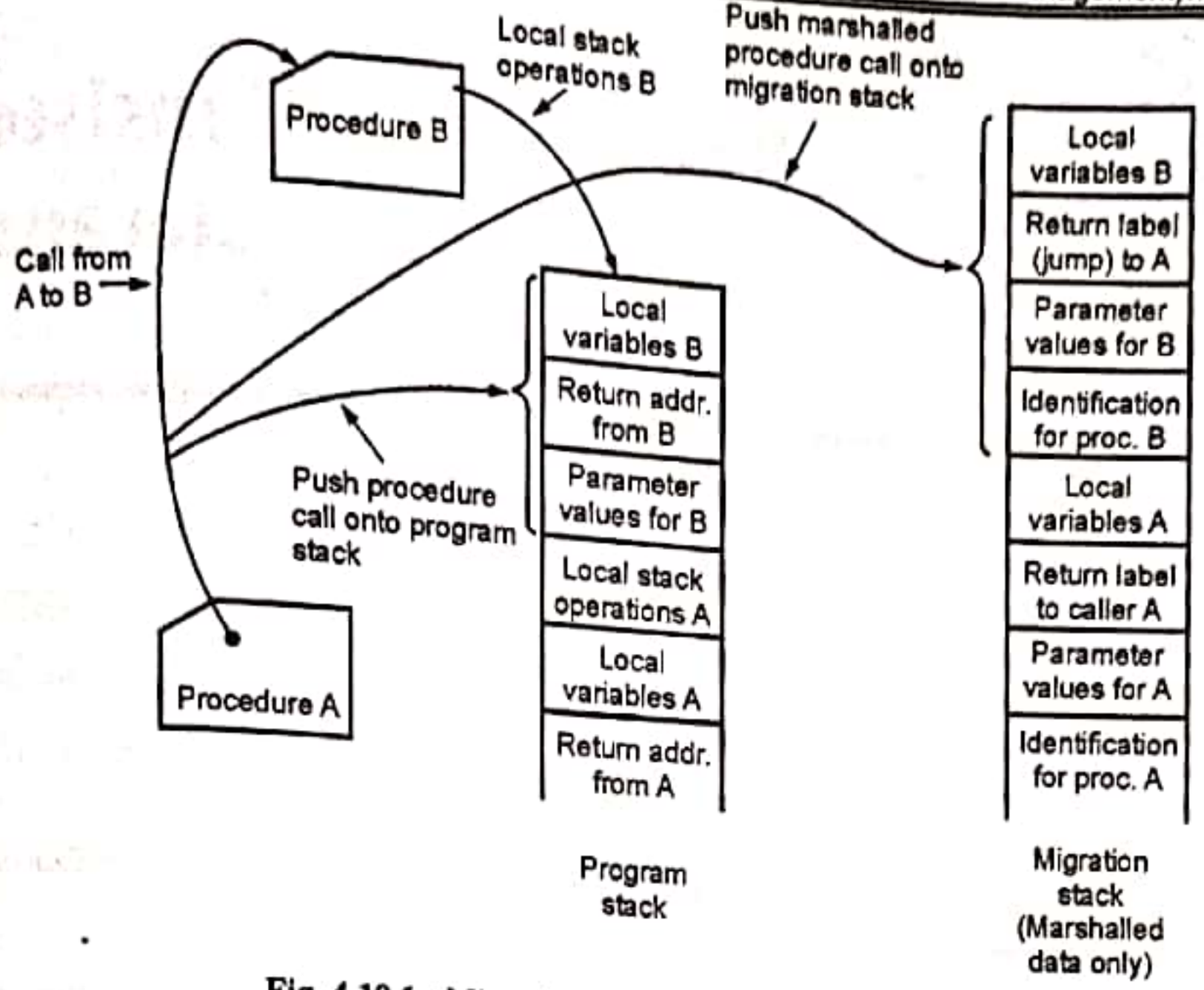


Fig. 4.10.1 : Migrations in Heterogenous System

- If code migration occurs when a subroutine is invoked, the runtime system marshals all global program-specific data that is part of the execution segment first. Machine-specific data, as well as the current stack, are ignored. The marshalled data, as well as the migration stack, are sent to the destination.
- Furthermore, the destination loads the code segment containing the binaries appropriate for its machine architecture and operating system. The execution segment's marshalled data is unmarshaled, and a new runtime stack is built by unmarshaling the migration stack. After that, execution can be resumed by simply entering the function that was called at the original location.

Descriptive Questions

Q.1 Explain desirable features of global scheduling algorithm.

Q.2 Explain the Load Balancing approach in a distributed system.

Q.3 Discuss the issues in load balancing algorithms.

Q.4 Explain with example the load sharing approach.

Q.5 Explain with example the task assignment approach.

Q.6 Differentiate between Process and Thread.

Q.7 Explain different multithreading models with neat diagrams.

Q.8 Explain the role of virtualization in distributed system.

Q.9 Discuss code migration problem in detail.

Q.10 What is the need for Code Migration? Explain the role of Process to resource and resource to Machine binding in Code Migration.

Chapter Ends...

❑❑❑

MODULE 5

CHAPTER 5

Replication, Consistency and Fault Tolerance

5.1 INTRODUCTION

- The replication of data in distributed systems is a significant problem. Usually, data replication is done to boost performance or reliability.
- Consistent replication is one of the main issues. Informally, this means that we must make sure that all copies are updated when one copy is; otherwise, the duplicates will no longer be identical.
- Here, the key inquiries are, "Why replication is useful and how scalability is related to it? What, exactly, consistency means".
- First, we focus on managing replicas, which considers both the placement of replica servers and the method through which content is transmitted to these servers.
- The second issue is how these replicas are kept consistent. Applications often require a high level of consistency. Informally, this means that updates should be quickly replicated between replicas.

5.2 REASONS FOR REPLICATION

Replication of data is done for two main reasons, in addition to performance and reliability :

(1) Replication of data improves system reliability

- If a file system has been replicated, switching to another replica may be all that is required to keep the system running if one replica fails. Furthermore, storing multiple copies allows for enhanced security against corrupted data.
- Consider the following scenario: a file is replicated three times, and every read and write operation is performed on each copy. We can protect ourselves from a single unsuccessful write operation by assuming the value returned by at least two copies is the actual value.

(2) Replication for Performance

(i) **Scaling in numbers :** Replication is crucial for performance when a distributed system needs to scale both in terms of users and in terms of geography. Scaling in numerical terms, for instance, occurs when a growing number of processes require access to data that is controlled by a single server. In this case, by duplicating the server and then splitting the workload, performance can be enhanced.

(ii) **Scaling geographically :** The basic idea is that by putting a copy of the data close to the process that uses them, the access time to the data is slashed. As a result, the process perceives an improvement in performance.

It could be challenging to assess the advantages of replication for performance. While a client process can see improved performance, it's also possible that additional network bandwidth is now used to keep all replicas current. However, the replication of data, sadly, comes at a cost.

The issues with replication are as follows :

(1) There may be a possibility of consistency issues brought on by multiple copies. Every time a copy is modified, it differs from the others. Therefore, in order to assure consistency, changes must be made to all copies. The cost of replication is determined by precisely when and how those modifications must be made.

(2) Cost of increased bandwidth required to maintain replication.

- To understand this problem, think about reducing the amount of time it takes to visit Web pages. Fetching a page from a distant Web server may occasionally even take a few seconds if certain precautions are not taken.

- Web browsers frequently save a duplicate of a previously retrieved Web page locally to increase performance (i.e., they cache a Web page).
- The browser immediately returns the local copy whenever a user requests that page again. The user rates the access time as excellent. The issue is that cached copies will become out-of-date if the page has been amended since changes will not have been transmitted to those copies.
- One way to avoid returning an outdated copy to the user is to prevent the browser from making local copies in the first place. This way, the server will take full control of replication. If there is no replica placed close to the user, this option can still result in slow access times.
- Another option is to allow the Web server to update or invalidate each cached copy, but this necessitates the server monitoring all caches and communicating with them. The server's overall performance could suffer as a result.

5.2.1 Replication as a Scaling Strategy

- Performance concerns are a common manifestation of scalability issues.
- By reducing access time, placing copies of the data close to the processes that need it can enhance performance and address scalability issues.
- The trade-off that would be necessary is that updating copies might use more network capacity. Consider a process P that updates local replica M times per second while accessing it N times per second. Assume that a local replica's prior iteration has been completely refreshed by an update.
- If N<<M, i.e. the access-to-update ratio, is very low, there will be many updated versions of the local replica that P would never access, making the network communication for those versions pointless. In this instance, it might have been preferable to avoid installing a local replica near P altogether or to use a different method of updating the replica.
- Keeping numerous copies consistent may itself be subject to major scaling issues, which is a more serious issue.
- It makes sense that a group of copies would be consistent if they were consistently the same. This implies that any read operation will always have the same outcome.
- As a result, no matter at which copy a subsequent operation is commenced or executed, the update should be propagated to all copies when an update operation is done on one copy.
- This type of consistency is sometimes informally referred to as *tight consistency* or synchronous replication.
- The fundamental concept is that an update is carried out across all copies as a single atomic operation, or transaction. All replicas need to be brought into sync. This implies that the precise timing of a local update must first be agreed upon by all replicas.
- For instance, using Lamport timestamps, replicas may need to decide on a global ordering of operations, or they may need to let a coordinator choose such an order.
- Although replication and caching are often used as scaling techniques, they have the following drawbacks :
  (1) Keeping replicas up-to-date needs networks use.
  (2) An atomic update is required (transaction)
  (3) Synchronization of replicas is required (time-consuming)
- In *loose consistency*, copies are not always the same everywhere.

5.3 CONSISTENCY MODELS

UQ. Explain the difference between Client-Centric and Data-Centric Consistency Models. Explain one model of each. (MU - May 16)

UQ. Explain the need of client-centric consistency models as compared to data-centric consistency model. Explain any client-centric consistency model. (MU - Dec. 16)

UQ. Discuss and differentiate various client-centric consistency models. (MU - May 17)

UQ. What are different data-centric consistency models? (MU - May 18)

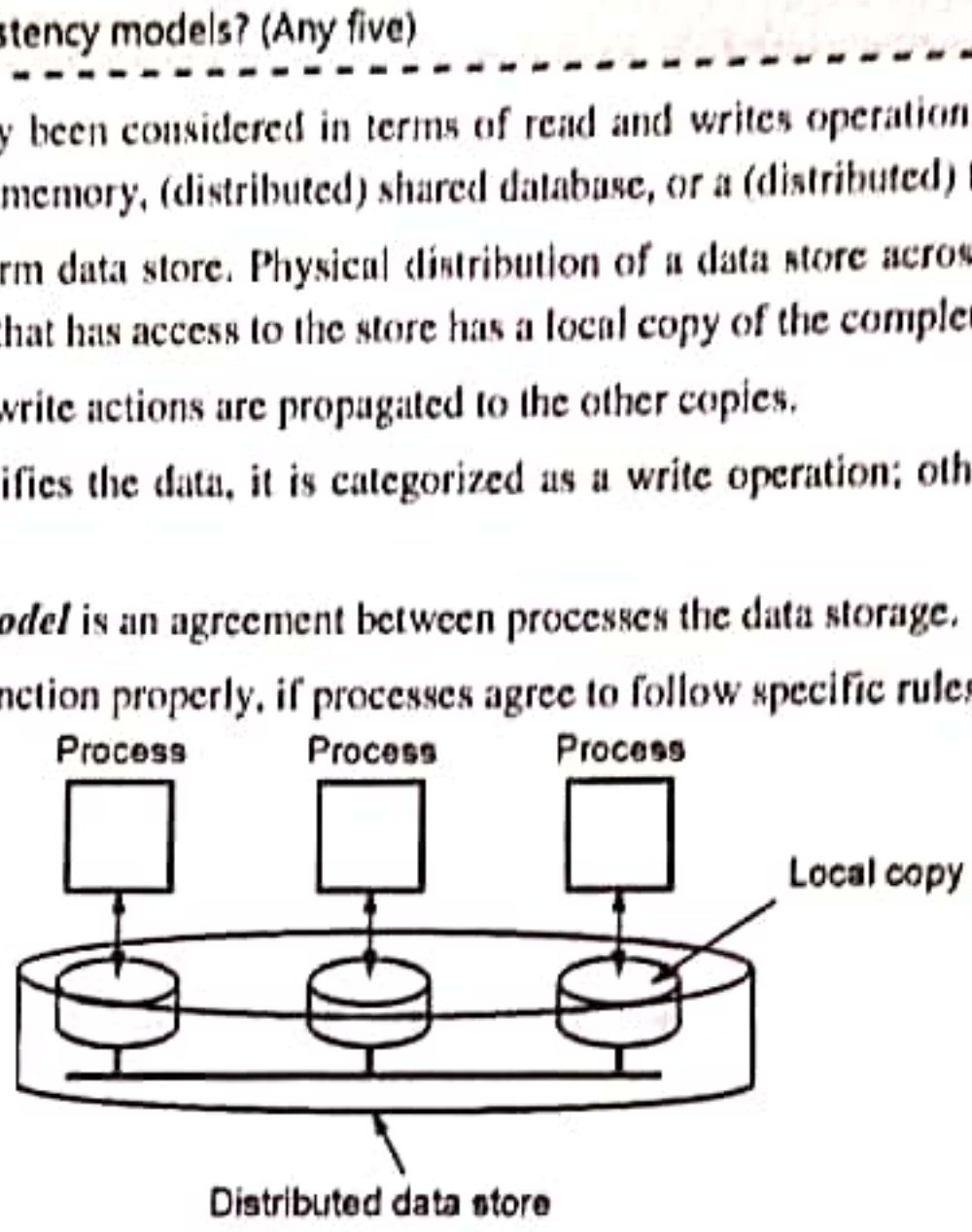
UQ. Discuss and differentiate various client-centric consistency models by providing suitable example application scenarios. (MU - Dec. 18)

UQ. Clearly explain how Monotonic Read Consistency Model is different from Read Your Write Consistency Model. Support your answer with suitable example application scenario where each of them can be distinctly used. (MU - May 19)

UQ. Discuss the role of consistency in the distributed system. What is the need of client-centric consistency models? Explain any two data-centric consistency models. (MU - Dec. 19)

UQ. What are different consistency models? (Any five) (MU - May 22)

- Consistency has traditionally been considered in terms of read and writes operations on shared data that is accessible through (distributed) shared memory, (distributed) shared database, or a (distributed) file system.
- Here, we use the broader term data store. Physical distribution of a data store across various devices is possible. It is expected that every process that has access to the store has a local copy of the complete store on hand.
- As seen below in Fig.5.3.1, write actions are propagated to the other copies.
- When a data operation modifies the data, it is categorized as a write operation; otherwise, it is categorized as a read operation.
- In essence, a ***Consistency Model*** is an agreement between processes the data storage.
- It states that the store will function properly, if processes agree to follow specific rules.



(IE1)**Fig. 5.3.1: The general organization of a logical data store, physically distributed and replicated across multiple processes**

- A process typically expects a read operation on a data item to deliver a value that displays the outcomes of the most recent write action on that data.
- It is challenging to pinpoint precisely which write operation was the last in the absence of a global clock. We must instead offer different definitions, which results in a variety of consistency models.
- The values that a read operation on a data item can return are essentially constrained by each model.
- Consistency models can be classified into the following two types :

(1) Data-Centric Consistency models

- These models define how data updates are propagated across the replicas to keep them consistent.
- Examples of Data-Centric models are *Strict consistency, sequential consistency (Linearizability), causal consistency, FIFO, Weak, Release, and Entry.*

(2) Client Centric Consistency models

- These models assume that clients connect to different replicas at different times.
- The model ensures that whenever a client connects to a replica, the replica is brought up to date with the replica that the client had previously connected to.
- Examples of Client Centric models are *Eventual Consistency, Monotonic Reads, Monotonic Writes, Read Your Writes, Writes Follow Reads.*

5.3.1 Data-Centric Consistency models

- Consistency issues resulting from data replication are difficult to address effectively in a general way.
- The only way we can possibly find effective solutions is if we relax our strictness. Sadly, there are no standard guidelines for losing consistency either. Exactly what can be tolerated depends much on the application.
- Applications can declare what inconsistencies they can tolerate in a variety of ways.
- We can adopt a broad perspective by defining inconsistencies along three separate axes :
  (1) Deviations in numerical values across replicas.
  (2) Deviations in staleness between replicas.
  (3) Deviations in terms of the ordering of update operations.
- Continuous consistency ranges are said to arise as a result of these variances.

(1) Deviations in numerical values

- Applications, where the data has a numerical interpretation, can measure inconsistency in terms of numerical deviations.
- One simple example is the replication of data holding stock market prices.
  - In this situation, an application may specify that the absolute numerical variation of the two copies should not exceed $0.02.
  - An alternative would be to specify a relative numerical deviation, e.g., that the difference between any two copies should not exceed 0.5%.
  - In both scenarios, it would be clear that replicas would still be regarded as being mutually consistent even if a stock rose (and one of the replicas was updated right away).
- The number of updates that have been made to a particular replica but have not yet been observed by other replicas can also be used to understand numerical deviation.
- A Web cache, for instance, might not have observed a batch of actions performed by a Web server. In this instance, the corresponding deviation from the value is also referred to as its **weight.**

(2) Deviations in staleness

- Staleness deviations are related to the most recent update to a replica. For some applications, providing outdated data from a replica is acceptable if it is not excessively outdated.

- For instance, during a period of time, let's say a few hours, weather reports often remain very accurate. When this happens, the primary server might get timely updates but choose to propagate these updates to the replicas only occasionally.

(3) Deviations in terms of the ordering of update operations

- Finally, some application types permit varying update ordering across different replicas if the differences stay within a certain range. For example, these updates could be viewed as being tentatively applied to a local copy while awaiting consensus from all replicas.
- Because of this, some adjustments might not take effect right away and could need to be implemented in a different order.
- Ordering deviations are substantially more difficult to understand conceptually than the other two consistency metrics.

The different types of Data-Centric Consistency Models are as follows :

(1) Strict Consistency	(2) Sequential Consistency
(3) Causal Consistency	(4) FIFO Consistency
(5) Weak Consistency	(6) Release Consistency
(7) Entry Consistency	

(1) Strict Consistency

- Any read on a data item 'x' returns a value corresponding to the result of the most recent write on 'x' (regardless of where the write occurred).
- Two operations in the same time interval are said to be in conflict if they operate on the same data and one of them is a write operation.
- With Strict Consistency, the distributed system maintains absolute global temporal order, and all writes are instantly visible to all processes.
- This is the "Holy Grail" consistency model - not easy in the real world, and very impossible in a DS.
- **Example :**

  Let's start with :

  - $W_i(x)a$ -write data item x by method P_i with the value a
  - $R_i(x)b$ - read by process P_i from data item x returning the value b

  Consider that

  - The time axis is depicted horizontally, with the passage of time accelerating from left to right.
  - Each data point starts out as NIL.

(a) A strictly consistent data-store **(b) A data-store that is not strictly consistent**

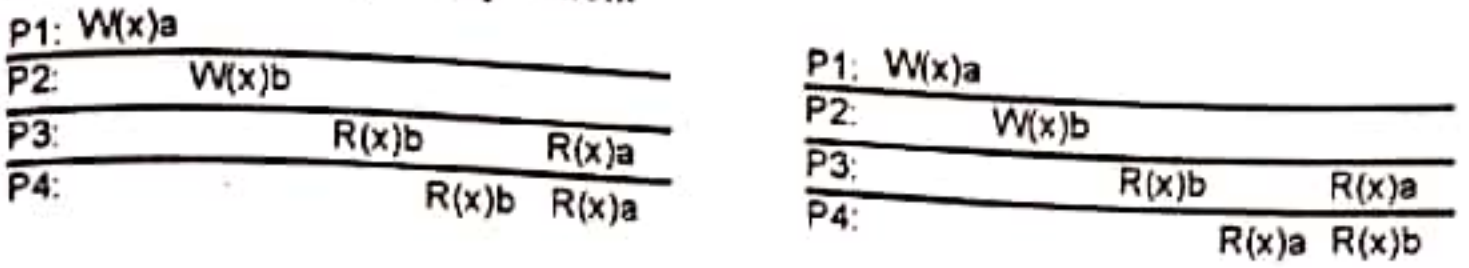
(1E2)**Fig. 5.3.2**

- P1 changes the value of data item x to a by writing to it.
- A local copy of the data store belonging to P1 is the first place where Operation $W_1(x)a$ is carried out. From there, it is propagated to the other local copies.

- After reading the value NIL in the future, P2 eventually reads a (from its local copy of the store).
- NOTE: It is acceptable if the propagation of the update of x to P2 took some time.

(2) Sequential Consistency

- Compared to strict consistency, Sequential Consistency is a weaker consistency model.
- ***The result of any execution is the same as if the read and write operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.***
- Operation interleaving is identical for all processes.
- No reference is given to the timing of the operation.



(a) A sequentially consistent data store **(b) A data store that is not sequentially consistent**

(IEJ)**Fig. 5.3.3**

- In Fig.5.3.3(a):
  - (i) Process P1 first performs W(x)a to x.
  - (ii) Later (in absolute time), process P2 performs a write operation, by setting the value of x to b.
  - (iii) Both P3 and P4 first read value b, and later value a.
  - (iv) Write operation of process P2 appears to have taken place before that of P1.
- In Fig.5.3.3(b):
  - (i) Violation of sequential consistency is seen because not all processes see the same interleaving of write operations.
  - (ii) To process P3, it appears as if the data item has first been changed to b, and later to a.
  - (iii) But, P4 will conclude that the final value is b.
- An adaptation of the Sequential Consistency models is the **Linearizability** concept.
- Linearizability is weaker than strict consistency but stronger than sequential consistency.
- The result of any execution is the same as if the read and write operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.
- In addition, if $ts_{OP1}(x) < ts_{OP2}(y)$, then operation OP1(x) should precede OP2(y) in this sequence.
- Operations receive a timestamp using a global clock, but with finite precision.
- Program order must be maintained. Data coherence must be respected.

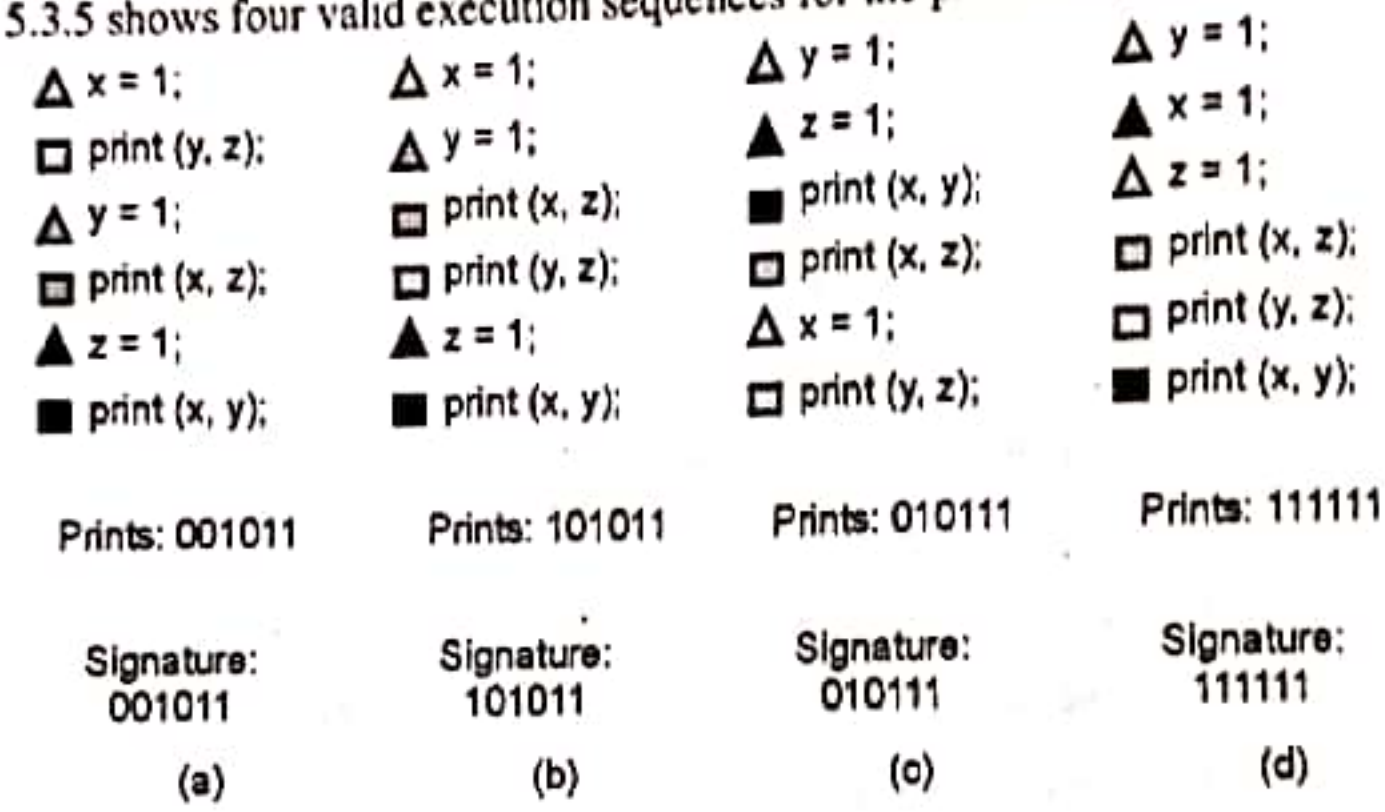
Example

- Consider three concurrently executing processes P1, P2, and P3.
- Three integer variables x, y, and z are stored in a (possibly distributed) shared sequentially consistent data store.
- Assume that each variable is initialized to 0.
- An assignment corresponds to a write operation, whereas a print statement corresponds to a simultaneous read operation of its two arguments.
- All statements are assumed to be indivisible.

Process P1	Process P2	Process P3	
		z = 1;	write
x = 1;	y = 1;	print (x, y);	read
print (y, z).	print (x, z);		

Fig. 5.3.4 : Three Concurrently Executing Processes

- Various interleaved execution sequences are possible.
- With six independent statements, there are potentially (6!) i.e., 720 possible execution sequences
- Consider the 120 (5!) sequences that begin with x =1.
  - Half of these have print (x,z) before y =1 and thus violate program order.
  - Half have print (x,y) before z =1 and violate program order.
  - Only 1/4 of the 120 sequences, or 30, are valid.
  - Another 30 valid sequences are possible starting with y = 1
  - Another 30 can begin with z =1, for a total of 90 valid execution sequences.
- The following Fig. 5.3.5 shows four valid execution sequences for the processes, where the vertical axis is time.



(IE4)**Fig. 5.3.5 : Four valid execution sequences for the processes. The vertical axis is time**

- According to the scenario depicted in Fig.5.3.5 (a)
  - (i) The three processes are in order - P1, P2, P3.
  - (ii) Each of the three processes prints two variables.
  - (iii) Since the only values each variable can take on are the initial value (0), or the assigned value (1), each process produces a 2-bit string.
  - (iv) The numbers after Prints are the actual outputs that appear on the output device.
  - (v) Output concatenation of P1, P2, and P3 in sequence produces a 6-bit string that characterizes a particular interleaving of statements.
  - (vi) This is the string listed as the Signature.
- Similarly, Fig.5.3.5 (b), (c), and (d) also represent valid execution sequences.
- Examples of possible output :
  - (i) 000000 is not permitted since it implies that the Print statements ran before the assignment statements, violating the requirement that statements are executed in program order.
  - (ii) 001001 is not permitted because
    - First two bits 00 - y and z were both 0 when P1 did its printing.

- This situation occurs only when P1 executes both statements before P2 or P3 start.
- Second two bits 10 - P2 must run after P1 has started but before P3 has started.
- Third two bits, 01- P3 must complete before P1 starts, but P1 execute first.
-

(3) Causal Consistency

- Two events are potentially causally related when there is a read followed by a write.
- Operations not causally related are said to be concurrent.
- Writes that are potentially causally related must be seen by all processes in the same order.
- Concurrent writes (i.e., writes that are NOT causally related) may be seen in a different order by different processes.

Examples :

(i) Consider an interaction through a distributed shared database.

(ii) Process P1 writes data item x.

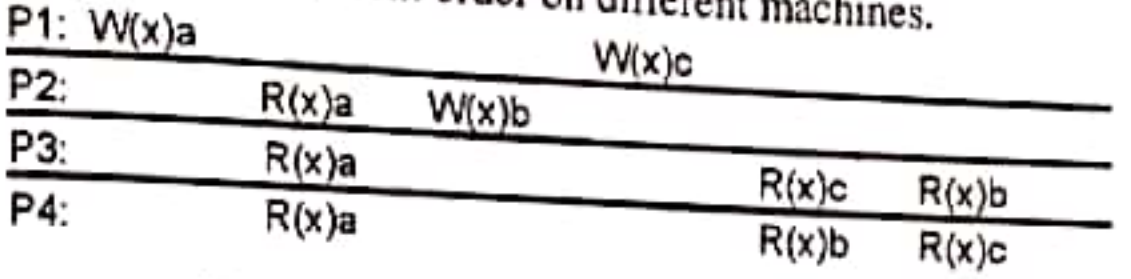
(iii) Then P2 reads x and writes y.

(iv) Reading of x and writing of y are potentially causally related because the computation of y may have depended on the value of x as read by P2 (i.e., the value written by P1).

- Conversely, if two processes spontaneously and simultaneously write two different data items, these are not causally related. Operations that are not causally related are said to be concurrent.
- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

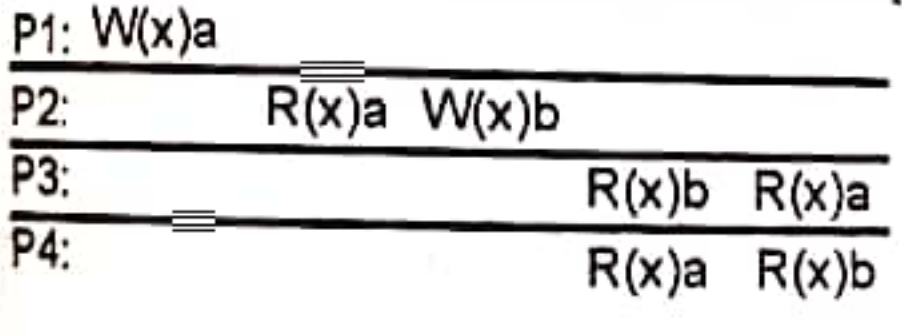
(i) Writes that are potentially causally related must be seen by all processes in the same order.

(ii) Concurrent writes may be seen in a different order on different machines.

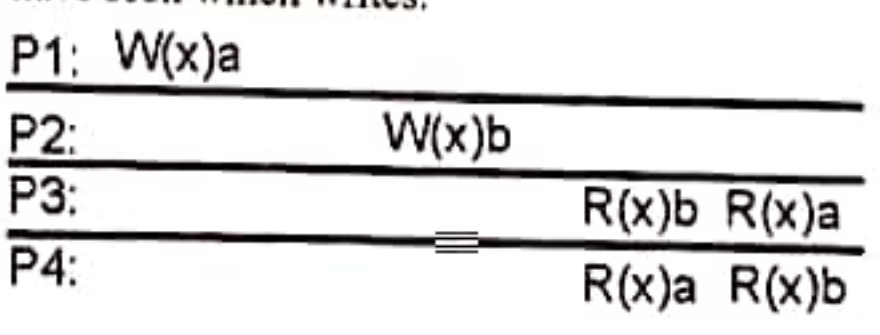


(1E5)**Fig. 5.3.6 : Causally Consistent Sequence**

- This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.
- Note that the writes $W_2(x)b$ and $W_1(x)c$ are concurrent.
- Causal consistency requires keeping tracks of which processes have seen which writes.



(a) A violation of a causally-consistent store



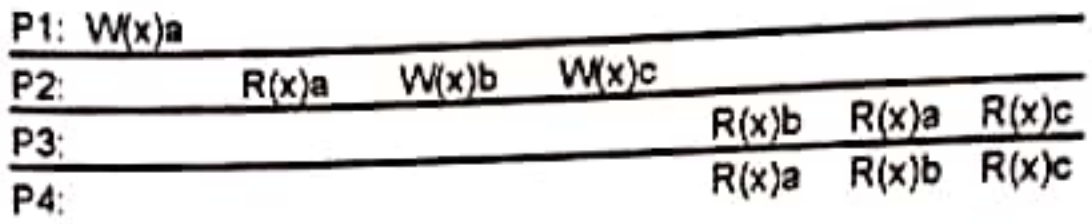
(b) A correct sequence of events in a causally-consistent store

(1E6)**Fig. 5.3.7**

- In Fig. 5.3.7(a), $W_2(x)b$ may be related to $W_1(x)a$ because the b may be a result of a computation involving the value read by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order.
- In Fig.5.3.7(b), $W_1(x)a$ and $W_2(x)b$ are concurrent and do not need to be globally ordered. This situation would not be acceptable by sequentially consistent store.

(4) FIFO Consistency

- Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
- It is also called "PRAM Consistency" – Pipelined RAM.
- It is easy to implement.
- There are no guarantees about the order in which different processes see writes – except that two or more writes from a single process must be seen in order.



(1E7)**Fig. 5.3.8 : A Valid Sequence of FIFO Consistency Events**

- The above Fig.5.3.8 depictsa sequence of events that holds valid for FIFO consistency but invalid for causal consistency.
- Note that none of the consistency models given so far would allow this sequence of events.

(5) Weak Consistency

- This model says that not all applications need to see all writes, let alone see them in the same order.
- This leads to inconsistent behavior (which is primarily designed to work with distributed critical sections).
- In this architecture, the idea of a synchronization variable is introduced. This concept is utilized to update all copies of the data store.
- **Properties of Weak Consistency**
  - (i) Accesses to synchronization variables linked to a data storage are sequentially consistent, which is a weak consistency.
  - (ii) Before performing any operations on a synchronization variable, all previous writes must have been finished everywhere.
  - (iii) Until all preceding operations to synchronization variables have been completed, no read or write activities on data items are permitted.
- What this means is that by synchronizing
- A process has the ability to force the recently written value to all other replicas.
- a process may be certain that it is receiving the most current value written before it reads.
- . The weak consistency models, as opposed to individual reads and writes, enforce consistency on a collection of activities (as is the case with strict, sequential, causal, and FIFO consistency).
- **The basic idea**
  - (i) It doesn't matter if other processes can immediately detect reads and writes of a series of activities.
  - (ii) You merely want to be aware of the series' overall impact.
- Convention
  - (i) S means access to synchronization variable
  - (ii) According to convention, a process should acquire the necessary synchronisation variables as soon as it enters its critical section and should release them as soon as it exits

Critical Section

- Code that accesses a shared resource (data structure or device) that cannot be used concurrently by more than one thread of execution is considered to be in a critical section.

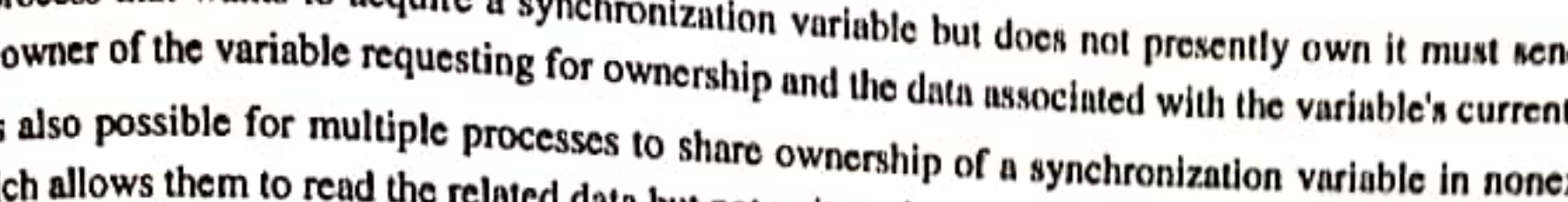
Synchronization variables

- (a) They are the primitives for synchronization that are used to synchronize the execution of processes depending on asynchronous events.
- (b) Synchronization variables are points where one or more processes can block until an event happens once, they have been assigned.
- (c) The processes can then either be unblocked individually or collectively.
- (d) Every synchronization variable has an active owner, which is the process that most recently obtained it.
  - The owner is permitted to repeatedly enter and leave the critical section without having to transmit any messages across the network.
  - A process that wants to acquire a synchronization variable but does not presently own it must send a message to the owner of the variable requesting for ownership and the data associated with the variable's current values.
  - It is also possible for multiple processes to share ownership of a synchronization variable in nonexclusive mode, which allows them to read the related data but not write to it.
  - The following criteria must be met :

    An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.

    Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.

    After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.



(a) A valid sequence of events for weak consistency **(b) An invalid sequence for weak consistency**

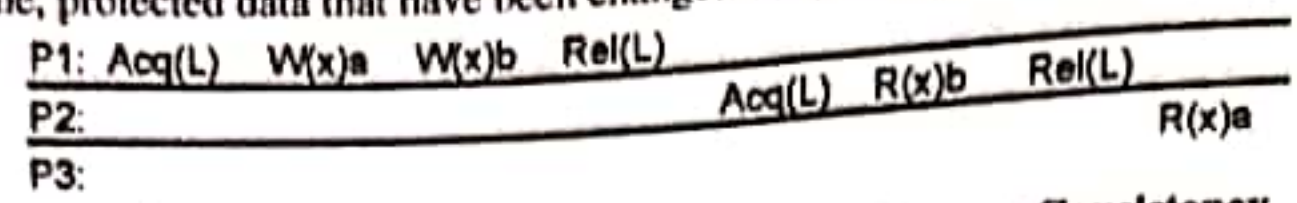
(IE8)Fig.5.3.9

- Fig. 5.3.9 (a) depicts a valid sequence of events for weak consistency because P2 and P3 have yet to synchronize, so there are no guarantees about the value in 'x'.
- Fig. 5.3.9 (b) depicts an invalid sequence of events for weak consistency because P2 has synchronized, so it cannot read 'a' from 'x', it should be getting 'b'.

(6) Release Consistency

- If it is possible to know the difference between entering a critical region or leaving it, a more efficient implementation might be possible. To do that, two kinds of synchronization variables are needed.
  - **acquire** operation to tell that a critical region is being entered.
  - **release** operation when a critical region is to be exited.
- When a process does an acquire, the data-store will ensure that all the local copies of the protected data are brought up to date to be consistent with the remote ones if needs be.

- When a release is done, protected data that have been changed are propagated out to the local copies of the data-store.



(1E9)**Fig. 5.3.10 : A Valid Event Sequence for Release Consistency**

- Fig. 5.3.10 depicts a valid event sequence for release consistency.
- Process P3 has not performed an *acquire*, so there are no guarantees that the read of 'x' is consistent.
- The data-store is simply not obligated to provide the correct answer.
- P2 does perform an *acquire*, so it's read of 'x' is consistent.
- **Rules for Release Consistency**
  - If a distributed data store complies with the guidelines below, it is "Release Consistent":
    - (1) All prior acquires made by the process must have been successful before a read or write operation is executed on shared data.
    - (2) All previous readings and writes by the process must be finished before a release may be carried out.
    - (3) Synchronization variable accesses are FIFO consistent (sequential consistency is not required).

▶ (7) Entry Consistency

Entry consistency also uses Acquire and release. However, the conditions for Entry consistency are as follows:

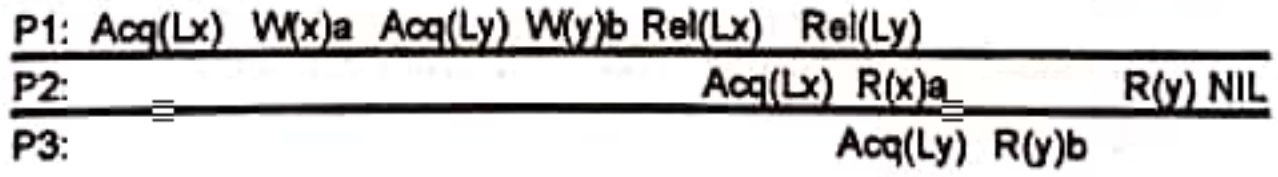
(i) It is not possible to perform an acquire access to a synchronisation variable with respect to a process until all updates to the guarded shared data have been completed with respect to that process.

(ii) No other process may hold a synchronisation variable, not even in nonexclusive mode, before an exclusive mode access by that process is allowed to perform with respect to that process.

(iii) After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has been performed with respect to that variable's owner.

Thus,

- At an acquire stage, all remote updates to protected data must be made current.
- Before writing to a data item, A process must confirm that no other process is attempting to write to that data item at the same time
- Locks are linked to specific data pieces rather than the full data store.
- *In a situation where there are numerous execution threads, a lock is a synchronization method for enforcing constraints on access to a resource.*



(1E10)**Fig. 5.3.11 : Valid Entry Consistency Model Sequence**

- In Fig. 5.3.11, P1 does an acquire for x, changes x once, after which it also does an acquire for y.
- Process P2 does an acquire for x but not for y, so that it will read value a for x, but may read NIL for y.
- Because process P3 first does an acquire for y, it will read the value b when y is released by P1.

Note : *P2's read on 'y' returns NIL* as no locks have been requested.

5.3.1(A) Summary of Data Centric Consistency Models

Consistency	Description: Consistency models that do not use synchronization operations.
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.
Consistency	**Description: Consistency models that do use synchronization operations.**
Weak	Shared data can be counted on to be consistent only after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

5.3.2 Client-Centric Consistency Models

5.3.2 (A) Need of Client-centric Consistency Models

- The consistency models discussed in the previous section focused on creating a system-wide consistent representation of a data store.
- It is crucial to provide consistency in the face of such parallelism since it is possible for concurrent processes to update the data store at the same time.
- Distributed systems must be able to manage concurrent operations on shared data while preserving sequential consistency.
- Sequential consistency may, for performance reasons, only be ensured when processes make use of synchronisation tools like transactions or locks.
- We examine a particular subset of distributed data repositories. The data stores we take into consideration are characterized by the absence of simultaneous updates or, if they do occur, by the ease with which they can be resolved.
- Most operations involve reading data.
- Thus, by introducing special client-centric consistency models, it turns out that many inconsistencies can be hidden in a relatively cheap way.

5.3.2(B) Types of Client-centric Consistency Models

(1) Eventual Consistency

- An eventual consistency is a weak consistency model.
- It lacks in simultaneous updates.
- When updates occur, they can easily be resolved.
- Most operations involve reading data.
- The eventual consistency model states that, when no updates occur for a long period of time, eventually all updates will propagate through the system and all the replicas will be consistent.

- Eventual consistency is inexpensive to implement.
- Requirements for Eventual Consistency are :
  (i) Few read/write conflicts (ii) No write/write conflicts
  (iii) Clients can accept temporary inconsistency
- **Example :** WWW

(2) Client-centric consistency based on work of Bayou.

- Bayou is a database system developed for mobile computing, where it is assumed that network connectivity is unreliable and subject to various performance problems.
- Wireless networks and networks that span large areas, such as the Internet, fall into this category.
- Bayou distinguishes four different consistency models:
  (i) Monotonic Reads (ii) Monotonic Writes
  (iii) Read your Writes (iv) Writes follow Reads
- **Notation:**
  (i) $x_i[t]$ denotes the version of data item x at local copy L_i at time t.
  (ii) WS $x_i[t]$ is the set of write operations at L_i that lead to version x_i of x (at time t);
  (iii) If operations in WS $x_i[t_1]$ have also been performed at local copy L_j at a later time t2, we write WS ($x_i[t_1]$, $x_j[t_2]$).
  (iv) If the ordering of operations or the timing is clear from the context, the time index will be omitted.

(a) Monotonic Reads

- According to monotonic-read consistency, if a process has seen a value of x at time t, it will never view an older version of x later.
- It states that, *if a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.*
- Example: Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

(a) A monotonic-read consistent data store **(b) A data store that does not provide monotonic reads**

(1E11)**Fig. 5.3.12**

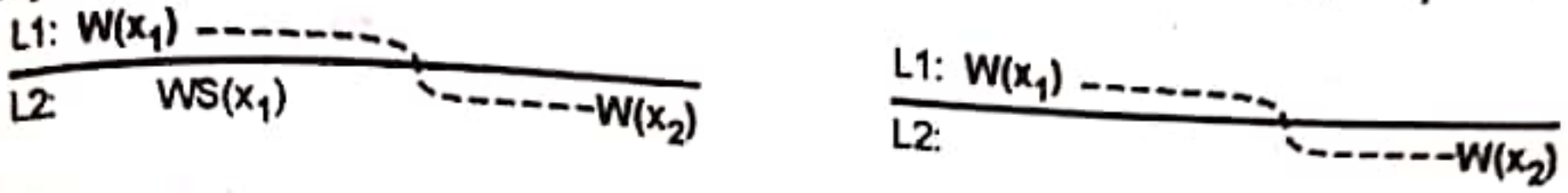
- In Fig. 5.3.12, the read operations are performed by a single process P at two different local copies of the same data store.
- On the vertical axis, two different local copies of the data store are shown as L1 and L2.
- Time is shown along the horizontal axis.
- Operations carried out by a single process P in boldface are connected by a dashed line representing the order in which they are carried out.
- In Fig. 5.3.12 (a), Process P first performs a read operation on x at L1, returning the value of x_1 (at that time).This value results from the write operations in WS (x_1) performed at L1.Later, P performs a read operation on x at L2, shown as R (x_2).To guarantee monotonic-read consistency, all operations in WS (x_1) should have been propagated

to L2 before the second read operation takes place.

- In Fig. 5.3.12 (b),monotonic-read consistency is not guaranteed. After process P has read x_1 at L1, it later performs the operation $R(x_2)$ at L2. But only the write operations in $WS(x_2)$ have been performed at L2.No guarantees are given that this set also contains all operations contained in WS (x_1).

(b) Monotonic Writes

- In a monotonic-write consistent store, the following condition holds : A write operation by a process on a data item x is completed before any successive write operation on x by the same process.
- As a result, a write operation on a copy of item x is done only if that copy has been brought up to date by any previous write operation that may have occurred on other copies of x. If necessary, the new write must wait for the old ones to finish.
- Example: Updating a program on server S2 and ensuring that all components on which compilation and linking rely are also stored on S2.



(a) A monotonic-write consistent data store (b) A data store that does not provide monotonic-write consistency

(1E12)Fig. 5.3.13

- In Fig.5.3.13, the write operations are performed by a single process P at two different local copies of the same data store.
- In Fig. 5.3.13(a), Process P performs a write operation on x at local copy L1, presented as the operation $W(x_1)$.Later, P performs another write operation on x, but this time at L2, shown as $W(x_2)$.To ensure monotonic-write consistency, the previous write operation at L1 must have been propagated to L2.This explains operation $W(x_1)$ at L2, and why it takes place before $W(x_2)$.
- In Fig. 5.3.13(b), monotonic-write consistency is not guaranteed. The propagation of $W(x_1)$ to copy L2 is missing. No guarantees can be given that the copy of x on which the second write is being performed has the same or more recent value at the time $W(x_1)$ completed at L1.

(c) Read your Writes

- A data store is said to provide read-your-writes consistency if the following condition holds : The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.
- As a result, no matter where the read operation occurs, a write operation is always completed before a subsequent read operation by the same process.
- Example: Updating your Web page and ensuring that your Web browser displays the most recent version rather than its cached copy.

(a) A data store that provides read-your-writes consistency (b) A data store that does not provide read-your-writes consistency

(1E13)Fig.5.3.14

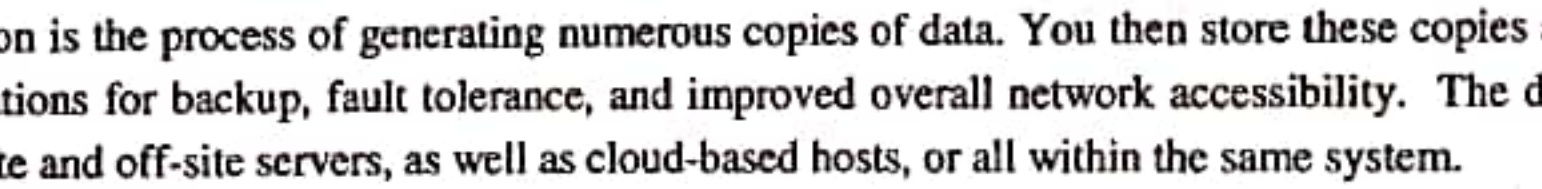
- In Fig.5.3.14(a), Process P performed a write operation $W(x_1)$ and later a read operation at a different local copy.

Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation. This is expressed by WS (x_1;x_2), which states that W (x_1) is part of WS (x_2).

- In Fig.5.3.14(b),W(x_1) has been left out of WS(x_2), meaning that the effects of the previous write operation by process P have not been propagated to L2.

(d) Writes follow Reads

- A data store is said to provide writes-follow-reads consistency, if the following holds : A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.
- As a result, any subsequent write operation on a data item x by a process will be executed on a copy of x that is up to date with the value most recently read by that process.
- For example, only view reactions to submitted articles if you have the initial posting (a read pulls in the corresponding write operation).



(a) A writes-follow-reads consistent data store (b) A data store that does not provide writes-follow-reads consistency

(IE14)**Fig. 5.3.15**

- In Fig.5.3.15(a), a process reads x at local copy L1.The write operations that led to the value just read, also appear in the write set at L2, where the same process later performs a write operation.(Note that other processes at L2 see those write operations as well.)
- In Fig. 5.3.15(b), no guarantees are given that the operation performed at L2. They are performed on a copy that is consistent with the one just read at L1.

5.4 REPLICATION

- Data Replication is the process of generating numerous copies of data. You then store these copies also called replicas in various locations for backup, fault tolerance, and improved overall network accessibility. The data replicas can be stored on on-site and off-site servers, as well as cloud-based hosts, or all within the same system.
- Data Replication in Distributed Systems refers to the distribution of data from a source server to other servers while keeping the data updated and synced with the source so that users can access data relevant to their activities without interfering with the work of others.

5.4.1 Need for Data Replication

Data in a Distributed System is stored among several computers in a network. Some of the reasons for Data Replication in Distributed Systems include :

(1) **Higher Availability :** In Distributed Systems, Replication is the most important aspect of increasing data availability. Data is replicated over numerous locations so that the user can access it even if some of the copies are unavailable due to site failures.

(2) **Reduced Latency :** By keeping data geographically closer to a consumer, Replication helps to reduce data query latency. For example, CDNs (Content Delivery Networks) such as Netflix, retain a copy of duplicated data closer to the user.

(c) **Read Scalability :** Read queries can be served from copies of the same data that have been replicated. This increases the overall throughput of queries.

(d) **Fault-Tolerant :** Even when there are network challenges, the system operates. If one replica fails, service can be supplied by another replica.

5.4.2 Types of Data Replication

There are several types of Data Replication in Distributed System based on certain types of architecture. Let's look at some of these below :

(1) **Asynchronous vs Synchronous Replication**

(i) **Asynchronous Replication :** In this replication, the replica gets modified after the commit(save) is fired onto the database.

(ii) **Synchronous Replication :** In this replication, the replica gets modified immediately after some changes are made in the relation table.

(2) **Active vs Passive Replication**

Active Replication

- Active Replication is a non-centralized replication mechanism. The central idea is that all replicas receive and process the same set of client requests.
- Consistency is ensured by assuming that replicas will generate the same output when given the same input in the same sequence. This assumption indicates that servers respond to queries in a deterministic manner.
- Clients do not address a single server, but rather a group of servers.
- Client requests can be broadcast to servers via an Atomic Broadcast for them to get the same input in the same sequence.

Passive Replication

- Client requests are processed by just one server (named primary) in Passive Replication.
- The primary server changes the status of the other (backup) servers after processing a request and responds to the client.
- One of the backup servers takes over if the primary server fails. Even non-deterministic processes can benefit from Passive Replication.
- The drawback of passive replication over active replication is that the response is delayed in the event of failure.

(3) **Based on Server Model**

(i) **Single Leader Architecture :** In this architecture, one server accepts client writes and replicas pull data from it. This is the most popular and traditional way. It's the synchronous technique, but it's also quite rigid.

(ii) **Multi Leader Architecture :** In this architecture, multiple servers can accept writes and serve as a model for replicas. To avoid delay, copies should be spread out and leaders should be near all of them.

(iii) **No Leader Architecture :** Every server in this architecture can receive writes and function as a replica model. While it provides maximum flexibility, it makes synchronization difficult.

(4) Based on Replication Schemes

Full Data Replication

- Full Replication refers to the replication of the whole database across all Distributed System sites.
- Across a wide area network, this technique maximizes data availability and redundancy.
- Since the results can be accessed from any local server, Full Replication speeds up the execution of global queries.
- The drawback of Full Replication is that the updating process is often sluggish. This makes maintaining current data copies in all locations challenging.

Partial Data Replication

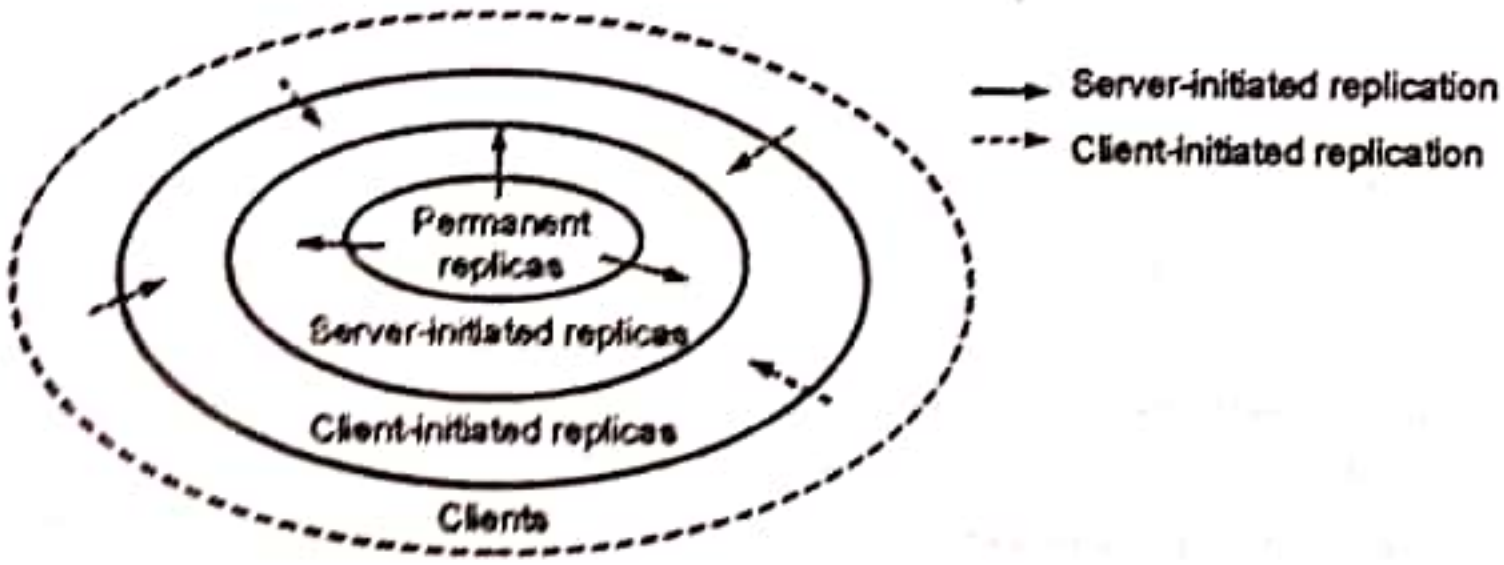
- Here, only selected parts of the database are replicated based on the significance of data at each site.
- The number of copies, in this case, can be anything from one to the total number of nodes in the Distributed System.
- This kind of replication can be effective for members of Sales and Marketing teams where a Partial Database is maintained on personal computers and synchronized with the main server regularly.

No Data Replication

- In this Replication scheme, each node in a distributed system receives just a copy of one section of the database.
- While the No Replication can be ascribed to the simplicity of data recovery, it might slow down query execution since several users access the same server.
- No Data Replication in DBMS gives low data availability when compared to alternative Replication techniques.

5.4.3 Replica Placement

The logical organization of different kinds of replicas of a data store into three concentric rings is shown in Fig.5.4.1 below.



(IT19)**Fig. 5.4.1 : Replica Placement**

(1) Permanent Replicas

- Permanent replicas are the first set of replicas that make up a distributed data storage.
- In many cases, the number of permanent replicas is small.
- A database is distributed and possibly replicated over several geographically dispersed sites.
- This architecture is typically used in federated databases.

Server-Initiated Replicas

- Server-initiated replicas are copies of a data store that exist to improve performance and are created at the request of the (data store's) owner.
- When the system performance metrics drop below a specific level, the data store owner makes a request to another server to host this replica.
- Server-initiated replicas are clustered near a high concentration of clients.
- A system makes dynamic decisions on where to place server-initiated replicas and when to generate new ones, destroy existing ones, or migrate existing ones.

Client-Initiated Replicas

- Client-initiated replicas are also referred to as (client) caches.
- A cache is essentially a local storage facility that a client uses to temporarily keep a copy of the data it has just requested.
- The data store from which the data was retrieved has nothing to do with maintaining cached data consistency.
- Client caches are only used to improve data access times.
- When a client needs to access data, it typically connects to the nearest copy of the data store, from which it retrieves the data it wants to read or stores the data it has just modified.
- Data is typically cached for a certain period of time, for example, to prevent severely stale data from being used, or simply to make space for new data.

5.4.4 Replication Models

There are three basic replication models the master-slave, client-server and peer-to-peer models.

(1) Master-Slave Model	(2) Client-Server Model	(3) Peer-to-Peer Model

(1) Master-Slave Model

- In this model one of the copy is the master replica and all the other copies are slaves.
- The slaves should always be identical to the master.
- In this model the functionality of the slaves are very limited, thus the configuration is very simple.
- The slaves essentially are read-only.
- Most of the master-slaves services ignore all the updates or modifications performed at the slave, and "undo" the update during synchronization, making the slave identical to the master.
- The modifications or the updates can be reliably performed at the master and the slaves must synchronize directly with the master.

(2) Client-Server Model

- The client-server model like the master-slave designates one server, which serves multiple clients.
- The functionality of the clients in this model is more complex than that of the slave in the master-slave model.
- It allows multiple inter-communicating servers; all types of data modifications and updates can be generated at the client.
- One of the replication systems in which this model is successfully implemented is Coda.
- Optimistic replication can use a client-server model.

- In Client- server replication all the updates must be propagated first to the server, which then updates all the other clients.
- In the client-server model, one replica of the data is designated as the special server replica.
- All updates created at other replicas must be registered with the server before they can be propagated further.
- This approach simplifies replication system and limits cost, but partially imposes a bottleneck at the server.
- Since all updates must go through the server, the server acts as a physical synchronization point.
- In this model the conflicts which occur are always be detected only at the server and only the server needs to handle them.
- However, if the single server machine fails or is unavailable, no updates can be propagated to other replicas.
- This leads to inconsistency as individual machines can accept their local updates, but they cannot learn of the updates applied at other machines.

(3) Peer-to-Peer Model

- The Peer-to-peer model is very different from both the master-slave and the client-server models.
- Here all the replicas or the copies are of equal importance, or they are all peers.
- In this model any replica can synchronize with any other replica, and any file system modification or update can be applied at any replica.
- Optimistic replication can use a peer-to-peer model.
- Peer-to-peer systems allow any replica to propagate updates to any other replicas.
- Peer-to-peer systems can propagate updates faster by making use of any available connectivity.
- They provide a very rich and robust communication framework. But they are more complex in implementation and in the states they can achieve.
- One more problem with this model is scalability.
- Peer models are implemented by storing all necessary replication knowledge at every site thus each replica has full knowledge about everyone else.
- As synchronization and communication is allowed between any replicas, this results in exceedingly large replicated data structures and clearly does not scale well.

5.4.5 Replica Consistency

- There are number of techniques for file replication that are used to maintain data consistency.
- Replication services maintain all the copies or replicas having the same versions of updates. This is known as maintaining consistency or synchronization.
- Replication techniques to provide consistency can be divided into two main classes :
  - (i) **Optimistic :** These schemes assume faults are rare and implement recovery schemes to deal with inconsistency
  - (ii) **Pessimistic :** These schemes assume faults are more common and attempt to ensure consistency of every access.
- Schemes that allow access when all copies are not available to use voting protocols to decide if enough copies are available to proceed.

(1) Pessimistic Replication

- This is a more conservative type of scheme using prime site techniques, locking, or voting for consistent data updates.

- As this approach assumes that failure is more common it guards against all concurrent updates.
- An update cannot be written if a lock cannot be obtained or if majority of other sites cannot be queried. In doing so you sacrifice data availability.
- The pessimistic model is a bad choice where frequent disconnections network and network partitions are common occurrence.
- It is used for more traditional DFS.

Optimistic Replication

- This approach assumes that concurrent updates or conflicts are rare.
- This scheme allows concurrent update, updates can be done at any replica or copy.
- This increases the data availability. However, when conflicts do occur, special action must be taken to resolve the conflict and merge the concurrent updates into a single data object.
- The merging is referred to as conflict resolution.
- When conflicts do occur, many can be resolved transparently and automatically without user involvement.
- This approach is used for mobile computing.

5.5 FAULT TOLERANCE

- The concept of partial failure distinguishes distributed systems from single-machine systems.
- A partial failure can occur when one component of a distributed system fails. This failure may impair the performance of other components while leaving others completely unaffected.
- In contrast, a failure in non-distributed systems is often total in the sense that it affects all components and can easily bring the entire system down.
- An important goal in distributed system design is to build the system in such a way that it can automatically recover from partial failures while maintaining overall performance.
- When a failure occurs, the distributed system should continue to function in an acceptable manner while repairs are being made, that is, it should tolerate faults and continue to function to some extent even in their presence.
- For a system to be fault tolerant, it is related to be dependable systems. Dependability covers some useful requirements in the fault tolerance system these requirements include: Availability, Reliability, Safety, and Maintainability.

(i) **Availability :** This is when a system is in a ready state and is ready to deliver its functions to its corresponding users. Highly available systems work at a given instant in time.

(ii) **Reliability :** This is the ability for a computer system run continuously without a failure. Unlike availability, reliability is defined in a time interval instead of an instant in time. A highly reliably system, works constantly in a long period of time without interruption.

(iii) **Safety :** This is when a system fails to carry out its corresponding processes correctly and its operations are incorrect, but no shattering event happens.

(iv) **Maintainability :** A highly maintainability system can also show a great measurement of accessibility, especially if the corresponding failures can be noticed and fixed mechanically.

5.5.1 Failure Models

- When a system fails, it is not providing the services for which it was created.
- If we take a distributed system to be a collection of servers that communicate with one another and with their clients, inadequate service provision implies that the servers, communication channels, or both are not performing as expected.
- However, a faulty server is not always the source of the problem. If such a server relies on other servers to offer acceptable services, the cause of a mistake may need to be found elsewhere.
- Such dependencies abound in distributed systems. A failed disc can make life difficult for a file server that is supposed to provide a highly available file system.
- If such a file server is part of a distributed database, the entire database's functionality may be jeopardised because just a portion of its data is available.
- Several classification schemes have been established to help determine the severity of a failure. Table 5.5.1 depicts one such scheme.

Table 5.5.1 : Different types of Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	A server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

(1) Crash Failure

- A crash failure occurs when a server abruptly stops working but was previously operational. An significant characteristic of crash failures is that nothing is heard from the server after it has stopped.
- A typical example of a crash failure is an operating system that grinds to a halt and has just one solution: reboot it.
- Many personal computer systems experience crash failures so frequently that individuals have grown to accept them as usual.
- As a result, moving the reset button from the back to the front of a cabinet was done for a valid purpose. Perhaps one day it will be moved back again, or even eliminated entirely.

(2) Omission Failure

- Omission failures arise when servers don't respond to requests. Several issues may arise. A receive omission failure may mean the server never received the request.
- Note that a client-server connection may be created but no thread listening to incoming requests. Since the server is unaware of messages transmitted to it, a receive omission failure usually does not change its status.
- A send omission failure occurs when the server completes its task but fails to respond. A server may fail if a send buffer overflows without preparation.

- Unlike a receive omission failure, the server may now be in a state indicating that it has finished serving the client. Thus, if its answer fails, the server must be ready for the client to resubmit.

Timing Failure

- Timing failures occur when responses are outside a real-time interval.
- If there isn't enough buffer space, sending data too soon can cause issues. However, a performance failure occurs when a server responds too late.

Response Failure

- Response failures, where the server responds incorrectly, are serious. Two types of response failures can occur. In a value failure, a server sends the erroneous response. For instance, a search engine that consistently delivers Web pages unrelated to search terms has failed.
- State transition failures are the other form of response failure. This failure occurs when the server responds unexpectedly to a request.
- A state transition failure occurs if a server receives a message it doesn't recognise. In particular, a malfunctioning server may perform default activities it shouldn't.

Arbitrary Failure

- Arbitrary failures, often known as Byzantine failures, are the most serious. Clients should be prepared for the worst when arbitrary failures occur.
- In particular, it is possible that a server is providing output that it should never have produced yet cannot be identified as inaccurate. Worse, a broken server may be maliciously collaborating with other servers to produce purposefully incorrect replies. Crash failures are strongly related to arbitrary failures.
- Finally, there are times when the server generates random output, yet this data is recognized as junk by other programs. The server then displays arbitrary failures, but in a benign manner. These errors are also known as fail-safe faults.

5.5.2 Failure Masking

If a system is to be fault tolerant, the best it can do is attempt to hide the occurrence of failures from other processes. The main strategy for hiding faults is redundancy.Three kinds are possible: information redundancy, time redundancy, and physical redundancy.

1) **Information Redundancy :** With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code, parity, CRC Checksum can be added to transmitted data to recover from noise on the transmission line.
2) **Time Redundancy :** With time redundancy, an action is performed once and then repeated if necessary. This method is used in transactions. If a transaction fails, it can be restarted with no consequences. When the defects are transient or intermittent, time redundancy is helpful.
3) **Physical Redundancy :** Physical redundancy involves the addition of extra equipment or processes to allow the system as a whole to tolerate the loss or malfunction of some components. Physical redundancy can thus be implemented in either hardware or software. Extra processes, for example, can be added to the system such that even if a small number of them fail, the system continues to work normally. In other words, a high degree of fault tolerance can be attained by replicating processes.

5.6 PROCESS RESILIENCE

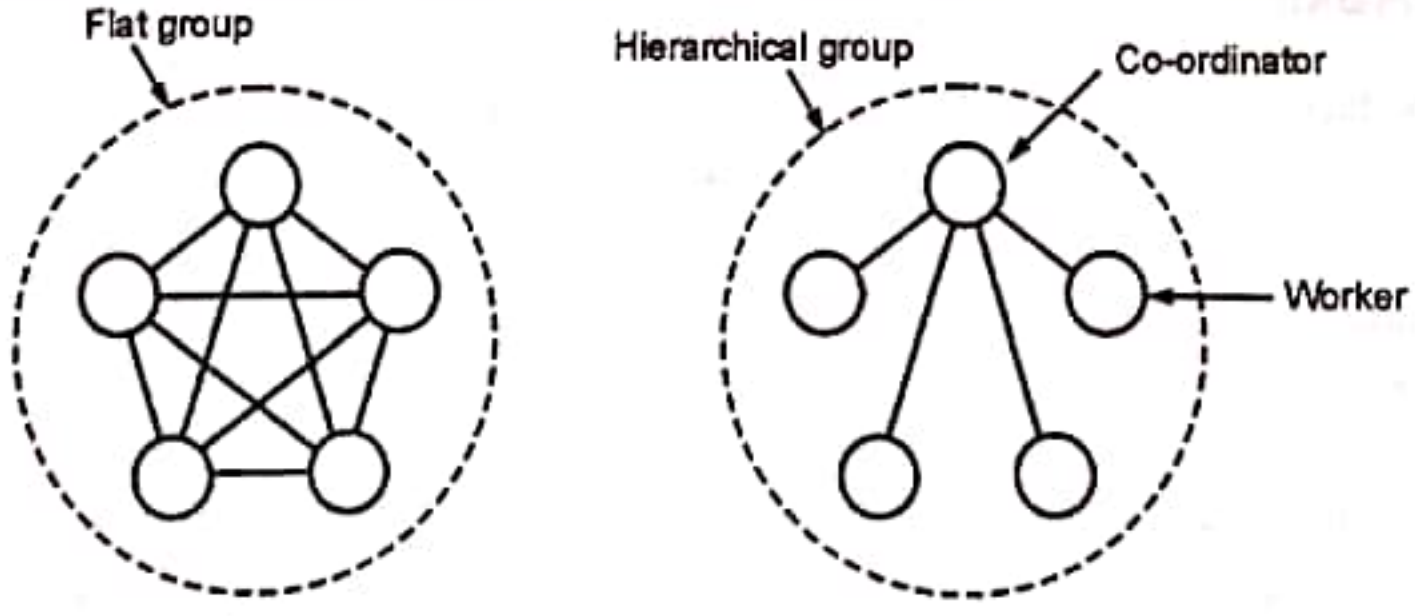
Now that we've covered the fundamentals of fault tolerance, let's look at how fault tolerance can be implemented in distributed systems.

5.6.1 Design Issues

- The key strategy to tolerating a faulty process is to cluster multiple identical processes together.
- When a message is sent to a group, all members of the group receive it. As a result, if one process in a group fails, another process should be able to take its place.
- Process groups can be dynamic. New groups can be formed, while old groups can be destroyed.
- During system functioning, a process may join or leave a group.
- A process can be a member of multiple groups at the same time. As a result, systems to manage groups and group membership are required.
- The introduction of groups is intended to allow processes to deal with collections of processes as a single abstraction.
- As a result, a process can send a message to a group of servers without knowing who they are, how many there are, or where they are, which may change from call to call.

5.6.1(A) Group Organization

- The internal structure of various groups is a significant point of differentiation.
- All the processes are equal in some groups. There is no supervisor, and all decisions are made in groups. Such group is called flat group.
- Other groups contain some level of hierarchy. For instance, one process serves as the coordinator while all the others serve as workers. When a job request is made under this approach, it is forwarded to the coordinator by either an external client or one of the workers. The coordinator then selects the worker who is best qualified to complete it and forwards it to them.
- Of course, more complicated hierarchies are also feasible. In Fig. 5.6.1, these communication patterns are depicted.



(a) Communication in a flat group **(b) Communication in a simple hierarchical group**

(1E16)**Fig. 5.6.1**

- Each of these organizations has its own advantages and disadvantages.
- The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue.

A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often must be taken, incurring some delay and overhead.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but if it is running, it can make decisions without bothering everyone else.

5.6.1(B) Group Membership

- When group communication is present, some method is required for group creation and deletion as well as for allowing processes to join and exit groups.
- Having a group server to which all of these requests can be forwarded is one strategy that might be used.
- The group server can then keep a complete database of all the groups and their precise membership. This approach is simple, effective, and comparatively simple to use.
- Unfortunately, it shares a significant drawback with all centralized techniques: a single point of failure.
- Group management ends if the group server fails. Most or all groups will probably need to be rebuilt from scratch, which could result in the termination of any ongoing activity.
- The alternative approach is to distribute group membership management. For example, if (reliable) multicasting is allowed, an outsider can send a message to all group members informing them of their desire to join the group.
- To leave a group, a member should ideally simply send a goodbye message to everyone. Assuming fail-stop semantics is often not appropriate in the context of fault tolerance.
- The problem is that, unlike when a process quits freely, there is no polite notification when a process crashes. The other members must figure this out for themselves by seeing that the crashed member no longer responds to anything. Once it is determined that the crashed member is truly down (rather than simply slow), it can be removed from the group.
- Another tricky issue is that leaving and joining must be synchronized with the transmission of data messages. In other words, from the moment a process joins a group, it must receive all messages issued to that group.
- Similarly, after a process has left a group, it must not receive any further messages from the group, and the remaining members must not receive any further messages from it.
- Converting a join or depart operation into a sequence of messages delivered to the entire group is one approach to ensure that it is integrated into the message stream at the correct point.
- One final issue with group membership is what to do if so, many machines fail that the group is unable to function at all. To rebuild the group, certain protocol is required. Invariably, some process will have to take the initiative to get things started, but what if two or three processes try at the same time? This must be permitted under the protocol.

5.6.2 Failure Masking and Replication

- Process groups are a component of the approach for constructing fault-tolerant systems.
- Having a group of identical processes, in particular, allows us to mask one or more faulty processes in that group. In other words, we can replicate processes and group them together to replace a single (susceptible) process with a collection of (fault tolerant) processes.
- As stated in the earlier section, there are two approaches to such replication: primary-based protocols and replicated-write protocols.
- In the event of fault tolerance, primary-based replication takes the shape of a primary-backup protocol. A collection of processes is clustered in this context, with a primary coordinating all write activities. In practice, the primary is fixed, although it can be replaced by one of the backups if necessary. When the primary fails, the backups use an election mechanism to select a new primary.

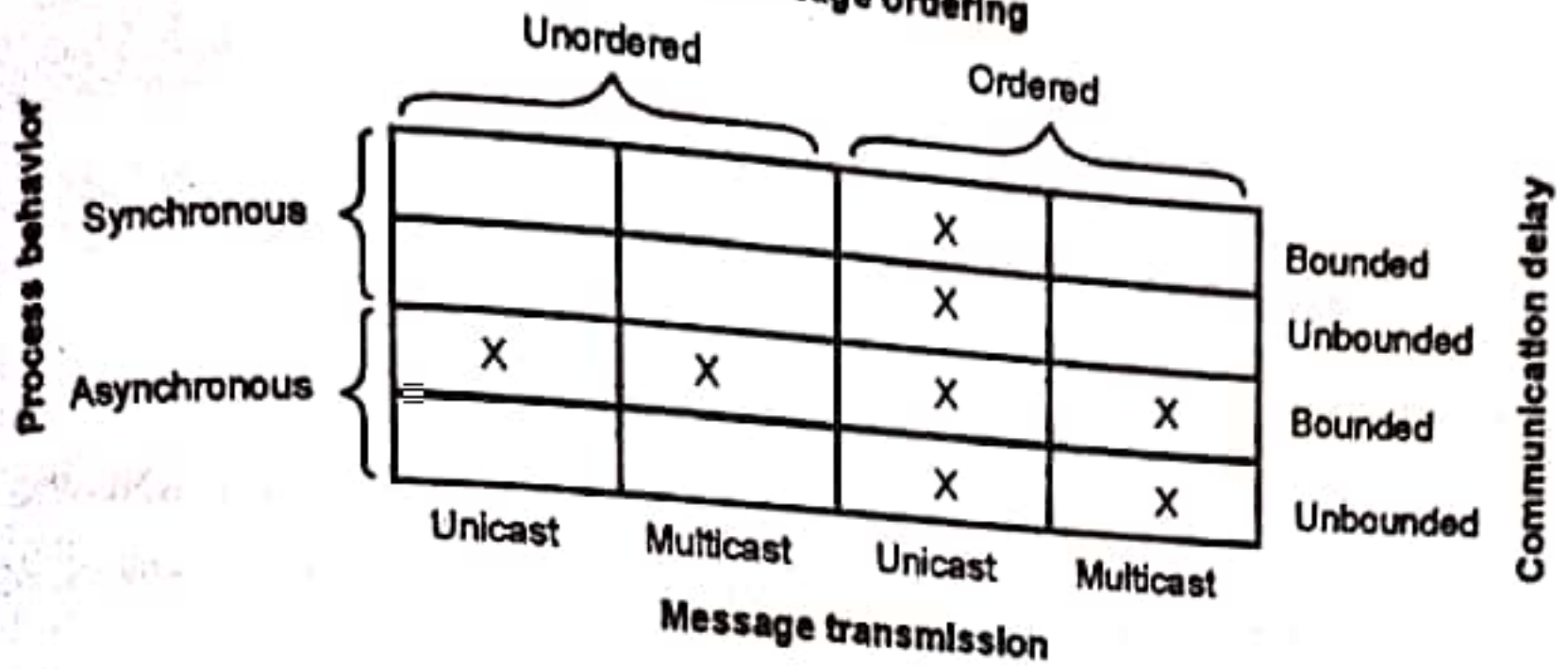
- Replicated-write protocols are utilised in both active replication and quorum-based protocols. These solutions correspond to grouping a collection of identical processes into a flat group. The key advantage is that such organizations have no single point of failure, at the expense of distributed coordination.
- The amount of replication required when employing process groups to tolerate faults is a significant consideration. To keep things simple, we'll only look at replicated-write systems. If a system can withstand faults in k components while still meeting its criteria, it is said to be k fault tolerant. If the components, say, processes, fail silently, then k + 1 of them is sufficient to offer k fault tolerance. If one of them simply stops, the other's answer can be used.
- However, if processes exhibit Byzantine failures, such as running when sick and sending out erroneous or random responses, a minimum of 2k + 1 processors are required to achieve k fault tolerance. In the worst-case scenario, the k failed processes may generate an identical response by accident (or perhaps intentionally). However, the remaining k + 1 will also yield the same result, thus the client or voter can simply trust the majority.
- Of course, in theory, it is OK to declare that a system is k fault tolerant and just let the k + 1 identical replies outvote the k identical replies, but it is difficult to imagine conditions in which k processes might fail but k + 1 processes cannot fail. Thus, even with a fault-tolerant system, statistical analysis may be required.
- Of course, in theory, it is fine to declare that a system is k fault tolerant and just let the k + 1 identical replies outvote the k identical replies, but it is difficult to imagine conditions in which k processes might fail but k + 1 processes cannot fail. Thus, even with a fault-tolerant system, statistical analysis may be required.
- An underlying requirement for this model to be meaningful is that all requests arrive at all servers in the same sequence, often known as the atomic multicast problem.
- This constraint can be significantly relaxed because reads are irrelevant and certain writes may commute, but the overall problem remains.

5.6.3 Agreement in Faulty Systems

- Organizing replicated processes into groups improves fault tolerance.
- As previously stated, if a client can base its decisions on a voting mechanism, we can even tolerate if k out of 2k + 1 processes lie about their results. The assumption we are making is that processes do not collaborate to produce an erroneous result.
- In general, things grow more complicated when we expect that a process group reach an agreement, which is often required.
- Among the many choices are: selecting a coordinator, deciding whether or not to commit to a transaction, assigning tasks to workers, and synchronisation.
- When communication and processes are perfect, attaining such an agreement is generally simple; but when they are not, issues occur.
- The overall objective of distributed agreement algorithms is to have all nonfaulty processes reach consensus on some issue in a finite number of steps.
- Following cases should be considered :
  - (1) **Synchronous versus asynchronous systems :** A system is synchronous if and only if the processes are known to operate in a lock-step mode. Formally, this means that there should be some constant c>=1, such that if any processor has taken c + 1 steps, every other process has taken at least 1 step. A system that is not synchronous is said to be asynchronous.
  - (2) **Communication delay is bounded or not :** Delay is bounded if and only if we know that every message is delivered with a globally and predetermined maximum time.

(3) **Message delivery is ordered or not :** In other words, we distinguish the situation where messages from the same sender are delivered in the order that they were sent, from the situation in which we do not have such guarantees.

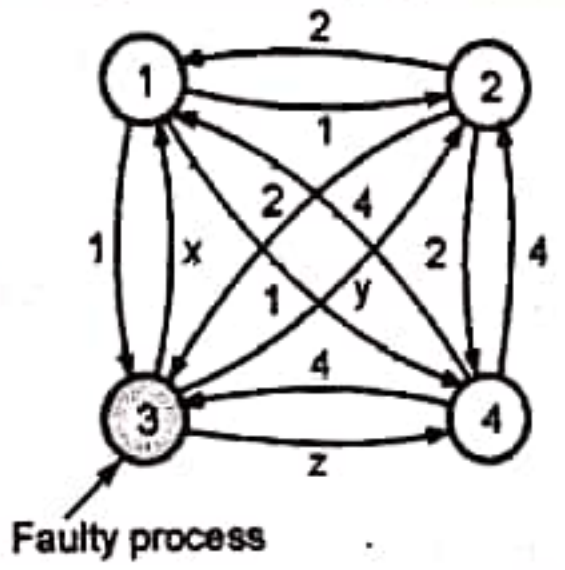
Message transmission is done through unicasting or multicasting.



(IE17)**Fig.5.6.2 : Circumstances under which distributed agreement can be reached**

5.6.3(A) Byzantine Agreement Problem

- In this case, we assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded.
- We assume that there are N processes where each process i will provide a value vi to the others. and there are at most k faulty processes.
- The goal is let each process construct a vector V of length N such that if process i is nonfaulty, V[i] = vi. Otherwise, V[i] is undefined.
- Let N = 4 and k = 1.For these parameters, the algorithm operates in four steps.



(IE18)**Fig. 5.6.3 : The Byzantine Agreement Problem for 3 Non-faulty and 1 Faulty Process**

1) Every non-faulty process i sends vi to every other process using reliable unicasting. Faulty processes may send anything and may send different values to different processes. Let vi =i. In Fig. 5.6.2 we see that process 1 reports 1, process 2 reports 2, process 3 lies to everyone, giving x, y, and z, respectively, and process 4 reports a value of 4.
2) The results of the announcements of step 1 are collected in the form of the vectors as shown in Fig. 5.6.4.

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(IE19)**Fig. 5.6.4 : The vectors that each process assembles from Fig.5.6.3.**

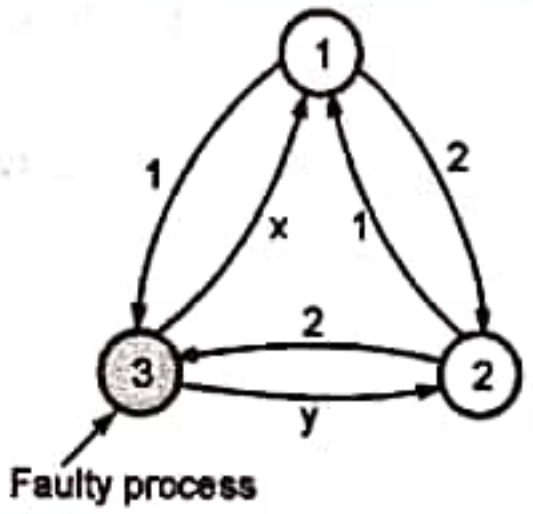
(3) Every process passes its vector from Fig. 5.6.4 to every other process. In this way, every process gets three vectors, one from every other process. Here, too, process 3 lies, inventing 12 new values, *a* through *l*. The results of step 3 are shown in Fig. 5.6.5.

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(1E20)**Fig.5.6.5 : The vectors that each process receives in step 3**

(4) Each process examines the i^{th} element of each of the newly received vectors.
   - If any value has a majority, that value is put into the result vector.
   - If no value has a majority, the corresponding element of the result vector is marked UNKNOWN.
   - From Fig.5.6.4, we see that 1, 2, and 4 all come to agreement on the values for v1, v2, and v4, which is the correct result.
   - The conclusion from these processes regarding v3 cannot be decided but is also irrelevant.

- The goal of Byzantine agreement is that consensus is reached on the value for the nonfaulty processes only.
- Now let us try this problem for N = 3 and k = 1, that is, only two nonfaulty process and one faulty one, as illustrated in Fig. 5.6.6.
- Here we see that in Fig. 5.6.6(c) neither of the correctly behaving processes sees a majority for element 1, element 2, or element 3, so all of them are marked UNKNOWN. The algorithm has failed to produce agreement.



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

(1E21)**Fig.5.6.6 : The Byzantine Agreement Problem for 2 Non-faulty and 1 Faulty Process**

- Lamport et al. proved that in a system with k faulty processes, agreement can be achieved only if 2k + 1 correctly functioning processes are present, for a total of 3k + 1.
- An agreement is possible only if more than two-thirds of the processes are working properly.

5.6.4 Failure Detection

- Failure detection means that a non-faulty member of a process group should be able to detect when a member has failed.
- This detection can be done by either sending probing messages or sending the status of all members to all other processes periodically.
- Gossiping protocol in which each node regularly announces to its neighbors that it is still up can be used.
- Eventually, every process knows about every other process and has enough information that is available locally to decide whether a process has failed or not. Once the failure detection is done, the group members create a spanning tree that is used for monitoring the failures of members.

The members send ping messages to their neighbours. When a neighbour does not respond, the pinging node immediately switches to a state in which it will also no longer respond to pings from other nodes.

By recursion, it is seen that a single node failure is rapidly promoted to a group failure notification.

5.7 RELIABLE CLIENT-SERVER COMMUNICATION

In many cases, fault tolerance in distributed systems focuses on faulty processes. However, we must also consider communication failures. Most of the failure models addressed previously apply equally well to communication channels.

A communication channel, in particular, may exhibit crash, omission, timing, and arbitrary failures. In practise, when constructing reliable communication channels, the emphasis is on masking crash and omission failures.

Arbitrary failures may arise in the form of duplicate messages, as a result of messages being delayed for a relatively long time in a computer network and being reinjected into the network after the original sender has already issued a retransmission.

5.7.1 Point-to-Point Communication

Many distributed systems employ a reliable transport protocol, such as TCP, to establish reliable point-to-point communication. TCP uses acknowledgments and retransmissions to mask omission failures, which manifest as lost messages. TCP clients are fully unaware of such failures.

Crash failures of connections, on the other hand, are not hidden. When a TCP connection is abruptly terminated for whatever reason, no more messages can be transmitted across the channel, and a crash failure occurs.

In most circumstances, an exception is raised to notify the client that the channel has crashed.

The only method to hide such failures is for the distributed system to try to automatically establish a new connection by simply resending a connection request.

The underlying assumption is that the other party is still, or will be in the future, responsive to such requests.

5.7.2 RPC Semantics in the Presence of Failures

In Remote Procedure Call (RPC), communication is established between the client and the server. In RPC, a client side process calls a procedure implemented on a remote machine or a server. The goal of RPC is to hide communication by making remote procedure calls look just like local ones.

If any error occurs, then masking differences between remote and local procedure calls is not easy. In RPC system following failure may occur.

(1) The client is unable to locate the server.
(2) The request message from the client to the server is lost.
(3) The server crashes after receiving a request.
(4) The reply message from the server to the client is lost.
(5) The client crashes after sending a request.

Each of these categories poses different problems and requires different solutions.

(1) The Client Cannot Locate the Server

In this case, it may happen that the client cannot locate a suitable server. All servers might be down. Alternatively, assume the client is built with a certain version of the client stub and the binary is not utilized for a long period of time.

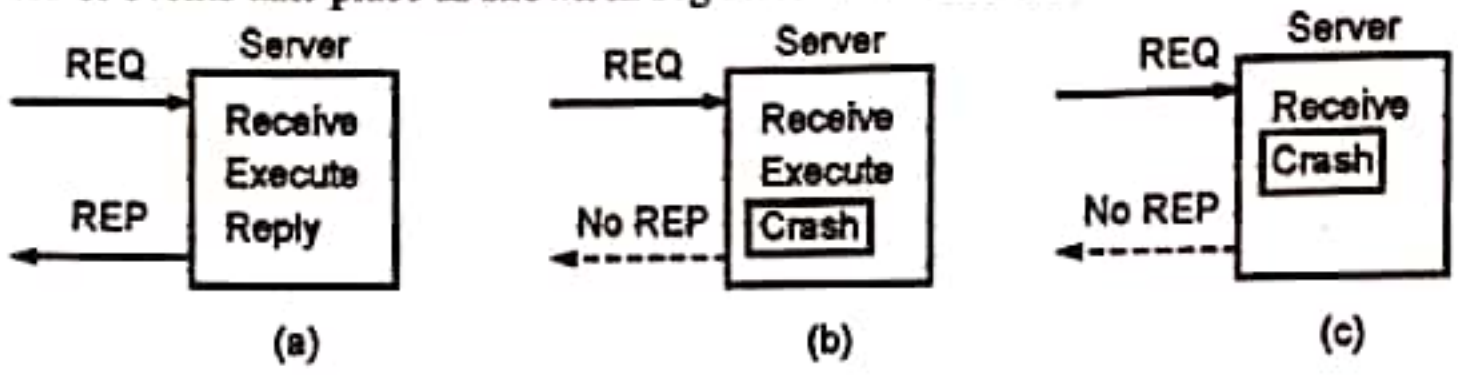
- Mean while, the server is evolving, and a new version of the interface is installed; new stubs are generated and used. When the client is launched, the binder will be unable to connect it to a server and will report failure.
- As a solution to this problem, an exception can be raised upon errors. In some languages, (e.g., Java), programmers can write special procedures that are invoked upon specific errors, such as division by zero.
- In C, signal handlers can be used for this purpose. This method, too, has drawbacks. To begin with, not all languages have exceptions or signals. Another problem is that having to build an exception or signal handler removes the transparency we were aiming for.

(2) Lost Request Messages

- To deal with this problem, the operating system or client stub start a timer when sending the request.
- If the timer runs out before a reply or acknowledgement is received, the message is sent again.
- If the message was actually lost, the server will be unable to distinguish between the resend and the original, and everything will function normally.
- Suppose so many request messages are lost that the client gives up and falsely concludes that the server is down, then situation will be like "Cannot locate server."
- If the request was not lost, the server will be able to detect it and deal with a retransmission.

(3) Server Crashes

- Following sequence of events take place as shown in Fig.5.7.1 when client sends request to the server.



(IT22)**Fig. 5.7.1 : Client-Server Communication**

- In (a), a request arrives at the server. Server has processed the request and reply is sent to the client.
- In (b), a request arrives and is processed by the server, just as before, but the server crashes before it can send the reply.
- In (c), again a request arrives, but this time the server crashes before it can even process it. And, of course, no reply is sent back.
- In case (b) and (c), different solutions are required. In (b), system ahs to report failure to the client. In (c), retransmission of request is required. Client retransmits when the timer expires.
- One approach is to wait until the server reboots (or rebind to a different server) before performing the operation again. The objective is to keep trying until you get a reply, then deliver it to the client. This technique is known as **at least once semantics**, and it ensures that the RPC has been performed at least once, if not multiple times.
- The second approach is giving up right away and report failure. This is known as **at-most-once semantics**, and it ensures that the RPC has been performed at most once, if at all.
- The third approach is to make no guarantees. When a server fails, the client receives no assistance and no assurances about what occurred. The RPC could have been performed anywhere from zero to many times. The fundamental advantage of this approach is its simplicity.

(4) Lost Reply Messages

- In this case client retransmits request message when reply does not arrive before timer goes off. It concludes that the request is lost and retransmits it again.

But, in this client actually remains unaware about what has happened actually. Idempotent operations can be repeated. These operations produce the same result even if repeated multiple times. Requesting first 1024 bytes of a file is idempotent. It will only overwrite at client side.

Some requests are non-idempotent. We need to configure all requests in idempotent way.

Another option is to have the client assign a sequence number to each request. The server can tell the difference between an original request and a retransmission by keeping track of the most recently received sequence number from each client that is using it and can refuse to carry out any request a second time by keeping track of the most recently received sequence number from each client that is using it. However, the server will still be required to respond to the client.

It should be noted that this strategy necessitates the server maintaining administration on each client. Furthermore, it is unclear how long this administration will be in place.

Another protection is to include a bit in the message header that distinguishes between initial requests and retransmissions (the idea being that it is always safe to perform an original request; retransmissions may require more care).

(5) Client Crashes

In this case, a client sends a request to a server to do some work and crashes before the server replies. At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan**.

Orphans can pose a range of issues that can disrupt the system's normal operation. At the very least, they waste CPU cycles. They can also encrypt files or otherwise starve important resources. Finally, if the client reboots and performs the RPC again, but the response from the orphan arrives quickly after, confusion may arise.

This orphan needs solution in RPC operation. Four possible solutions are suggested as below :

(i) **Orphan Extermination :** In this solution, before a client stub sends an RPC message, it makes a log entry telling what it is about to do. The log is kept on disk or some other medium that survives crashes. After a reboot, the log is checked, and the orphan is explicitly killed off. This solution is called orphan extermination.

(ii) **Reincarnation :** In this solution, no need to write log on disk. Time is divided in sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations on behalf of that client are killed. Of course, if the network is partitioned, some orphans may survive. Fortunately, however, when they report back, their replies will contain an obsolete epoch number, making them easy to detect.

(iii) **Gentle Reincarnation :** When an epoch broadcast comes in, each machine checks if it has any remote computations running locally, and if so, tries its best to locate their owners. The computation is killed only if the owners cannot be located anywhere.

(iv) **Expiration :** In this solution, each RPC is given a quantum time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum, which is a nuisance. On the other hand, if after a crash the client waits a time T before rebooting, all orphans are sure to be gone.

5.8 RELIABLE GROUP COMMUNICATION

Multicast is a popular implementation of group communication, for example, data can be replicated within database cluster with multicasts.

(1) Ordering Multicast

Let's first look at 3 types of events :

- Send event: the event of transferring a message from a sender.
- Receive event: the event of buffering data by the receiver.
- Deliver event: the event of passing data from the buffer to the **application layer** by the receiver.

The order of message delivery is important as it affects the correctness of the distributed systems. There are 4 flavours of reliable multicast:

(i) Unordered multicast

(ii) FIFO-ordered multicast

(iii) Causal-ordered multicast

Also, we could have an additional constraint - *the totally-ordered multicast*, that can be used together with any of the 3 flavours above to form hybrid styles.

(2) Unordered Multicast

- This is synchronous multicast where order of delivery by different receivers are not guaranteed to be the same.
- In another words, every receiver might deliver the messages in the order that they received, regardless of other receivers in the same group. For instance:

(3) FIFO-ordered Multicast

- The messages are delivered in the order they are send by the sender.
- The receivers, therefore, are expected to *deliver* the messages to its application layer in this exact order with respect to each individual sender. For instance, if a process A sends m1 follows by m2, but a receiving process B receives m2 first, then it has to wait for m1 to arrive before delivering m1 followed by m2 to the application layer

```
Process A sends m1 => then sends m2

Process B receives: m2 then wait for m1 => start delivering m1 then m2
```

- By definition, this type of multicast does not care about the order of delivery between different senders. So, for example:

```
Process A sends m1 => then sends m2

Process B sends m3 => then sends m4
```

Process C receives messages in order: m1,m3,m2,m4 => deliver m1,m2 => then deliver m3,m4

```
Process D receives messages in order: m3,m1,m2,m4 => deliver m3,m4 => then deliver m1,m2
```

- In above example, processes C and D deliver in different order, however, they still deliver in correct sending order with respect to each sender.

(4) Causal-ordered Multicast

- In this multicast, messages delivery must follow the causal order of the messages. For instance, a message m1 casually happens before another message m2, regardless whether they were sent from same or different senders, then the receiver will always delivered m1 before m2.We can use vector timestamp to determine the causality of the messages.

- It is easy to see that if a multicast obeys casual ordering, it will also obey FIFO ordering, i.e. if m1 were sent before m2 by the same sender, then m1 also happens-before m2. However, the reverse is *not true*, the multicast obeying FIFO ordering might not deliver messages by different senders with respect to their causal order.
- Casual-ordered multicast is useful in some applications such as social networks, bulletin boards, comments on websites, etc. For example, when you participate in a group chat, and you send a first message:
  (i) If a friend respond to your message, then it is necessary that anyone should see your message before this friend's response.
  (ii) However, if two friends respond concurrently, then they can be seen in any order at receivers.

Totally-ordered Multicast or Atomic Multicast

- Totally-ordered multicast is an additional constraint that is used in conjunction with another type of multicast.
- Total-ordered multicast ensures that all receivers in the group deliver messages in exactly the same order, i.e. in replicated services where the replicas receive update and forward the update to every others, they need to do that in the same order.
- So in the example of FIFO-multicast above, both processes C and D can deliver messages in the order m1,m3,m2,m4 which they can satisfy both FIFO and total orderings, this is called FIFO atomic multicast.
- Total-ordered multicast is an application of Lamport's logical clock. Let's consider a group of processes where each multicasts messages to others, including itself. It's reasonable to assume that the order of receiving messages will be in the order that the same sender send, with no lost messages.
- When a process receives a message, it puts into a local queue according to the timestamp.
- If we use Lamport's clock, then the received message is lower of the acknowledgments.
- The process will be removed and delivered the message at the head of the queue to its' application layer if the message has been acknowledged by other processes. These acknowledgments will also be removed when the messaged has been delivered.
- As a result, every process in the group will have the same local queue, and subsequently deliver messages in the same order.

5.9 RECOVERY

- So far, we've mostly focused on algorithms that allow us to tolerate faults. However, once a failure has occurred, it is critical that the process where the failure occurred can be restored to its original state.
- In what follows, we will first discuss what it means to recover to a correct state, and then when and how the state of a distributed system can be documented and recovered to using check pointing and message logging.
- The recovery after an error is fundamental to fault tolerance. Remember that an error is a component of a system that can cause it to fail. The goal of error recovery is to replace an incorrect state with a correct state. Error recovery can be divided into two categories.
  (I) The basic goal of **backward recovery** is to return the system from its current erroneous state to a previously accurate condition. To do this, it will be essential to periodically record the system's state and restore such a recorded state when things go wrong. A checkpoint is stated to be made whenever (part of) the system's current state is recorded.

(2) **Forward recovery** is another type of mistake recovery. When the system enters an erroneous state, rather than returning to a previous, check pointed state, an attempt is made to get the system back into a correct new state from which it can continue to execute. The fundamental issue with forward error recovery techniques is that the errors that may occur must be understood in advance. Only in this situation can those errors be corrected and a new state established.

- When examining the implementation of reliable communication, the contrast between backward and forward error recovery is clearly explained. The most typical way to recover from a lost packet is to allow the sender to retransmit the packet. In fact, packet retransmission indicates that we are attempting to return to a previous, correct state, namely the one in which the lost packet is being delivered.
- Reliable communication via packet retransmission is thus an application of backward error recovery techniques.
- Backward error recovery strategies, in general, are frequently used as a general mechanism for recovering from errors in distributed systems. The main advantage of backward error recovery is that it is a method that can be used to any system or process. In other words, it can be integrated as a general-purpose service into (the middleware layer) of a distributed system.
- Finally, although backward error recovery requires checkpointing, some states can simply never be rolled back to.
- **Checkpointing** enables for the restoration of a previously correct state. Taking a checkpoint, on the other hand, is frequently an expensive process with a significant performance penalty. As a result, many fault-tolerant distributed systems incorporate checkpointing and message logging.
- In this scenario, after reaching a checkpoint, a process logs its messages before sending them off (called **sender-based logging**). Another option is for the receiving process to log an incoming message before delivering it to the programme that is currently running. This method is also known as **receiver-based logging**. When a receiving process breaks, the most recently checkpointed state must be restored, and the messages that have been transmitted must then be replayed.
- As a result, integrating checkpoints with message logging allows you to recover a state that is past the most recent checkpoint without incurring the penalty of checkpointing.

5.9.1 Stable Storage

- To be able to recover to a previous state, the information required for recovery must be safely kept. In this case, safely means that recovery information survives process crashes and site failures, as well as any storage medium failures. When it comes to distributed system recovery, stable storage is critical.
- Storage is classified into three types :
  (1) RAM memory, which is erased when the power goes out or a machine crashes.
  (2) Disc storage, which can survive CPU failures but is vulnerable to disc head crashes.
  (3) Finally, there is stable storage, which is meant to withstand all but extreme disasters such as floods and earthquakes. Stable storage can be done using a pair of conventional discs. Each block on drive 2 is a carbon copy of its counterpart on drive 1. When a block is updated, the block on drive 1 is updated and confirmed first, followed by the same block on drive 2.
- Assume the system crashes after updating drive 1 but before updating drive 2. When the disk is recovered, it can be compared block by block. When two corresponding blocks differ, drive 1 is assumed to be the correct one (since drive 1 is always updated before drive 2), and the new block is copied from drive 1 to drive 2. When the recovery process is finished, both drives will be identical once more.
- Another potential issue is a block's spontaneous degradation. Dust particles or normal wear and tear can lead a previously valid block to have an unexpected checksum mistake without a cause or warning. When an error of this nature is found, the defective block can be regenerated from the corresponding block on the other drive.

...Stable storage, because of its implementation, is particularly suited to applications that demand a high degree of fault ...erance, such as atomic transactions.

...When data is written to stable storage and then read back to ensure that it was correctly written, the likelihood of them ...being lost is extremely low.

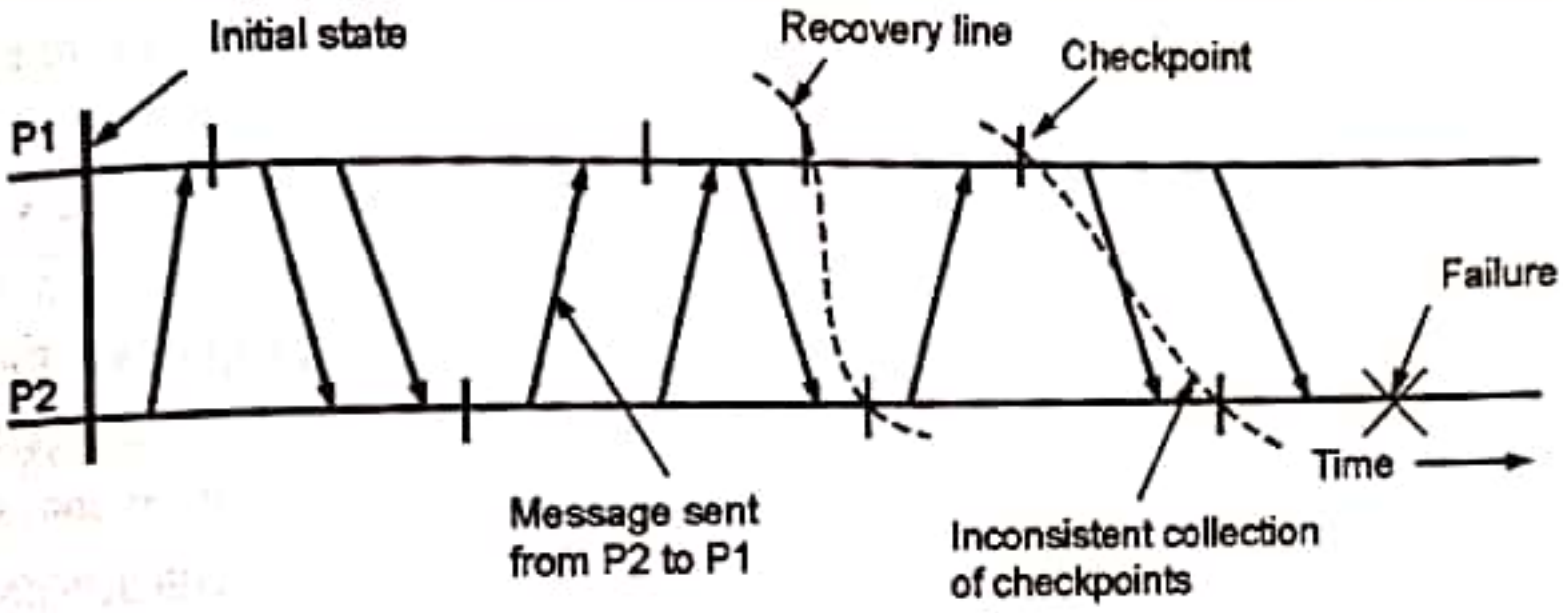
5.9.2 Checkpointing

...Backward error recovery in a fault-tolerant distributed system necessitates that the system preserve its state on stable ...orage on a regular basis.

...We must, in particular, capture a consistent global state, often known as a distributed snapshot.

...If a process P has recorded the receipt of a message in a distributed snapshot, there should also be a process Q that has ...recorded the transmission of that message. After all, it must have originated somewhere.

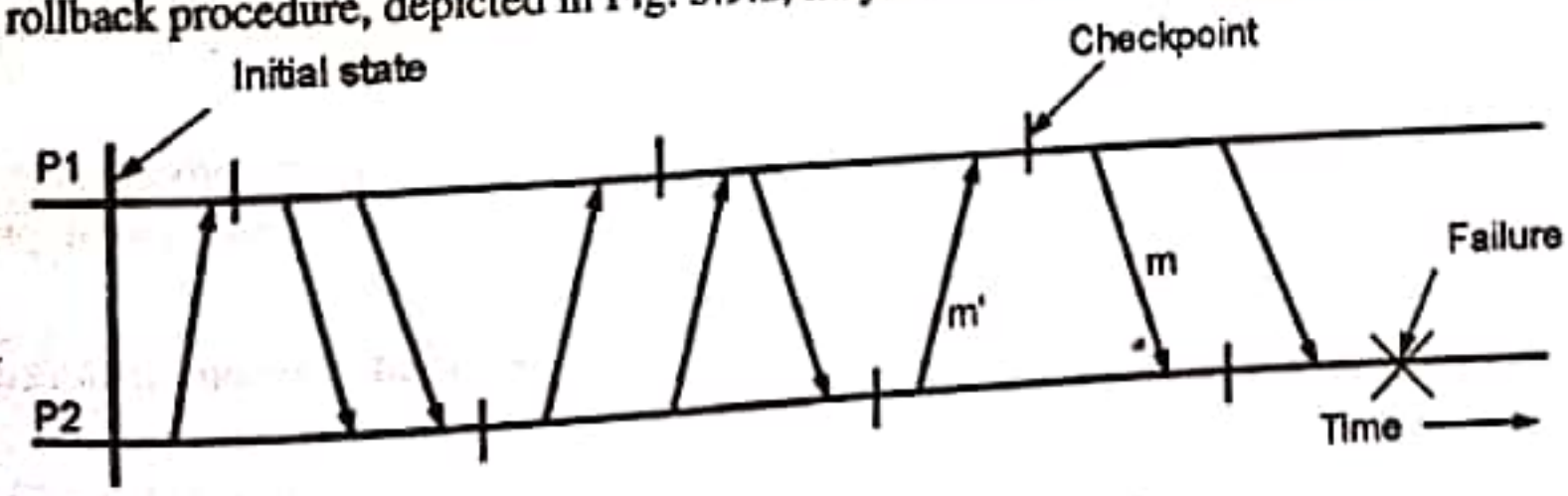
...In backward error recovery techniques, each process periodically saves its state to a locally available stable storage. To ...recover from a process or system failure, we must build a consistent global state from these local states. It is very ...important to recover to the most recent distributed snapshot, commonly known as a recovery line. In other words, as ...shown in Fig. 5.9.1, a recovery line corresponds to the most recent consistent collection of checkpoints.



(1E23)**Fig.5.9.1 : A Recovery Line**

...dependent Checkpointing

- If each process just records its local state in an uncoordinated method from time to time, it may be difficult to find a ...recovery line.
- To find a recovery line, each process must be rolled back to its most recently saved state.
- If these local states do not create a distributed snapshot, more rolling back is required.
- This cascaded rollback procedure, depicted in Fig. 5.9.2, may result in the **domino effect**.



(1E24)**Fig.5.9.2 : The Domino Effect**

- When process P2 crashes, we must restore to the most recently saved checkpoint.
- As a result, procedure P1 will also need to be rolled back.

- Unfortunately, the two most recently saved local states do not constitute a coherent global state: the P2 state shows receipt of a message m, but no other process can be recognized as its sender. As a result, P2 must be rolled back to an earlier state.
- The next state to which P2 is rolled back, on the other hand, cannot be used as part of a distributed snapshot.
- P1 will have recorded the receipt of message m' in this situation, but there is no recorded event of this message being transmitted. As a result, P1 must likewise be rolled back to a previous state. In this case, it turns out that the recovery line reflects the system's original state.
- Because processes take distinct checkpoints, this strategy is also known as independent checkpointing.
- An alternative solution is to globally coordinate checkpointing, which we will describe further below, however coordination necessitates global synchronization, which may cause performance issues.
- Another disadvantage of independent checkpointing is that each local store must be cleaned up on a regular basis, such as by running a specialized distributed garbage collector.
- The biggest drawback, however, is in calculating the recovery line.

Coordinated Checkpointing

- As the name implies, coordinated checkpointing synchronizes all processes to jointly write their state to local stable storage.
- The fundamental benefit of coordinated checkpointing is that the saved state is automatically globally consistent, avoiding cascaded rollbacks that cause the domino effect.
- A two-phase blocking protocol offers a simpler option. A coordinator sends a CHECKPOINT_REQUEST message to all processes first. When a process receives such a message, it takes a local checkpoint, queues any subsequent messages passed to it by the application it is executing, and notifies the coordinator that it has done so.
- When the coordinator receives acknowledgement from all processes, it broadcasts a CHECKPOINT_DONE message, allowing the (blocked) processes to proceed.
- Because no incoming message will ever be registered as part of a checkpoint, it is easy to understand how this strategy will result in a globally consistent state. This is because any message that follows a request to take a checkpoint is not considered part of the local checkpoint.
- Outgoing messages (as passed to the checkpointing process by the application) are queued locally until the CHECKPOINT_DONE message is received.
- This approach can be improved by multicasting a checkpoint request just to processes that rely on the coordinator's recovery and ignoring all other processes. A process is dependent on the coordinator if it has received a message that is causally related to a message issued by the coordinator since the last checkpoint. This gives rise to the concept of an incremental snapshot.
- To take an incremental snapshot, the coordinator sends a checkpoint request to just the processes to which it has sent a message since the last checkpoint. When a process P receives such a request, it transmits it to all processes to which P has delivered messages since the last checkpoint, and so on.
- A process only sends the request once. When all processes have been identified, a second multicast is used to initiate checkpointing and allow the processes to resume where they left off.

5.9.3 Message Logging

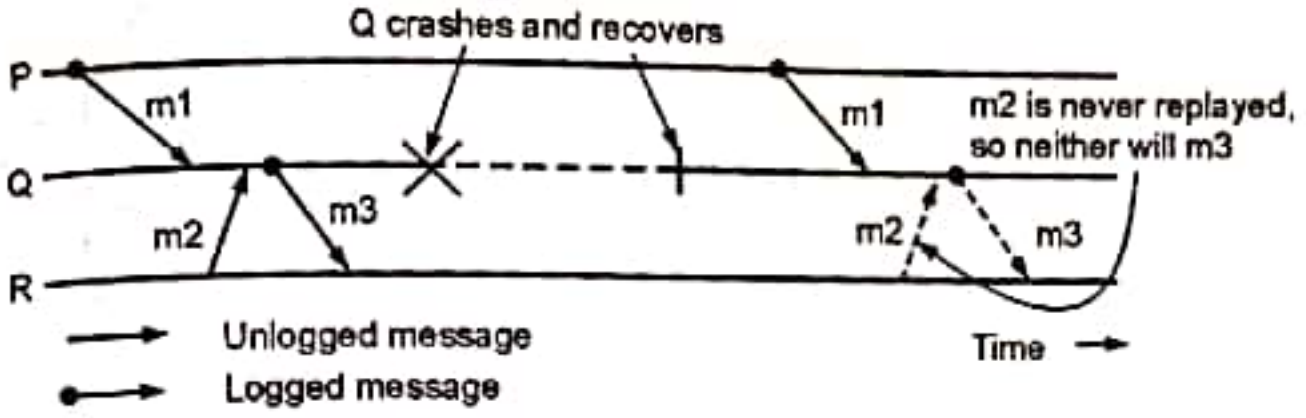
- Given that checkpointing is an expensive operation, particularly when it comes to the operations involved in writing state to stable storage, strategies have been developed to reduce the number of checkpoints while still allowing recovery. Message logging is an important approach in distributed systems.

The essential notion behind message logging is that if message transmission may be replayed, we can still achieve a globally consistent state without having to recover it from stable storage. Instead, a checkpointed state is used as a starting point, and any messages delivered since then are simply retransmitted and processed appropriately.

Given that message logs are required to recover from a process crash and restore a globally consistent state, it is critical to understand when messages should be logged. Many existing message-logging techniques can be easily described if we focus on how they handle orphan processes. An orphan process is one that survives the crash of another process but whose state after recovery differs from the crashed process.

Consider the situation depicted in Figure 5.9.3. Process Q gets messages m1 and m2 from processes P and R, and then delivers message m3 to R. Message m2, however, is not logged, unlike all previous messages.

If process Q crashes and then recovers, just the logged messages required for Q's recovery, in our case, m1, are replayed. Because m2 was not logged, its transmission will not be replayed, which means that m3's transmission may likewise fail.



(IE25)**Fig.5.9.3 : Incorrect replay of messages after recovery, leading to an orphan process**

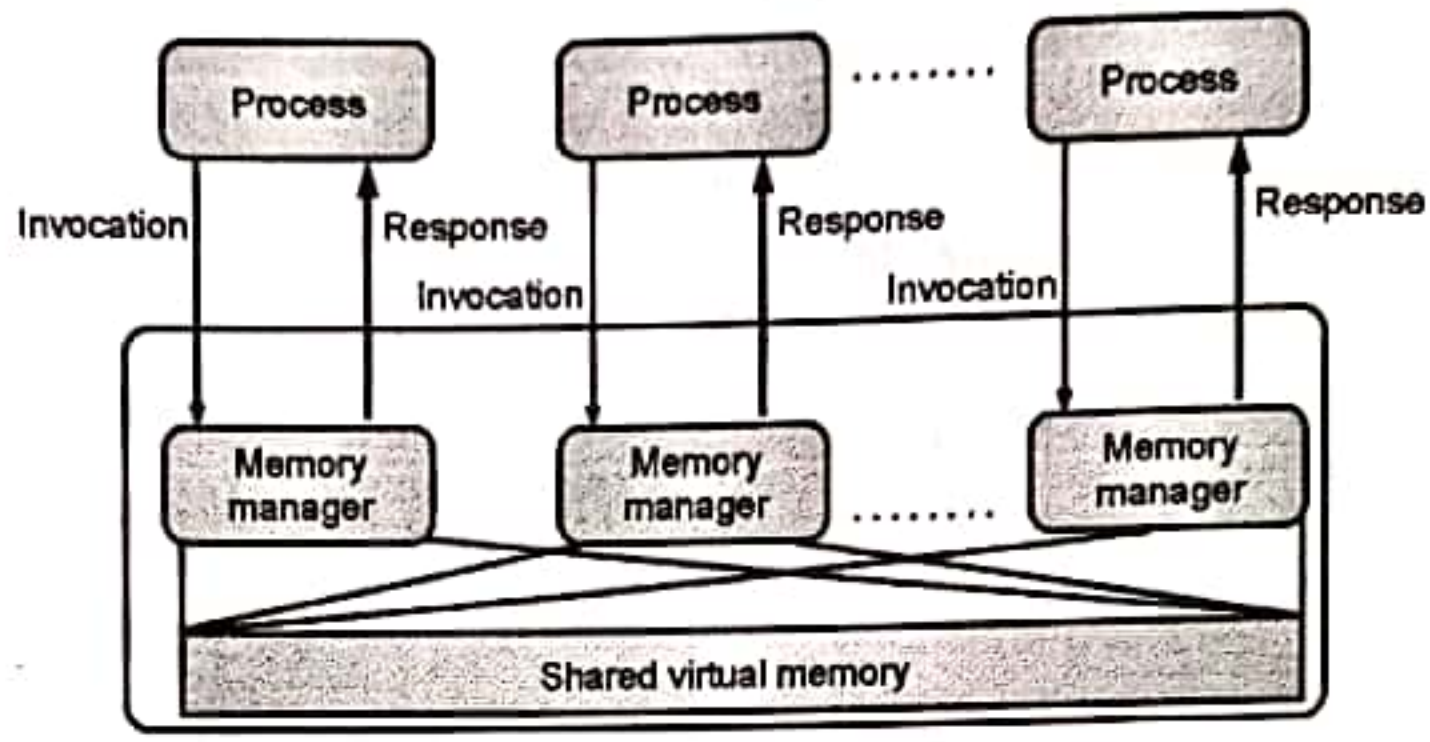
However, the situation following Q's recovery is incompatible with that which existed prior to its recovery. R, in the instance, holds a message (m3) that was sent before the crash but whose receipt and delivery do not occur while replaying what occurred prior to the crash. Such inconsistencies should be avoided at all times.

5.9.4 Recovery-Oriented Computing

- Another way to deal with recovery is to basically start over. This way of hiding failures is based on the idea that it may be much cheaper to optimise for recovery than to try to make systems that don't fail for a long time. This way of doing things is also called "recovery-oriented computing."
- Recovery-oriented computing comes in many forms. One option is to simply reboot (a part of a system), which has been tried to restart Internet servers. To be able to restart only a certain part of the system, the problem must be properly localized. At that point, rebooting just means deleting all instances of the identified components and the threads that are running on them, and (often) just restarting the related requests.
- To make rebooting a useful way to fix a problem, components need to be mostly independent of each other, which means that they can't depend on each other. If there are strong dependencies, fault localization and analysis may still require that the whole server be restarted. At that point, it may be more efficient to use traditional recovery methods like the ones we just talked about.
- Another way to do recovery-oriented computing is to use checkpointing and recovery techniques, but to keep running the program in an environment that has changed.
- The main idea here is that many failures can be easily avoided by giving programs more buffer space, clearing memory before allocating it, changing the order in which messages are sent (as long as it doesn't change the meaning of the message), and so on.
- The main idea is to fix software that doesn't work. Since software execution is very predictable, changing the environment in which it runs may save the day, but of course, it won't fix anything.

5.10 DISTRIBUTED SHARED MEMORY

- A distributed shared memory (DSM) system is a collection of many nodes/computers which are connected through some network, and all have their local memories.
- The DSM system manages the memory across all the nodes. All the nodes/computers transparently interconnect and process.



(1E26)**Fig.5.10.1 : Distributed Shared Memory**

- The DSM also makes sure that all the nodes are accessing the virtual memory independently without any interference from other nodes.
- The DSM does not have any physical memory, instead, a virtual space address is shared among all the nodes, and the transfer of data occurs among them through the main memory.

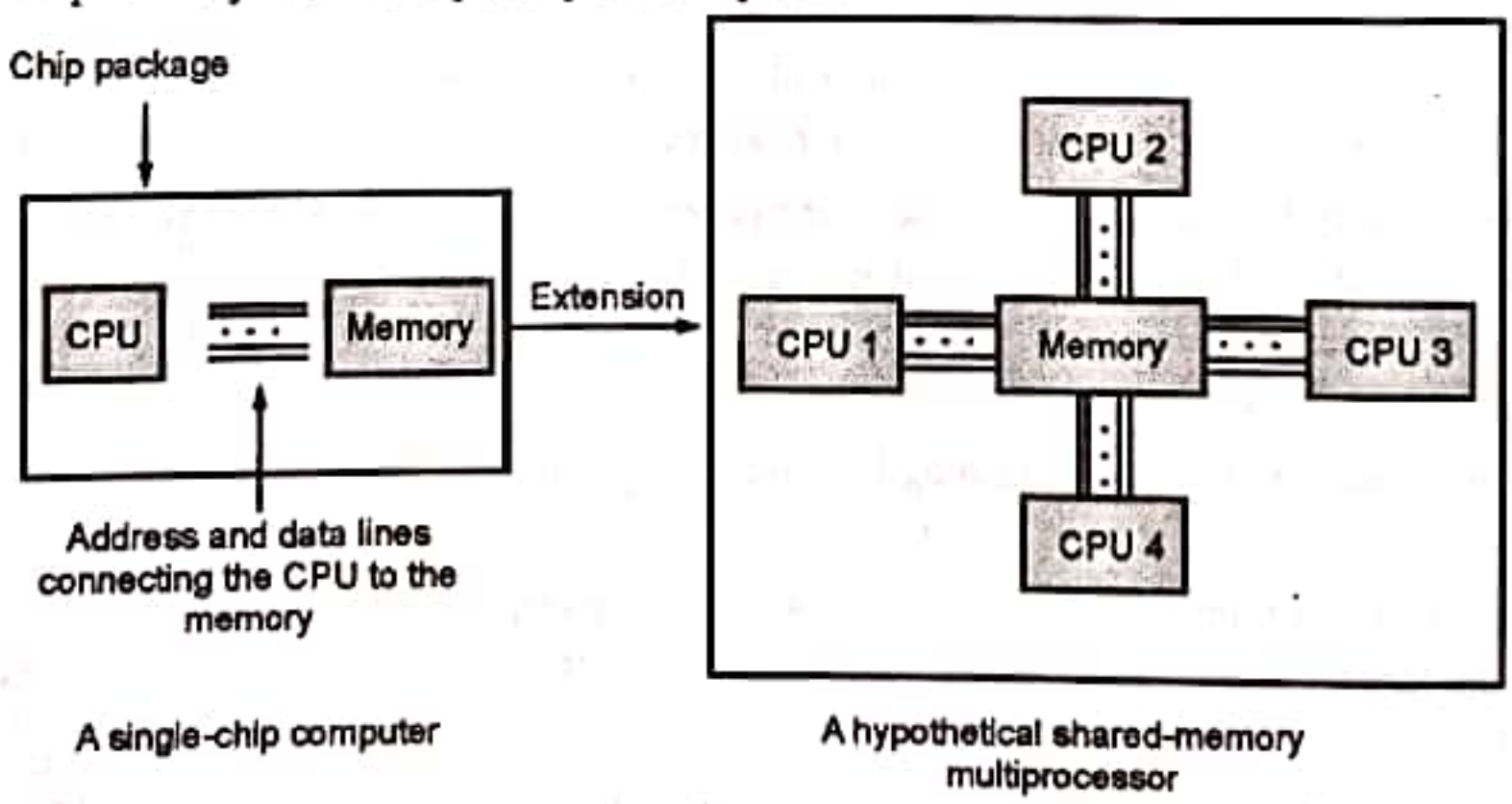
5.10.1 Types of Distributed Shared Memory

(1) On-Chip Memory

(i) All the data are stored in the CPU's chip.

(ii) There is a direct connection between Memory and address lines.

(iii) The On-Chip Memory DSM is very costly and complicated.



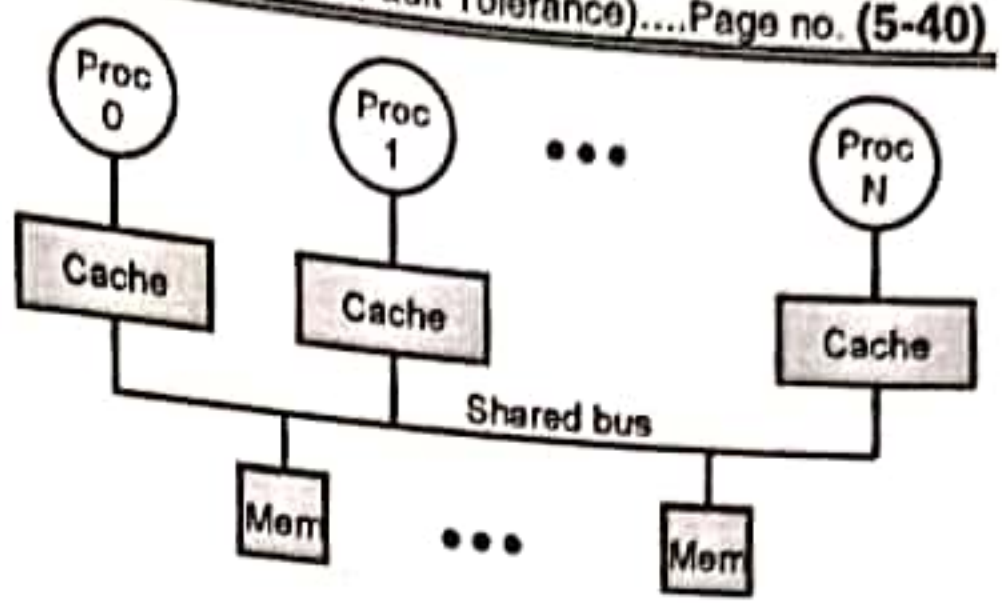
(1E27)**Fig. 5.10.2 : On-Chip Memory DSM**

Bus-Based Microprocessor

(i) The connection between the memory and the CPU is established through a number of parallel wires that acts as a bus.

(ii) All the computer follows some protocols to access the memory, and an algorithm is implemented to prevent memory access by the systems at the same time.

(iii) The network traffic is reduced by the cache memory.



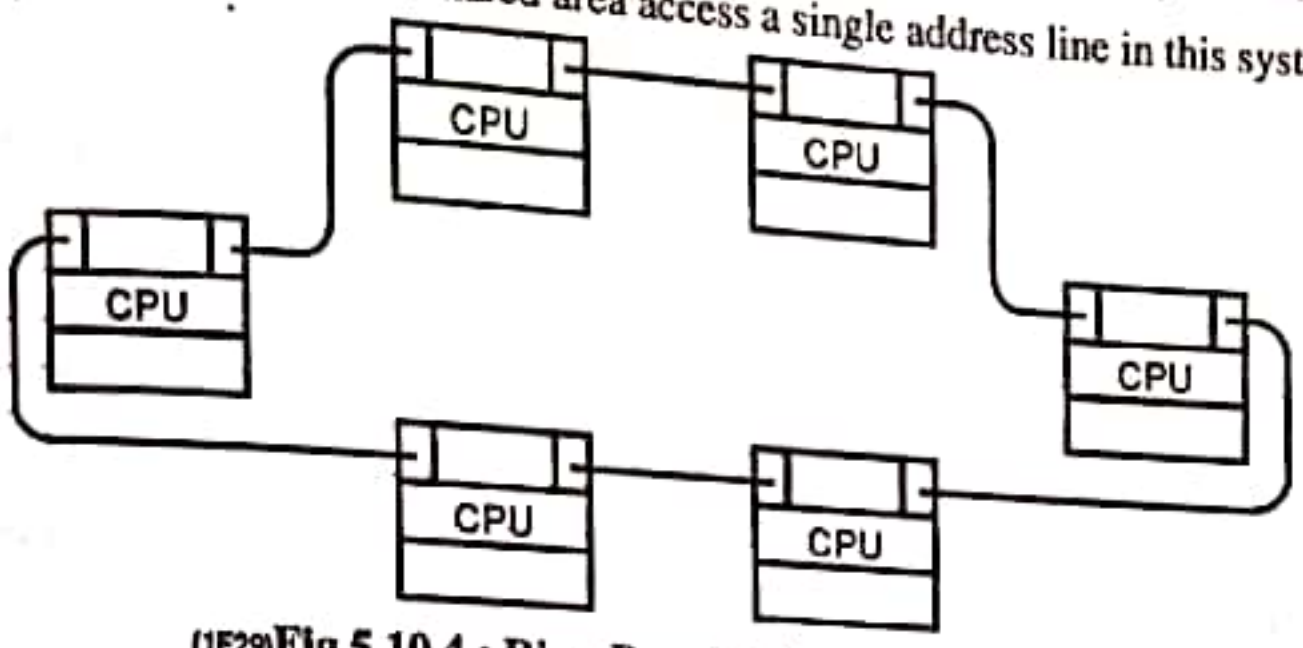
(1E28)**Fig.5.10.3 : Bus-Based Microprocessor DSM**

Ring-Based Microprocessor

(i) In Ring-based DSM, there is no centralized memory present

(ii) All the nodes are connected through some link/network, and accessing of memory is done by the token passing

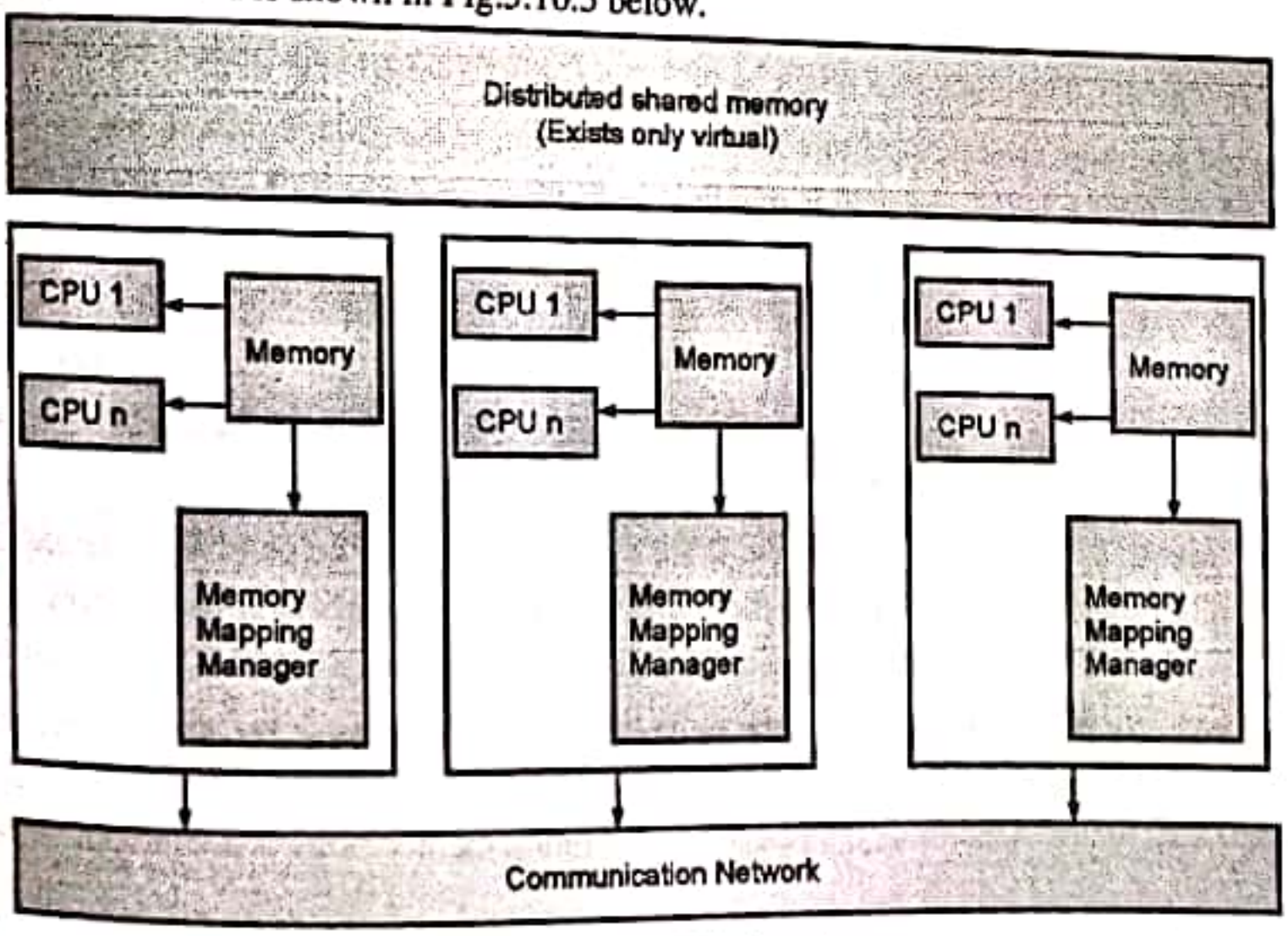
(iii) All the nodes/computers present in the shared area access a single address line in this system



(1E29)**Fig.5.10.4 : Ring-Based Microprocessor DSM**

5.10.2 Architecture of DSM

A general architecture of DSM is shown in Fig.5.10.5 below.



(1E30)**Fig.5.10.5 : DSM Architecture**

- Every node consists of one or additional CPU's and a memory unit.
- High-speed communication network is employed for connecting the nodes.
- A straightforward message passing system permits processes on completely different nodes to exchange one another.

Memory Mapping Manager Unit

(i) Memory mapping manager routine, in every node, maps the native memory onto the shared computer storage.

(ii) For mapping operation, the shared memory house is divided into blocks.

(iii) Information caching may be a documented answer to deal with operation latency.

(iv) DMA uses information caching to scale back network latency. The maximum memory of the individual nodes is employed to cache items of the shared memory house.

(v) Memory mapping manager of every node reads its native memory as an enormous cache of the shared memory house for its associated processors. The bass unit of caching may be a memory block.

(vi) Systems that support DSM, information moves between secondary memory and main memory also as between main reminiscences of various nodes.

Communication Network Unit

(i) Once method access information within the shared address house, mapping manager maps the shared memory address to the physical memory.

(ii) The mapped layer of code enforced either within the operating kernel or as a runtime routine.

(iii) Physical memory on every node holds pages of shared virtual–address house. Native pages area unit gift in some node's memory and remote pages in some other node's memory.

5.10.3 Design Issues in DSM Systems

DSM is a mechanism that manages memory across multiple nodes and makes inter-process communications transparent to end-users. To design information shared memory, we might deal with certain issues discussed below :

(1) **Granularity :** Granularity refers to the block size of a DSM system. Granularity refers to the unit of sharing and the unit of data moving across the network when a network block shortcoming then we can utilize the estimation of the block size as words/phrases. The block size might be different for the various networks.

(2) **Structure of shared memory space :** Structure refers to the design of the shared data in the memory. The structure of the shared memory space of a DSM system is regularly dependent on the sort of applications that the DSM system is intended to support.

(3) **Memory coherence and access synchronization :** In the DSM system the shared data things ought to be accessible by different nodes simultaneously in the network. The fundamental issue in this system is data irregularity. The data irregularity might be raised by the synchronous access. To solve this problem in the DSM system we need to utilize some synchronization primitives, semaphores, event count, and so on.

(4) **Data location and access :** To share the data in the DSM system it ought to be possible to locate and retrieve the data as accessed by clients or processors. Therefore, the DSM system must implement some form of data block finding system to serve network data to meet the requirement of the memory coherence semantics being utilized.

(5) **Replacement strategy :** In the local memory of the node is full, a cache miss at the node implies not just a get of the gotten to information block from a remote node but also a replacement. A data block of the local memory should be replaced by the new data block. Accordingly, a position substitution methodology is additionally vital in the design of a DSM system.

Thrashing : In DSM system data blocks move between nodes on demand. There is a probability that two nodes compete for write access to the single data item. The data relating data block might be moved to back and forth at such a high rate that no genuine work can get gone. The DSM system should utilize an approach to keep away from a situation generally known as thrashing.

Heterogeneity : The DSM system worked for homogeneous systems and need not address the heterogeneity issue. In any case, assuming the underlined system environment is heterogeneous, the DSM system should be designed to deal with heterogeneity, so it works appropriately with machines having different architectures.

Descriptive Questions

Explain and differentiate various data-centric consistency models.

Explain and differentiate various client-centric consistency models.

What is replication? Write the advantages of replication.

Explain replication as scaling technique.

Explain different failure models.

Explain Byzantine Agreement Problem with example.

Explain different failures that can occur in RPC and suggest their solutions.

Explain different types of ordering of the messages in group communication.

What is checkpointing? Explain independent and coordinated checkpointing approaches.

What is message logging? What are the advantages of message logging?

Explain recovery-oriented computing.

What is DSM? What are the design issues in DSM?

Chapter Ends...

□□□

MODULE 6

CHAPTER 6

Distributed File Systems

6.1 INTRODUCTION TO DISTRIBUTED FILE SYSTEM

- A **distributed file system (DFS)** is a file system that is distributed on various file servers and locations.
- It permits programs to access and store isolated data in the same method as in the local files.
- It also permits the user to access files from any system.
- It allows network users to share information and files in a regulated and permitted manner.
- Although, the servers have complete control over the data and provide users access control.
- DFS's primary goal is to enable users of physically distributed systems to share resources and information through the **Common File System (CFS)**.
- It is a file system that runs as a part of the operating systems.
- Its configuration is a set of workstations and mainframes that a LAN connects.
- The process of creating a namespace in DFS is transparent to the clients.
- DFS has two components in its services, and these are as follows:

(i) **Location Transparency :** Location Transparency is achieved through the namespace component.

(ii) **Redundancy :** Redundancy is done through a file replication component.

6.1.1 Features of DFS

Q. What are the desirable features of good distributed file systems? Explain file sharing semantic of it. **(MU - May 16)**

1. Transparency

- *Structure transparency*: There is no need for the client to know about the number or locations of file servers and the storage devices. Multiple file servers should be provided for performance, adaptability, and dependability.
- *Access transparency*: Both local and remote files should be accessible in the same manner. The file system should be automatically located on the accessed file and send it to the client's side.
- *Naming transparency*: There should not be any hint in the name of the file to the location of the file. Once a name is given to the file, it should not be changed during transferring from one node to another.
- *Replication transparency*: If a file is copied on multiple nodes, both the copies of the file and their locations should be hidden from one node to another.

(i) **User mobility :** It will automatically bring the user's home directory to the node where the user logs in.

(ii) **Performance :** Performance is based on the average amount of time needed to convince the client requests. This time covers the CPU time + time taken to access secondary storage + network access time. It is advisable that the performance of the Distributed File System be similar to that of a centralized file system.

(iii) **Simplicity and ease of use :** The user interface of a file system should be simple and the number of commands in the file should be small.

(iv) **High availability :** A Distributed File System should be able to continue in case of any partial failures like a link failure, a node failure, or a storage drive crash. A high authentic and adaptable distributed file system should have different and independent file servers for controlling different and independent storage devices.

(v) **Scalability :** Since growing the network by adding new machines or joining two networks together is routine, the distributed system will inevitably grow over time. As a result, a good distributed file system should be built to scale quickly as the number of nodes and users in the system grows. Service should not be substantially disrupted as the number of nodes and users grows.

(vi) **High reliability :** The likelihood of data loss should be minimized as much as feasible in a suitable distributed file system. That is, because of the system's unreliability, users should not feel forced to make backup copies of their files. Rather, a file system should create backup copies of key files that can be used if the originals are lost. Many file systems employ stable storage as a high-reliability strategy.

(vii) **Data Integrity :** Multiple users frequently share a file system. The integrity of data saved in a shared file must be guaranteed by the file system. That is, concurrent access requests from many users who are competing for access to the same file must be correctly synchronized using a concurrency control method. Atomic transactions are a high-level concurrency management mechanism for data integrity that is frequently offered to users by a file system.

(viii) **Security :** A distributed file system should be secure so that its users may trust that their data will be kept private. To safeguard the information contained in the file system from unwanted & unauthorized access, security mechanisms must be implemented.

(ix) **Heterogeneity**: Heterogeneity in distributed systems is unavoidable as a result of huge scale. Users of heterogeneous distributed systems have the option of using multiple computer platforms for different purposes.

6.1.2 File Sharing Semantics

- Semantics is a concept that is used by users to check file systems that are supporting file sharing in their systems.
- Basically, it is a specification to check how in a single system multiple users are getting access to the same file at the same time.
- They are used to check various things in files, like when will modification by some user in some file is noticeable to others.
- In Andrew file system, successful implementation of sharing semantics is found.
- To access the same file by a user process is always enclosed between open() and close() operations. When there is a series of access taking place for the same file, then it makes up a file session.
- Following are the file-sharing semantics:

(1) UNIX Semantics

The file systems in UNIX use the following consistency semantics:

- The file that which user is going to write will be visible to all users who are sharing that file at that time.
- There is one mode in UNIX semantics to share files via sharing pointer of the current location. But it will affect all other sharing users.

In this, a file that is shared is associated with a single physical image that is accessed as an exclusive resource. This single image causes delays in user processes.

(2) Session Semantics

The file system in Andrew uses the following consistency semantics.

- The file that the user is going to write will not be visible to all users who are sharing that file at that time.
- After closing the file, changes done to that file by the user are only visible only in sessions starting later. If the changed file is already opened by the other user, then changes will not be visible to that user.

In this, a file that is shared is associated with several images and there is no delay in this because it allows multiple users to perform both read and write accesses concurrently on their images.

(3) Immutable Shared File Semantics

In this, users are not allowed to modify the file, which is declared as shared by its creator. An Immutable file has two properties which are as follows:

- Its name may not be reused.
- Its content may not be altered.
- In this file system, the content of the file is fixed. The implementation of semantics in a distributed system is simple because sharing is disciplined.

6.2 FILE MODELS IN DISTRIBUTED FILE SYSTEM

This section discusses the concept of File Models in Distributed Systems. In Distributed File Systems (DFS), multiple ...dels are used to provide the file system's facility. Different file systems often employ different conceptual models. The ...dels based on structure and mobility are commonly used for the modeling of files.

Unstructured and Structured files

In the unstructured model, a file is an unstructured sequence of bytes. The interpretation of the meaning and structure of the data stored in the files is up to the application (e.g. UNIX and MS-DOS). Most modern operating systems use the unstructured file model.

In structured files (rarely used now) a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different sizes.

The structured files further consist of two types:

- **Files with Non-Indexed records :** In files with non-indexed records, the retrieving of records is performed concerning a position in the file. For example, the third record from the beginning, the third record from the last/end.
- **Files with Indexed records :** In files with indexed records, one or more key fields exist in each record, each of which can be addressed by providing its value. To locate records fast, a file is maintained as a B-tree or other equivalent data structure or hash table.

Mutable and Immutable files

Based on the modifiability criteria, files are of two types, mutable and immutable. Most existing operating systems use the mutable file model. An update performed on a file overwrites its old contents to produce the new content.

In the immutable model, rather than updating the same file, a new version of the file is created each time a change is made to the file contents and the old version is retained unchanged. The problems in this model are increased use of disk space and increased disk activity.

6.3 FILE ACCESSING MODELS

Q. Explain file accessing models. **(MU – Dec. 17, 18)**

The specific client's request for accessing a particular file is serviced on the basis of the file accessing model used by the distributed file system. The file accessing model basically depends on

(1) the method used for accessing remote files and (2) the unit of data access

Accessing remote files : A distributed file system may use one of the following models to service a client's file access request when the accessed file is remote :

(a) **Remote service model :** Processing of a client's request is performed at the server's node. Thus, the client's request for file access is delivered across the network as a message to the server, the server machine performs the access request, and the result is sent to the client. Need to minimize the number of messages sent and the overhead per message.

(b) **Data-caching model :** This model attempts to reduce the network traffic of the previous model by caching the data obtained from the server node. This takes advantage of the locality feature of the found in file accesses. A replacement policy such as LRU is used to keep the cache size bounded. While this model reduces network traffic it has to deal with the cache coherency problem during writes, because the local cached copy of the data needs to be updated, the original file at the server node needs to be updated and copies in any other caches need to be updated.

☞ **Advantage of Data-caching model over the Remote service model**

The data-caching model offers the possibility of increased performance and greater system scalability because it reduces network traffic, contention for the network, and contention for the file servers. Hence almost all distributed file systems implement some form of caching.

Example : NFS uses the remote service model but adds caching for better performance.

(2) **Unit of Data Transfer :** In file systems that use the data-caching model, an important design issue is to decide the unit of data transfer. This refers to the fraction of a file that is transferred to and from clients as a result of single read or write operation.

(a) **File-level transfer model :** In file-level transfer model, the complete file is moved while a particular operation necessitates the file data to be transmitted all the way through the distributed computing network amongst client and server. This model has better scalability and is efficient. This model requires sufficient storage space on the client machine. This approach fails for very large files, especially when the client runs on a diskless workstation. If only a small fraction of a file is needed, moving the entire file is wasteful.

(b) **Block-level transfer model :** In block-level transfer model, file data transfers through the network amongst client and a server is accomplished in units of file blocks. In short, the unit of data transfer in block-level transfer model is file blocks. The block-level transfer model might be used in distributed computing environment comprising several diskless workstations. When an entire file is to be accessed, multiple server requests are needed, resulting in more network traffic and more network protocol overhead.NFS uses block-level transfer model.

(c) **Byte-level transfer model :** In byte-level transfer model, file data transfers the network amongst client and a server is accomplished in units of bytes. In short, the unit of data transfer in byte-level transfer model is bytes. The byte-level transfer model offers more flexibility in comparison to the other file transfer models since, it allows retrieval and storage of an arbitrary sequential subrange of a file. The major disadvantage of byte-level transfer model is the trouble in cache management because of the variable-length data for different access requests.

(d) **Record-level transfer model :** The record-level file transfer model might be used in the file models where the file contents are structured in the form of records. In record-level transfer model, file data transfers through the network amongst client and a server is accomplished in units of records. The unit of data transfer in record-level transfer model is record.

6.4 FILE-CACHING SCHEMES

UQ. Discuss file caching for Distributed Algorithm. (MU - Dec. 17)

UQ. Write short note on File Caching Schemes. (MU - May 17, 22)

Every distributed file system uses some form of caching. The reasons are :

(1) **Better performance** since repeated accesses to the same information is handled additional network accesses and disk transfers. This is due to locality in file access patterns.

(2) It contributes to the **scalability and reliability** of the distributed file system since data can be remotely cached on the client node.

Key decisions to be made in file-caching scheme for distributed systems:

(1) Cache location

(2) Modification Propagation

(3) Cache Validation

Cache Location : This refers to the place where the cached data is stored. Assuming that the original location of a file is at its server's disk, there are three possible cache locations in a distributed file system:

(a) **Server's Main Memory:** A cache located in the server's main memory eliminates the disk access cost on a cache hit which increases performance compared to no caching.

Advantages

(i) Easy to implement

(ii) Totally transparent to clients

(iii) Easy to keep the original file and the cached data consistent.

(b) **Client's Disk :** In this case a cache hit costs one disk access. This is somewhat slower than having the cache in server's main memory. Having the cache in server's main memory is also simpler.

Advantages

(i) Provides reliability against crashes since modification to cached data is lost in a crash if the cache is kept in main memory.

(ii) Large storage capacity.

(iii) Contributes to scalability and reliability because on a cache hit the access request can be serviced locally without the need to contact the server.

(c) **Client's Main Memory :** Eliminates both network access cost and disk access cost. This technique is not preferred to a client's disk cache when large cache size and increased reliability of cached data are desired.

Advantages

(i) Maximum performance gain. (ii) Permits workstations to be diskless.

(iii) Contributes to reliability and scalability.

Modification Propagation : When the cache is located on client's nodes, a file's data may simultaneously be cached on multiple nodes. It is possible for caches to become *inconsistent* when the file data is changed by one of the clients and the corresponding data cached at other nodes are not changed or discarded.

There are two design issues involved:

(1) When to propagate modifications made to a cached data to the corresponding file server.

(2) How to verify the validity of cached data.

The modification propagation scheme used has a critical effect on the system's performance and reliability. Techniques used include :

(a) **Write-through scheme:** When a cache entry is modified, the new value is immediately sent to the server for updating the master copy of the file.

Advantage

- High degree of reliability and suitability for UNIX-like semantics.
- The risk of updated data getting lost in the event of a client crash is low.

☞ **Disadvantage**

- This scheme is only suitable where the ratio of read-to-write accesses is fairly large. It does not reduce network traffic for writes.
- This is due to the fact that every write access has to wait until the data is written to the master copy of the server. Hence the advantages of data caching are only read accesses because the server is involved for all write accesses.

(b) **Delayed-write scheme**: To reduce network traffic for writes the delayed-write scheme is used. In this case, the new data value is only written to the cache and all updated cache entries are sent to the server at a later time.

There are three commonly used delayed-write approaches:

(i) **Write on ejection from cache**: Modified data in cache is sent to server only when the cache-replacement policy has decided to eject it from client's cache. This can result in good performance but there can be a reliability problem since some server data may be outdated for a long time.

(ii) **Periodic write**: The cache is scanned periodically and any cached data that has been modified since the last scan is sent to the server.

(iii) **Write on close**: Modification to cached data is sent to the server when the client closes the file. This does not help much in reducing network traffic for those files that are open for very short periods or are rarely modified.

☞ **Advantages of the delayed-write scheme**

- Write accesses complete more quickly because the new value is written only client cache. This results in a performance gain.
- Modified data may be deleted before it is time to send them to the server (e.g. temporary data). Since modifications need not be propagated to the server this results in a major performance gain.
- Gathering all file updates and sending them together to the server is more efficient than sending each update separately.

☞ **Disadvantage of the delayed-write scheme**

- Reliability can be a problem since modifications not yet sent to the server from a client's cache will be lost if the client crashes.

(a) **Cache Validation schemes :** The modification propagation policy only specifies when the master copy of a file on the server node is updated upon modification of a cache entry. It does not tell anything about when the file data residing in the cache of other nodes is updated.

A file data may simultaneously reside in the cache of multiple nodes. A client's cache entry becomes stale as soon as some other client modifies the data corresponding to the cache entry in the master copy of the file on the server.

It becomes necessary to verify if the data cached at a client node is consistent with the master copy. If not, the cached data must be invalidated and the updated version of the data must be fetched again from the server.

There are two approaches to verify the validity of cached data: the client-initiated approach and the server-initiated approach.

(1) **Client-initiated approach :** The client contacts the server and checks whether its locally cached data is consistent with the master copy. Two approaches may be used:

(i) **Checking before every access :** This defeats the purpose of caching because the server needs to be contacted on every access.

(ii) **Periodic checking :** A check is initiated every fixed interval of time.

Disadvantage of client-initiated approach: If frequency of the validity check is high, the cache validation approach generates a large amount of network traffic and consumes precious server CPU cycles.

Server-initiated approach: A client informs the file server when opening a file, indicating whether a file is being opened for reading, writing, or both. The file server keeps a record of which client has which file open and in what mode. So server monitors file usage modes being used by different clients and reacts whenever it detects a potential for inconsistency. E.g., if a file is open for reading, other clients may be allowed to open it for reading, but opening it for writing cannot be allowed. So also, a new client cannot open a file in any mode if the file is open for writing.

When a client closes a file, it sends intimation to the server along with any modifications made to the file. Then the server updates its record of which client has which file open in which mode.

When a new client makes a request to open an already open file and if the server finds that the new open mode conflicts with the already open mode, the server can deny the request, queue the request, or disable caching by asking all clients having the file open to remove that file from their caches.

Note: On the web, the cache is used in read-only mode so cache validation is not an issue.

Disadvantage of server-initiated approach

It requires that file servers be stateful. Stateful file servers have a distinct disadvantage over stateless file servers in the event of a failure.

6.5 FILE REPLICATION

High availability is a desirable feature of a good distributed file system and *file replication* is the primary mechanism for improving file availability.

A *replicated file* is a file that has multiple copies, with each file on a separate file server.

6.5.1 Difference Between Replication and Caching

A replica of a file is associated with a server, whereas a cached copy is normally associated with a client.

The existence of a cached copy is primarily dependent on the locality in file access patterns, whereas the existence of a replica normally depends on availability and performance requirements.

As compared to a cached copy, a replica is more persistent, widely known, secure, available, complete, and accurate.

A cached copy is contingent upon a replica. Only by periodic revalidation with respect to a replica can a cached copy be useful.

6.5.2 Advantages of Replication

Increased Availability : Alternate copies of a replicated data can be used when the primary copy is unavailable.

Increased Reliability : Due to the presence of redundant data files in the system, recovery from catastrophic failures (e.g. hard drive crash) becomes possible.

Improved response time : It enables data to be accessed either locally or from a node to which access time is lower than the primary copy access time.

Reduced network traffic : If a file's replica is available with a file server that resides on a client's node, the client's access request can be serviced locally, resulting in reduced network traffic.

Improved system throughput : Several client's request for access to a file can be serviced in parallel by different servers, resulting in improved system throughput.

Better scalability : Multiple file servers are available to service client requests since due to file replication. This improves scalability.

6.5.3 Replication Transparency

- Replication of files should be transparent to the users so that multiple copies of a replicated file appear as a single logical file to its users. This calls for the assignment of a single identifier/name to all replicas of a file.
- In addition, replication control should be transparent, i.e., the number and locations of replicas of a replicated file should be hidden from the user. Thus replication control must be handled automatically in a user-transparent manner.

6.5.4 Multicopy Update Problem

Maintaining consistency among copies when a replicated file is updated is a major design issue of a distributed file system that supports file replication.

1. **Read-only replication :** In this case the update problem does not arise. This method is too restrictive.
2. **Read-Any-Write-All Protocol :** A read operation on a replicated file is performed by reading any copy of the file and a write operation by writing to all copies of the file. Before updating any copy, all copies need to be locked, then they are updated, and finally the locks are released to complete the write.
3. **Disadvantage :** A write operation cannot be performed if any of the servers having a copy of the replicated file is down at the time of the write operation.
4. **Available-Copies Protocol :** A read operation on a replicated file is performed by reading any copy of the file and a write operation by writing to all available copies of the file. Thus if a file server with a replica is down, its copy is not updated. When the server recovers after a failure, it brings itself up to date by copying from other servers before accepting any user request.
5. **Primary-Copy Protocol :** For each replicated file, one copy is designated as the primary copy and all the others are secondary copies. Read operations can be performed using any copy, primary or secondary. But write operations are performed only on the primary copy. Each server having a secondary copy updates its copy either by receiving notification of changes from the server having the primary copy or by requesting the updated copy from it. E.g., for UNIX-like semantics, when the primary-copy server receives an update request, it immediately orders all the secondary-copy servers to update their copies. Some form of locking is used and the write operation completes only when all the copies have been updated. In this case, the primary-copy protocol is simply another method of implementing the read-any-write-all protocol.

6.6 CASE STUDY ON NETWORK FILE SYSTEM (NFS)

- NFS is an abbreviation of the Network File System. It is a protocol of a distributed file system. This protocol was developed by the Sun Microsystems in the year of 1984. It is an architecture of the client/server, which contains a client program, server program, and a protocol that helps for communication between the client and server.
- It is that protocol which allows the users to access the data and files remotely over the network. Any user can easily implement the NFS protocol because it is an open standard. Any user can manipulate files as same as if they were on like other protocols. This protocol is also built on the ONC RPC system.
- This protocol is mainly implemented on computing environments where the centralized management of resources and data is critical. It uses the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) for accessing and delivering the data and files.
- Network File System is a protocol that works on all the networks of IP-based. It is implemented in that client/server application in which the server of NFS manages the authorization, authentication, and clients. This protocol is used with Apple Mac OS, Unix, and Unix-like operating systems such as Solaris, Linux, FreeBSD, AIX.

6.6.1 Benefits of Using NFS

- Multiple clients can use the same files, which allows everyone on the network to use the same data, accessing it on remote hosts as if it were acceding local files.
- Computers share applications, which eliminates the need for local disk space and reduces storage costs.
- All users can read the same files, so data can remain up-to-date, and it's consistent, and reliable.
- Mounting the file system is transparent to all users.
- Support for heterogeneous environments allows you to run mixed technology from multiple vendors and use interoperable components.
- System admin overhead is reduced due to the centralization of data.
- Fewer removable disks and drives laying around provides a reduction of security concerns—which is always good!

6.6.2 Working of Network File System

- Fundamentally, the NFS client-server protocol begins with a "**mount**" command, which specifies client and server software options or attributes.
- NFSv4 is a stateful protocol, with over 30 unique options that can be specified on the mount command ranging from read/write block size, and protocol used. Security protocols validate client access to data files as well as data security options, etc.
- Some of the more interesting NFS protocol software options include caching options, shared file locking characteristics, and security support.
- File locking and caching interact together, and both must be properly specified for shared file access to work.
- If file (read or write) data only resides in a host cache and some other host tries to access the same file, the data it reads could be wrong, unless both (or rather) all clients of the NFS storage server use the SAME locking options and caching options for the mounted file system.
- File locking was designed to support shared file access. That is when a file is accessed by more than one application or (compute) thread. Shared file access could be occurring within a single host (with or without multi-core/multi-thread) or across different hosts accessing the same file over NFS.

6.6.3 Disadvantages of Network File System

There are many challenges with the current NFS Internet Standard that may or may not be addressed in the future; for example, some reviews of NFSv4 and NFSv4.1 suggest that these versions have limited bandwidth and scalability (improved with NFSv4.2) and that NFS slows down during heavy network traffic. Here are some others :

(1) **Security :** First and foremost is a security concern, given that NFS is based on RPCs which are inherently insecure and should only be used on a trusted network behind a firewall. Otherwise, NFS will be vulnerable to internet threats.

(2) **Protocol chattiness :** The NFS client-server protocol requires a lot of request activity to be set up to transfer data. The NFS protocol requires many small interactions or steps to read and write data, which equates to a ton of overhead for someone actively interacting with today's AI/ML/DL workloads that consume a tremendous number of small files.

(3) **File sharing is highly complex :** Configuring and setting up proper shared file access via file locking and caching is a daunting task at best. On the one hand, it adds a lot of the protocol overhead, leading to the chattiness mentioned above. On the other hand, it still leaves a lot to be desired, inasmuch any each host's mount command for the same file system can easily go awry.

(4) **Parallel file access :** NFS was designed as a way to sequentially access a shared network file, but these day's applications are dealing with larger files and non-sequential or parallel file access is required. This was added to NFSv4, but not a lot of clients support it yet.

(5) **Block size limitations :** The current NFS protocol standard allows for a maximum of 1MB of data to be transferred during one read or write request. In 1984, 1MB was a lot of data, but that's no longer the case. There are classes of applications that should be transferring GBs not MBs of data.

6.7 CASE STUDY ON ANDREW FILE SYSTEM (AFS)

UQ. Write short note on: Andrew File System. **(MU - Dec. 16, 19)**

- Andrew File System (AFS) is a distributed network file system developed by Carnegie Mellon University.
- Enterprises use an AFS to facilitate stored server file access between AFS client machines located in different areas.
- AFS supports reliable servers for all network clients accessing transparent and homogeneous namespace file locations.
- An AFS may be accessed from a distributed environment or location independent platform.
- A user accesses an AFS from a computer running any type of OS with Kerberos authentication and single namespace features.
- Users share files and applications after logging into machines that interact within the Distributed Computing Infrastructure (DCI).
- In distributed networks, an AFS relies on local cache to enhance performance and reduce workload. For example, a server responds to a workstation request and stores the data in the workstation's local cache. When the workstation requests the same data, the local cache fulfils the request.
- AFS networks employ server and client components, as follows:
  (i) A client may be any type of machine that generates requests for AFS server files stored on a network.
  (ii) After a server responds and sends a requested file, the file is stored in the client machine's local cache and presented to the client machine user.
  (iii) When a user accesses the AFS, the client uses a call-back mechanism to send all changes to the server. Frequently used files are stored for quick access in the client machine's local cache.
- Files are transferred to client machines as necessary and cached on local disk. The server part of AFS is called the AFS File Server, and the client part of AFS is called the AFS Cache Manager.
- AFS provides Access Control Lists (ACLs) which provide for more control and flexibility than standard Linux file permissions.
- AFS provides transparent access to local and remote files by using a consistent name space.
- All files in AFS are found under the Linux directory /afs. Under the /afs directory are the various sites which run AFS and make their file-system available to the Internet community. These sites are called AFS Cells.
- AFS equips users with multiple access control permissions, as follows:
  (i) **Look Up (l) :** Users may access and list AFS directory and subdirectory content and review a directory's access control list (ACL).
  (ii) **Insert (i) :** Users may add new subdirectories or files.
  (iii) **Delete (d) :** Users may remove directory files.
  (iv) **Administer (a) :** Users may modify the home directory's ACL.

(v) **Read (r) :** Users may view file directory or subcategory contents, as AFS supports the standard Unix Owner Read permission control set.

(vi) **Write (w) :** Users may modify or write files, as AFS supports the Unix Owner Write permission control set.

(vii) **Look (k) :** Processors may use the directory to execute programs requiring Flock files.

6.8 NAMING SERVICES AND DOMAIN NAME SYSTEM

- In a distributed system, names are used to refer to a wide variety of resources (computers, services, remote objects and files, ...).
- Names facilitate communication and resource sharing.
- Processes cannot share particular resources managed by a computer system unless they can name them consistently.
- Users cannot communicate with one another via a distributed system unless they can name one another, for example, with email addresses.
- Any process that requires access to a specific resource must possess a name for it.
  - file names: /etc/passwd
  - URLs: http://www.cdk4.net
  - Internet domain names: www.cdk4.net

6.8.1 Name and Address

- **Name :** an identifier permanently associated with an object, independent of its location within the distributed system. A "name" is how an endpoint is referenced. Typically, there is no structurally significant hierarchy. E.g. "David", "Tokyo", "itu.int"
- **Address :** an identifier associated with the current location of the object. An address is how you get to an endpoint. Typically, it is hierarchical (for scaling) like 950 Charter Street, Redwood City CA.

  Example : Email addresses usually have to change when they move between organizations or ISPs; they are not enough in themselves to refer to a specific individual over time.

6.8.2 Binding and Attribute

- We say that a name is resolved when it is translated into data about the named resource or object, often in order to invoke an action upon it. The association between a name and an object is called a **binding**.
- In general, names are bound to attributes of the named objects, rather than the implementation of the object themselves. An **attribute** is the value of a property associated with an object (for instance, its address).
- Example: "www.imm.dtu.dk" (name) is bound to the IP address (attribute) "192.38.82.230" of the IMM Web server (resource).

6.8.3 Naming Services

- A name service stores a collection of one or more naming contexts.
- **Naming contexts :** They are the sets of bindings between textual names and attributes for objects (such as users, computers, services and remote objects).
- Major operation of a name service is**name resolution**, that is to look up attributes from a given name.
- Operations are also required for creating new bindings, deleting bindings and listing bound names and adding and deleting contexts.

- **Example :** Domain Name System (DNS) names objects across the Internet (maps human-friendly hostnames into ip addresses)

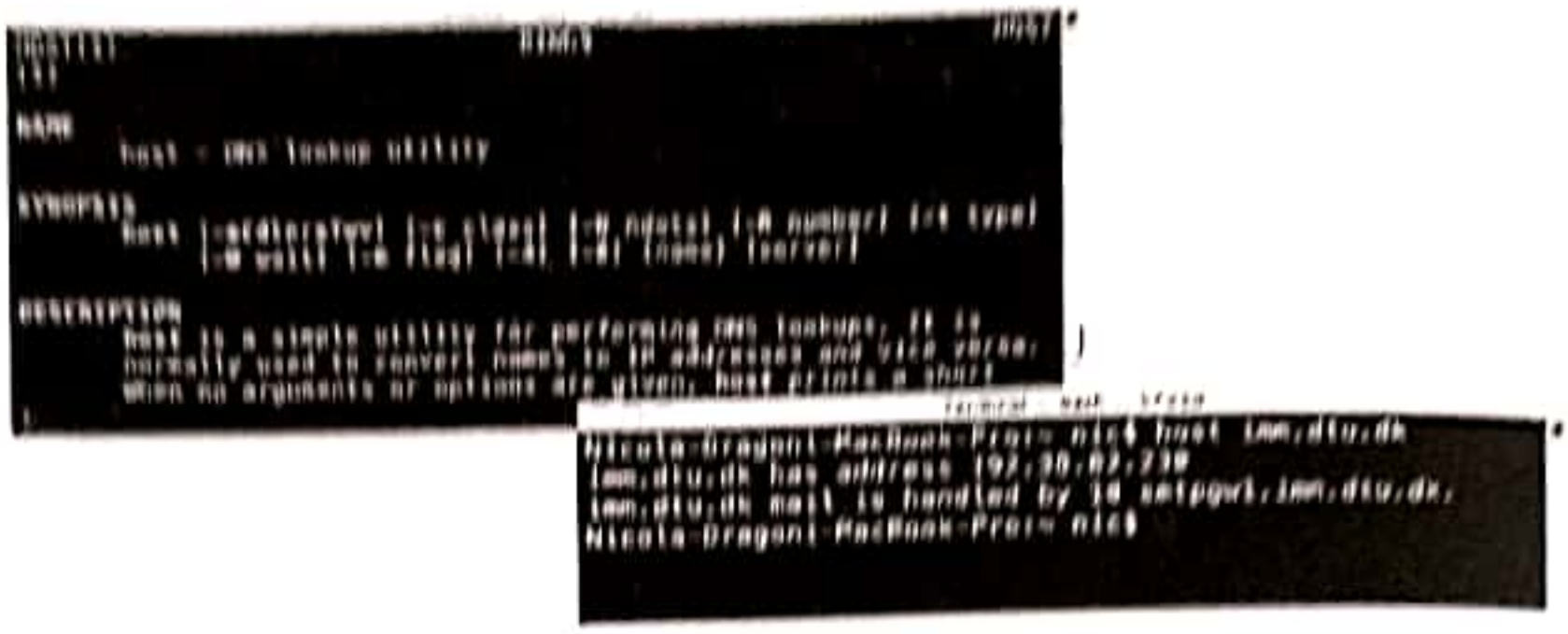


Fig. 6.8.1 : DNS Example

6.8.4 Name Spaces

- Name space is a collection of all valid names recognized by a particular service. Example: file systems (/etc/passwd is different from /oldetc/passwd).
- Names may have an internal structure that represents their position in a hierarchic name space.
- Most important advantage of hierarchic name spaces: each part of a name is resolved relative to a separate context, and the same name may be used with different meanings in different contexts.

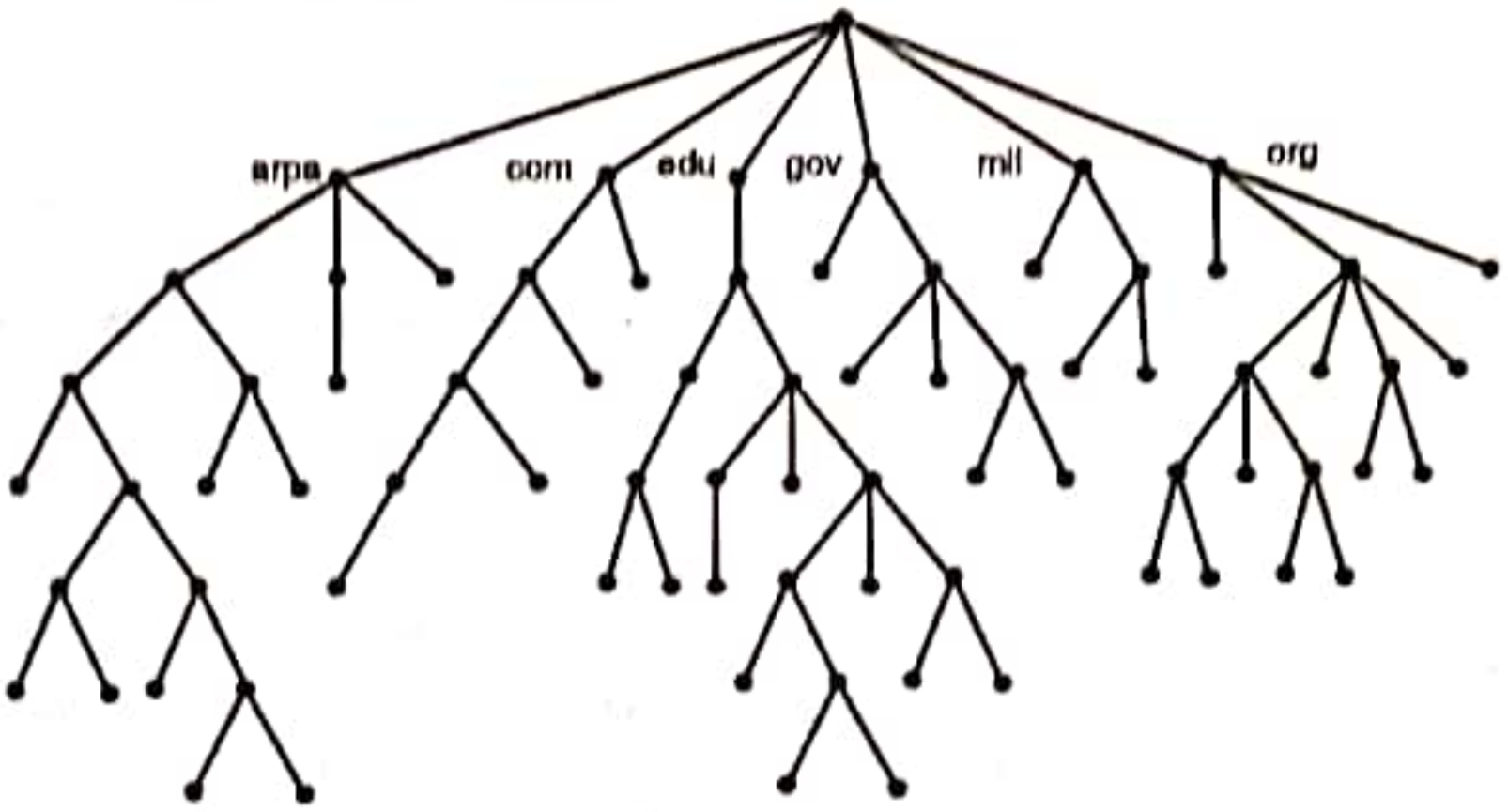


Fig. 6.9.2 : DNS Namespace

- **Example : DNS**
  - DNS names are called **domain names**. Example: www.cdk4.net (computer), net, com, ac.uk (domains).
  - The Domain Name System (DNS) is a distributed database system that translates domain names to numerical Internet Protocol (IP) addresses. It is an essential part of Internet infrastructure.
  - The DNS name space has a hierarchic structure: a domain name consists of one or more strings called name components or labels, separated by the delimiter ".".
  - The **name components** are non-null printable strings that do not contain ".".
  - In general, a **prefix** of a name is an initial section of the name that contains only zero or more entire components. Example: www and www.cdk4 are both prefixes of www.cdk4.net.
  - DNS names are not case-sensitive (www.cdk4.net == WWW.CDK4.NET)

- The DNS has three components: authoritative servers, recursive servers and clients (resolvers).
- A client (resolver) is installed in any computer with TCP/IP software. A client is any computer with Internet access capability.
- **Working of DNS**

(i) The client software initiates a DNS resolution (in response to a request by another software running on the computer) to access another computer using its name. Configuring a resolver is easy and all that it requires is the IP address of a recursive server.

(ii) The recursive server receives DNS queries from the resolvers and is responsible for executing all the resolution steps needed to provide the final answer to the client (resolver). The recursive servers query multiple servers until they obtain the answer to the query.

(iii) Finally, the authoritative server has the authority over a given domain name. It is the trusted source of information for a domain name. It contains the information on the IP addresses that are associated with the computers using the domain name in question.

(iv) A domain name resolution is initiated by the client software and the query is then sent to a recursive server. This other server generally initiates the resolution by querying the root servers or those servers responsible for the highest level of the domain name hierarchy.

(v) Based on the information received from the root servers, the recursive server queries other servers until it reaches the server responsible for the desired domain name.

6.8.5 Naming Domains

- Naming domain is the name space for which there exists a single overall administrative authority for assigning names within it.
- This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task.

6.8.6 Name Resolution

- In general, resolution is an iterative process whereby a name is repeatedly presented to naming contexts.
- A naming context either maps
  - a given name onto a set of primitive attributes directly
  - or it maps it onto a further naming context and a derived name to be represented to that context.
- To resolve a name:
  - it is first presented to some initial naming context
  - resolution iterates as long as further contexts and derived names are output.
- **Example :** UNIX File System
  - In the case of file systems, each directory represents a context.
  - /etc/passwd is a hierarchic name with two components.
    - The first, "etc" is resolved relative to the context "/" or root.
    - The second part, "passwd", is relative to the context "/etc".
  - The name /oldetc/passwd can have a different meaning because its second component is resolved in a different context.
  - Similarly, the same name /etc/passwd may resolve to different files in the contexts of two different computers.

6.9 DIRECTORY SERVICES

- Directory services are software systems that store, organize and provide access to directory information in order to unify network resources.
- Directory services map the network names of network resources to network addresses and define a naming structure for networks.
- The directory service provides transparency to protocols and network topology, permitting users to access resources without having to be aware of the physical location of the devices.
- It's an important component of the network operating system and is a central information repository for a service delivery platform.
- Directory services are network services that identify every resource such as email address, peripheral devices and computers on the network, and make these resources accessible to users and applications.
- Specific directory services called naming services map the names of resources in the network to the respective network address. This directory service relieves users from having to know the physical addresses of network resources. Directory services also define namespaces for networks, which hold one or more objects as name entries.
- Directory services hold shared information infrastructure to administer, manage, locate and organize common items and network resources. It is also a vital component of network operating systems.
- Two of the most widely used directory services are **Lightweight Directory Access Protocol**, which is used for email addresses, and **Netware directory service**, which is used in Novell Netware networks.
- Directory services generally have the following characteristics:

1. **Hierarchical naming model :** A hierarchical naming model uses the concept of containment to reduce ambiguity between names and simplify administration. The name for most objects in the directory is relative to the name of some other object which conceptually contains it. For example, the name of an object representing an employee of a particular company contains the name of the object representing the company, and the name of the company might contain the name of the objects representing the country where the company operates, e.g., cn=John Smith, o=Example Corporation, c=US. Together the names of all objects in the directory service form a tree, and each Directory Server holds a branch of that tree, which in the Sun Java System Directory Server documentation is also referred to as a suffix.
2. **Extended search capability :** Directory services provide robust search capabilities, allowing searches on individual attributes of entries.
3. **Distributed information model :** A directory service enables directory data to be distributed across multiple servers within a network.
4. **Shared network access :** While databases are defined in terms of APIs, directories are defined in terms of protocols. Directory access implies network access by definition. Directories are designed specifically for shared access among applications. This is achieved through the object-oriented schema model. By contrast, most databases are designed for use only by particular applications and do not encourage data sharing.
5. **Replicated data :** Directories support replication (copies of directory data on more than one server) which make information systems more accessible and more resistant to failure.
6. **Datastore optimized for reads :** The storage mechanism in a directory service is generally designed to support a high ratio of reads to writes.
7. **Extensible schema :** The schema describes the type of data stored in the directory. Directory services generally support the extension of schema, meaning that new data types can be added to the directory.

6.10 CASE STUDY ON GLOBAL NAME SERVICE (GNS)

- Designed and implemented by Lampson and colleagues at the DEC Systems Research Center (1986).
- Provide facilities for resource location, email addressing and authentication.
- When the naming database grows from small to large scale, the structure of name space may change and the service should accommodate it.
- The GNS manages a naming database that is composed of a tree of directories holding names and values.
- Directories are named by multi-part pathnames referred to a root, or relative to a working directory, much like file names in a UNIX file system.
- Each directory is also assigned an integer, which serves as a unique directory identifier (DI).
- A directory contains a list of names and references.
- The values stored at the leaves of the directory tree are organized into value trees, so that the attributes associated with names can be structured values.
- Names in the GNS have two parts: <directory name, value name>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree.
- **GNS Structure**
  - Tree of directories holding names and values
  - Muti-part pathnames refer to the root or relative working directory (like Unix file system)
  - Unique Directory Identifier (DI)
  - A directory contains list of names and references
  - Leaves of tree contain value trees (structured values)

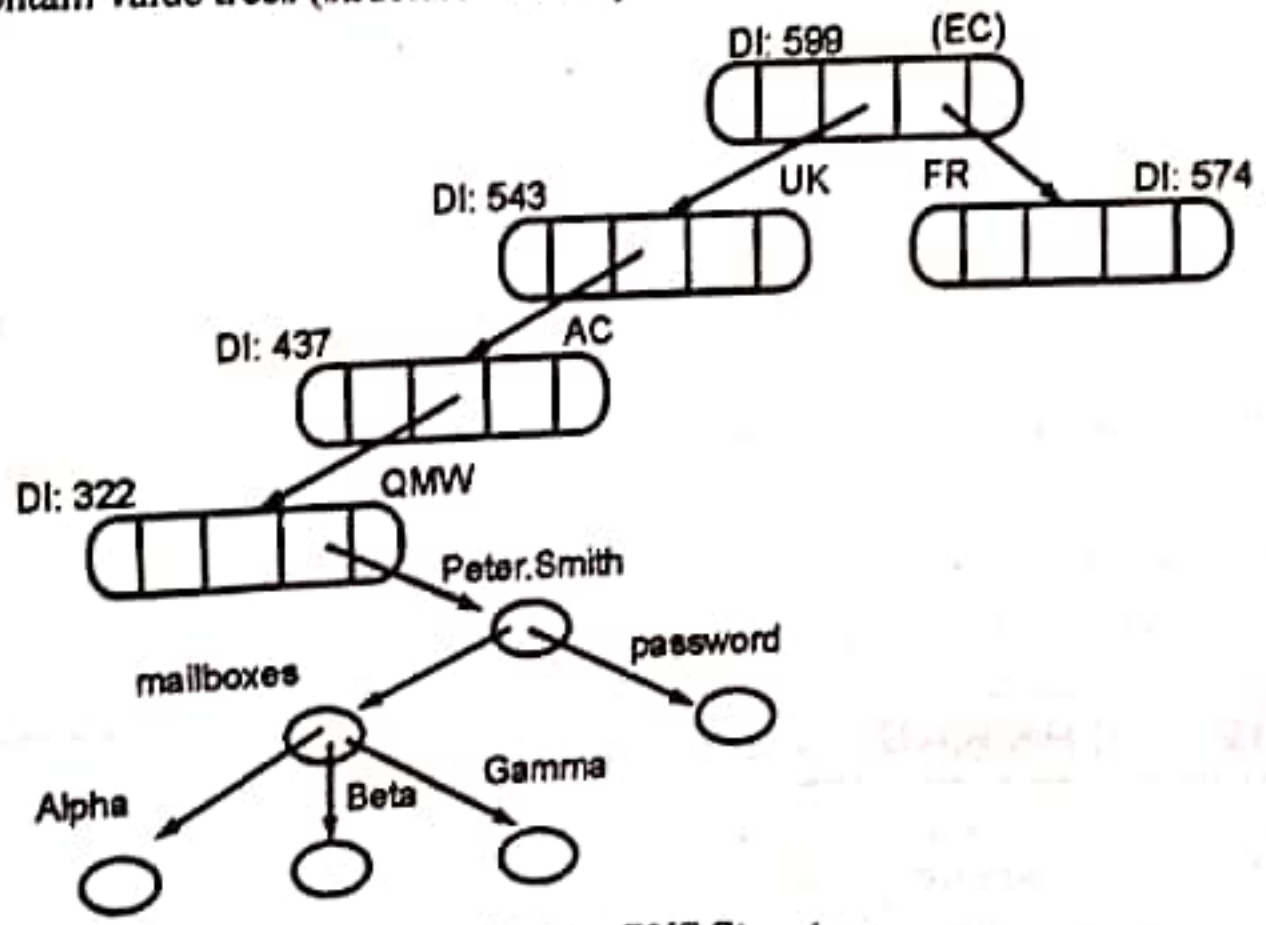


Fig. 6.10.1 : GNS Structure

6.11 X.500 DIRECTORY SERVICE

- The X.500 directory service is a global directory service.
- Its components cooperate to manage information about objects such as countries, organizations, people, machines, and so on in a worldwide scope.

- It provides the capability to look up information by name (a *white-pages service*) and to browse and search for information (a *yellow-pages service*).
- The information is held in a *directory information base* (DIB).
- Entries in the DIB are arranged in a tree structure called the *directory information tree* (DIT).
- Each entry is a named object and consists of a set of attributes. Each attribute has a defined attribute type and one or more values.
- The directory schema defines the mandatory and optional attributes for each class of object (called the *object class*).
- Each named object may have one or more object classes associated with it.
- The *X.500* namespace is hierarchical. An entry is unambiguously identified by a *distinguished name* (DN).
- A distinguished name is the concatenation of selected attributes from each entry, called the *relative distinguished name* (RDN), in the tree along a path leading from the root down to the named entry.
- Users of the *X.500* directory may (subject to access control) interrogate and modify the entries and attributes in the DIB.
- The *X.500* standard itself does not define any string representation for names. What is communicated between the *X.500* components is the structural form of names. The reasoning behind this is that the standard is sufficient to allow different implementations to interoperate.
- String names are never communicated between different implementations. Instead, they are necessary only for interaction with end-users.
- For that purpose, the standard allows any representation, not necessarily only string representations.
- Systems that are based on the *X.500*, such as the LDAP, the DCE Directory, Novell's NDS, and Microsoft's Active Directory, each define its own string representation.
- For example, in the LDAP, a DN's RDNs are arranged right to left, separated by the comma character (",").
- Here's an example of a name that starts with "c=us" at the top and leads to "cn=Rosanna Lee" at the leaf. **cn=Rosanna Lee, ou=People, o=Sun, c=us**
- Here's an example of the same name using the string representation of the DCE Directory and Microsoft's Active Directory **/c=us/o=Sun/ou=People/cn=Rosanna Lee**
- The convention for these systems is that RDNs are ordered left to right and separated by the forward slash character ("/").
- The X.500 standard defines a protocol (among others) for a client application to access the X.500 directory. Called the *Directory Access Protocol* (DAP), it is layered on top of the Open Systems Interconnection (OSI) protocol stack.

6.12 CASE STUDY ON HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

UQ Explain Hadoop Distributed File System (HDFS). **MU – May 18, 22**

UQ Write short note on Hadoop Distributed File System. **(MU - May 19)**

- HDFS (Hadoop Distributed File System) is the primary storage system used by Hadoop applications. This open-source framework works by rapidly transferring data between nodes.
- It's often used by companies who need to handle and store big data.
- HDFS is a key component of many Hadoop systems, as it provides a means for managing big data, as well as supporting big data analytics.
- HDFS is fault-tolerant and designed to be deployed on low-cost, commodity hardware.

- HDFS provides high throughput data access to application data and is suitable for applications that have large data sets and enables streaming access to file system data in Apache Hadoop.

6.12.1 HDFS Architecture

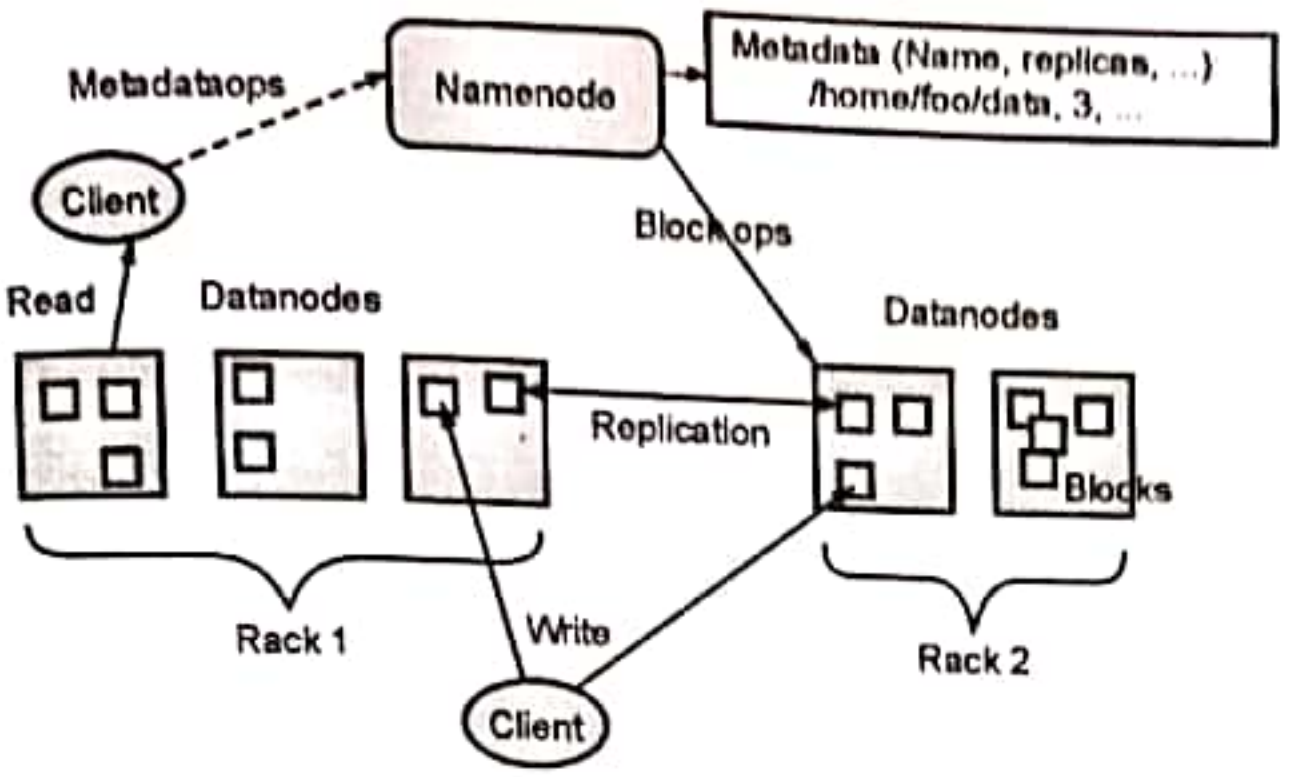


Fig. 6.12.1 : HDFS Architecture

- As we can see in Fig. 6.12.1, it focuses on NameNodes and DataNodes.
- The NameNode is the hardware that contains the GNU/Linux operating system and software.
- The Hadoop distributed file system acts as the master server and can manage the files, control a client's access to files, and overseas file operating processes such as renaming, opening, and closing files.
- A DataNode is hardware having the GNU/Linux operating system and DataNode software. For every node in a HDFS cluster, you will locate a DataNode. These nodes help to control the data storage of their system as they can perform operations on the file systems if the client requests, and also create, replicate, and block files when the NameNode instructs.
- The HDFS meaning and purpose is to achieve the following goals:
  - (i) **Manage large datasets :** Organizing and storing datasets can be a hard talk to handle. HDFS is used to manage the applications that have to deal with huge datasets. To do this, HDFS should have hundreds of nodes per cluster.
  - (ii) **Detecting faults :** HDFS should have technology in place to scan and detect faults quickly and effectively as it includes a large number of commodity hardware. Failure of components is a common issue.
  - (iii) **Hardware efficiency :** When large datasets are involved it can reduce the network traffic and increase the processing speed.

6.12.2 Working of HDFS

- HDFS enables the rapid transfer of data between compute nodes.
- At its outset, it was closely coupled with MapReduce, a framework for data processing that filters and divides up work among the nodes in a cluster, and it organizes and condenses the results into a cohesive answer to a query.
- Similarly, when HDFS takes in data, it breaks the information down into separate blocks and distributes them to different nodes in a cluster.
- With HDFS, data is written on the server once, and read and reused numerous times after that.
- HDFS has a primary NameNode, which keeps track of where file data is kept in the cluster.

- HDFS also has multiple DataNodes on a commodity hardware cluster – typically one per node in a cluster. The DataNodes are generally organized within the same rack in the data center. Data is broken down into separate blocks and distributed among the various DataNodes for storage. Blocks are also replicated across nodes, enabling highly efficient parallel processing.
- The NameNode knows which DataNode contains which blocks and where the DataNodes reside within the machine cluster. The NameNode also manages access to the files, including reads, writes, creates, deletes and the data block replication across the DataNodes.
- The NameNode operates in conjunction with the DataNodes. As a result, the cluster can dynamically adapt to server capacity demand in real time by adding or subtracting nodes as necessary.
- The DataNodes are in constant communication with the NameNode to determine if the DataNodes need to complete specific tasks. Consequently, the NameNode is always aware of the status of each DataNode.
- If the NameNode realizes that one DataNode isn't working properly, it can immediately reassign that DataNode's task to a different node containing the same data block.
- DataNodes also communicate with each other, which enables them to cooperate during normal file operations.
- Moreover, the HDFS is designed to be highly fault tolerant.
- The file system replicates or copies each piece of data multiple times and distributes the copies to individual nodes, placing at least one copy on a different server rack than the other copies.

6.12.3 Advantages of HDFS

As an open-source subproject within Hadoop, HDFS offers the following benefits when dealing with big data:

(1) **Fault tolerance :** HDFS has been designed to detect faults and automatically recover quickly ensuring continuity and reliability.

(2) **Speed :** Because of its cluster architecture, it can maintain 2 GB of data per second.

(3) **Access to more types of data :** Specifically, it accesses streaming data. Because of its design to handle large amounts of data for batch processing it allows for high data throughput rates making it ideal to support streaming data.

(4) **Compatibility and portability :** HDFS is designed to be portable across a variety of hardware setups and compatible with several underlying operating systems ultimately providing users optionality to use HDFS with their own tailored setup. These advantages are especially significant when dealing with big data and were made possible by the way HDFS handles data.

(5) **Scalable :** You can scale resources according to the size of your file system. HDFS includes vertical and horizontal scalability mechanisms.

(6) **Data locality :** When it comes to the Hadoop file system, the data resides in data nodes, as opposed to having the data move to where the computational unit is. Shortening the distance between the data and the computing process, decreases network congestion and makes the system more effective and efficient.

(7) **Cost-effective :** Initially when we think of data we may think of expensive hardware and hogged bandwidth. When hardware failure strikes, it can be very costly to fix. With HDFS, the data is stored inexpensively as it's virtual, which can drastically reduce file system metadata and file system namespace data storage costs. What's more, because HDFS is open source, businesses don't need to worry about paying a licensing fee.

(8) **Stores large amounts of data :** Data storage is what HDFS is all about – meaning data of all varieties and sizes – but particularly large amounts of data from corporations that are struggling to store it. This includes both structured and unstructured data.

(9) **Flexible :** Unlike some other more traditional storage databases, there's no need to process the data collected before

storing it. You're able to store as much data as you want, with the opportunity to decide exactly what you'd like to do with it and how to use it later. This also includes unstructured data like text, videos, and images.

6.13 DESIGNING DISTRIBUTED SYSTEMS : GOOGLE CASE STUDY

- Google is a US-based corporation, which was born out of a research project at Stanford with the company launched in 1998.
- It offers Internet search and broader web applications and earns revenue largely from advertising associated with such services.
- Google's search engine is now a major player in cloud computing. It provides good performance in terms of scalability, reliability, performance, and openness.
- Google Search Engine consist of a set of services like :
  (1) **Crawling** : To locate and retrieve the contents of the web and pass the content onto the indexing subsystem. Performed by a software called Googlebot.
  (2) **Indexing** : Produce an index for the contents of the web that is similar to an index at the back of a book, but on a much larger scale.
  (3) **Ranking** : Relevance of the retrieved links. Ranking algorithm is called PageRank, a page will be viewed as important if it is linked to by a large number of other pages.

6.13.1 Google File System (GFS)

- Google Inc. developed the Google File System (GFS), a scalable distributed file system (DFS), to meet the company's growing data processing needs.
- GFS offers fault tolerance, dependability, scalability, availability, and performance to big networks and connected nodes.
- GFS is made up of a number of storage systems constructed from inexpensive commodity hardware parts.
- The search engine, which creates enormous volumes of data that must be kept, is only one example of how it is customized to meet Google's various data use and storage requirements.
- The Google File System reduced hardware flaws while gains of commercially available servers.
- The GFS node cluster consists of a single master and several chunk servers that various client systems regularly access.
- On local disks, chunk servers keep data in the form of Linux files.
- Large (64 MB) pieces of the stored data are split up and replicated at least three times around the network.
- Reduced network overhead results from the greater chunk size.
- Without hindering applications, GFS is made to meet Google's huge cluster requirements.
- Hierarchical directories with path names are used to store files.
- The master is in-charge of managing metadata, including namespace, access control, and mapping data.
- The master communicates with each chunk server by timed heartbeat messages and keeps track of its status updates.
- More than 1,000 nodes with 300 TB of disc storage capacity make up the largest GFS clusters. This is available for constant access by hundreds of clients.

6.13.2 GFS Architecture

A group of computers makes up GFS. A cluster is just a group of connected computers. There could be hundreds or even thousands of computers in each cluster. There are three basic entities included in any GFS cluster as follows :

(1) **GFS Clients :** They can be computer programs or applications which may be used to request files. Requests may be made to access and modify already-existing files or add new files to the system.

(2) **GFS Master Server :** It serves as the cluster's coordinator. It preserves a record of the cluster's actions in an operation log. Additionally, it keeps track of the data that describes chunks, or metadata. The chunks' place in the overall file and which files they belong to are indicated by the metadata to the master server.

(3) **GFS Chunk Servers :** They are the GFS's workhorses. They keep 64 MB-sized file chunks. The master server does not receive any chunks from the chunk servers. Instead, they directly deliver the client the desired chunks. The GFS makes numerous copies of each chunk and stores them on various chunk servers in order to assure stability; the default is three copies. Every replica is referred to as one.

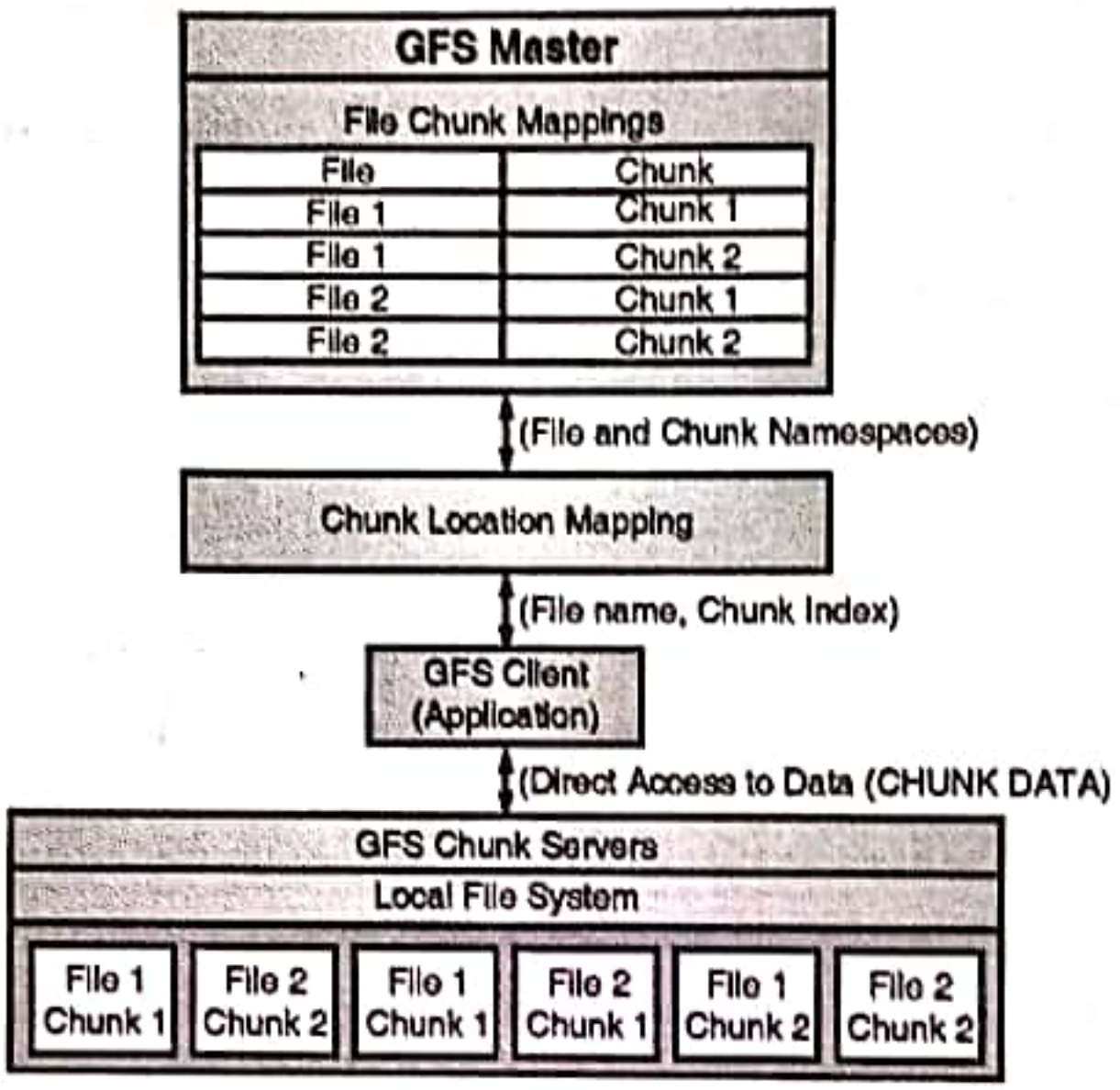


Fig. 6.13.1 : GFS Architecture

6.13.3 Features, Advantages and Disadvantages of GFS

Features of GFS

(1) Namespace management and locking.

(2) Fault tolerance.

(3) Reduced client and master interaction because of large chunk server size.

(4) High availability.

(5) Critical data replication.

(6) Automatic and efficient data recovery.

(7) High aggregate throughput.

Advantages of GFS

(1) High accessibility Data is still accessible even if a few nodes fail. (replication) Component failures are more common than not, as the saying goes.

(2) Excessive throughput. many nodes operating concurrently.

(3) Dependable storing. Data that has been corrupted can be found and duplicated.

Disadvantages of GFS

(1) Not the best fit for small files.

(2) Master may act as a bottleneck.

(3) Unable to type at random.

(4) Suitable for procedures or data that are written once and only read (appended) later.

Descriptive Questions

Q.1 What are the desirable features of good distributed file systems? Explain file sharing semantic of it.

Q.2 Explain file accessing models.

Q.3 Explain File Caching Schemes.

Q.4 Write a note on: Andrew File System (AFS)

Q.5 Write a note on: Network File System (NFS)

Q.6 Write a note on: Google File System (GFS)

Q.7 Explain X.500 Directory Service.

Q.8 Explain Hadoop Distributed File System (HDFS).

Chapter Ends...

❑❑❑

LAB MANUAL

Experiment No. 1 : Inter-process Communication

☞ **AIM : To implement Inter-process Communication using TCP Based on Socket Programming.**

This program is to implement inter-process communication between client and server. Client will send integers to the server one by one. To stop sending integers, "stop" message is written. Server computes sum of integers send by client and returns back the result.

```
//TCPServer.java
import java.util.*;
import java.io.*;
import java.net.*;
public class TCPServer
{
    public static void main(String args[]) throws Exception
    {
        ServerSocket server = new ServerSocket(25);
        System.out.println("Connecting...");
        Socket ss = server.accept();
        System.out.println("Connected");
        DataInputStream din = new DataInputStream(ss.getInputStream());
        DataOutputStreamdout= new DataOutputStream(ss.getOutputStream());
        String str = "";
        int sum = 0;
        System.out.println("Receiving integers from client...");
        while (true)
        {
            str = din.readUTF();
            if (str.equals("stop"))
            break;
            sum = sum + Integer.parseInt(str);
        }
        dout.writeUTF(Integer.toString(sum));
        dout.flush();
        din.close();
        ss.close();
        server.close();
    }
}
```

```
//TCPClient.java
import java.io.*;
import java.util.*;
import java.net.*;
public class TCPClient
{
    public static void main(String args[]) throws Exception
    {
        System.out.println("Connecting...");
        Socket client = new Socket("127.0.0.1", 25);
        System.out.println("Connected");
        DataInputStream din = new DataInputStream(client.getInputStream());
        DataOutputStreamdout = new DataOutputStream(client.getOutputStream());
        Scanner sc = new Scanner(System.in);
        String send = "";
        while (!send.equals("stop"))
        {
            System.out.print("Send: ");
            send = sc.nextLine();
            dout.writeUTF(send);
        }
        dout.flush();
        String recv = din.readUTF();
        System.out.println("Sum of the integers is: " + recv);
        dout.close();
        din.close();
        client.close();
    }
}
```

Output

(1) Compile TCPServer.java and TCPClient.java in 2 different command prompts.

(2) Execute TCPServer.java

(3) Execute TCPClient.java

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac TCPServer.java

C:\Program Files\Java\jdk1.8.0_341\bin>java TCPServer
Connecting...
Connected
Receiving integers from client...
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac TCPClient.java

C:\Program Files\Java\jdk1.8.0_341\bin>java TCPClient
Connecting...
Connected
Send: 1
Send: 2
Send: 3
Send: 4
Send: stop
Sum of the integers is: 10
```

Experiment No. 2 : Client/Server using RPC/RMI

AIM : To implement Client-Server Application using Java RMI.

In this program, we find the sum of two integers using RMI application.

```
//AddInterface.java
import java.rmi.*;
public interface AddInterface extends Remote
{
    public int sum(int n1,int n2) throws RemoteException;
}
//Add.java
import java.rmi.*;
import java.rmi.server.*;
public class Add extends UnicastRemoteObject implements AddInterface
{
    int num1,num2;
    public Add() throws RemoteException {}
    public int sum(int n1,int n2) throws RemoteException
    {
        num1=n1;
        num2=n2;
        return num1+num2; }
    }
    //AddServer.java
    import java.rmi.Naming;
    public class AddServer
    {
        public static void main(String args[])
        {
        try
            {
                Naming.rebind("Add",new Add());
```

```
            System.out.println("Server is connected and waiting for the client");
        } catch(Exception e)
        {
            System.out.println("Server could not connect: "+e);
        }
    }
}
//AddClient.java
import java.rmi.Naming;
public class AddClient
{
    public static void main(String args[])
    {
        try
        {
            AddInterface ai=(AddInterface)Naming.lookup("//localhost/Add");
            System.out.println("The sum of 2 numbers is: "+ai.sum(10,2));
        } catch(Exception e)
        {
            System.out.println("Client Exception: "+e);
        }
    }
}
```

Output

(1) Compile AddInterface.java, Add.java, AddServer.java and AddClient.java

(2) Start rmiregistry using command ***rmiregistry&*** in command prompt.

(3) Execute AddServer.java

(4) Execute AddClient.java

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac AddInterface.java Add.java AddServer.java AddClient.java

C:\Program Files\Java\jdk1.8.0_341\bin>rmiregistry &
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java AddServer
Server is connected and waiting for the client
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java AddClient
The sum of 2 numbers is: 12
```

Experiment No. 3 : Group Communication

AIM : To implement a program to demonstrate group communication.

In this program, we will have one master and multiple slaves. Master will broadcast messages to all the slaves. So, we are implementing a one-to-many type of group communication. Both Master and Slaves are connected to Server for synchronization.

```
//GCServer.java
import java.util.*;
import java.io.*;
import java.net.*;
public class GCServer
{
    static ArrayList<ClientHandler> clients = new ArrayList<ClientHandler>();
    public static void main(String args[]) throws Exception
    {
        ServerSocket server = new ServerSocket(25);
        Message msg = new Message();
        int count = 0;
        while (true)
        {
            Socket ss = server.accept();
            DataInputStream din = new DataInputStream(ss.getInputStream());
            DataOutputStreamdout = new DataOutputStream(ss.getOutputStream());
            ClientHandlerchlr = new ClientHandler(ss, din, dout, msg);
            Thread t = chlr;
            clients.add(chlr);
            count++;
            t.start();
        }
    }
}
class Message
{
    String msg;
    public void setMsg(String msg)
    {
        this.msg = msg;
    }
    public void getMsg()
    {
        System.out.println("\nNEW GROUP MESSAGE: " + this.msg);
        for(int i = 0; i<GCServer.clients.size(); i++)
```

```
    {
        Try
        {
            System.out.print("Client: " + GCServer.clients.get(i).ip + "; ");
            GCServer.clients.get(i).out.writeUTF(this.msg);
            GCServer.clients.get(i).out.flush();
        }
        catch(Exception e)
        {
            System.out.print(e);
        }
    }
}
class ClientHandler extends Thread
{
    DataInputStreamin;
    DataOutputStreamout;
    Socket socket;
    int sum;
    float res;
    booleanconn;
    Message msg;
    String ip;
    public ClientHandler(Socket s, DataInputStream din, DataOutputStreamdout, Message msg) {
    this.socket = s;
    this.in = din;
    this.out = dout;
    this.conn = true;
    this.msg = msg;
    this.ip = (((InetSocketAddress)
    this.socket.getRemoteSocketAddress()).getAddress()).toString().replace("/","");
}
public void run()
{
    while(conn == true)
    {
        try
        {
            String input = this.in.readUTF();
            this.msg.setMsg(input);
            this.msg.getMsg();
        } catch (Exception E)
```

```
        {
            conn = false;
            System.out.println(E);
        }
    }
    closeConn();
}
public void closeConn()
{
    try
    {
        this.out.close();
        this.in.close();
        this.socket.close();
    } catch(Exception E)
    {
        System.out.println(E);
    }
}
}
//GCMaster.java
import java.io.*;
import java.util.*;
import java.net.*;
public class GCMaster
{
    public static void main(String args[]) throws Exception
    {
        Socket client = new Socket("127.0.0.1", 25);
        DataInputStream din = new DataInputStream(client.getInputStream());
        DataOutputStreamdout = new DataOutputStream(client.getOutputStream());
        System.out.println("Connected as Master");
        Scanner sc = new Scanner(System.in);
        String send = "";
        do
        {
        System.out.print("Message('close' to stop): ");
        send = sc.nextLine();
        dout.writeUTF(send);
        dout.flush();
    } while (!send.equals("stop"));
    dout.close();
    din.close();
```

```
        client.close();
    }
}
```

GCSlave.java

```
import java.io.*;
import java.util.*;
import java.net.*;
public class GCSlave
{
    public static void main(String args[]) throws Exception
    {
        Socket client = new Socket("127.0.0.1", 25);
        DataInputStream din = new DataInputStream(client.getInputStream());
        System.out.println("Connected as Slave");
        String recv = "";
        do
        {
            recv = din.readUTF();
            System.out.println("Master says: " + recv);
        } while (!recv.equals("stop"));
        din.close();
        client.close();
    }
}
```

Output

(1) Compile GCServer.java, GCMaster.java, GCSlave.java

(2) Execute GCMaster.java

(3) Execute GCSlave.java in multiple command prompts.

(4) Type message in GCMaster.java window. That message will be broadcasted to all Slaves connected.

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac GCServer.java

C:\Program Files\Java\jdk1.8.0_341\bin>java GCServer

NEW GROUP MESSAGE: Hello Everyone
Client: 127.0.0.1; Client: 127.0.0.1; Client: 127.0.0.1;
NEW GROUP MESSAGE: How are you?
Client: 127.0.0.1; Client: 127.0.0.1; Client: 127.0.0.1;
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac GCMaster.java

C:\Program Files\Java\jdk1.8.0_341\bin>java GCMaster
Connected as Master
Message('close' to stop): Hello Everyone
Message('close' to stop): How are you?
Message('close' to stop): _
```

Slave 1

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac GCSlave.java

C:\Program Files\Java\jdk1.8.0_341\bin>java GCSlave
Connected as Slave
Master says: Hello Everyone
Master says: How are you?
```

Slave 2

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac GCSlave.java

C:\Program Files\Java\jdk1.8.0_341\bin>java GCSlave
Connected as Slave
Master says: Hello Everyone
Master says: How are you?
```

Experiment No. 4 : Clock Synchronization Algorithms

AIM : To implement Lamport's Clock Synchronization Algorithm.

```
//lamport.java
import java.util.*;
import java.util.Scanner;
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

public class lamport
{
    int e[][]=new int[10][10];
    int en[][]=new int[10][10];
    int ev[]=new int[10];
    int i,p,j,k;
    HashMap<Integer,Integer> hm=new HashMap<Integer,Integer>();
    int xpoints[] =new int[5];
    int ypoints[] =new int[5];
    class draw extends Jframe
    {
```

```
private final int ARR_SIZE = 4;

    void drawArrow(Graphics g1, int x1, int y1, int x2, int y2)
{
    Graphics2D g = (Graphics2D) g1.create();

    double dx = x2 - x1, dy = y2 - y1;
    double angle = Math.atan2(dy, dx);
    int len = (int) Math.sqrt(dx*dx + dy*dy);
    AffineTransform at = AffineTransform.getTranslateInstance(x1, y1);
    at.concatenate(AffineTransform.getRotateInstance(angle));
    g.transform(at);
    // Draw horizontal arrow starting in (0, 0)
    g.drawLine(0, 0, len, 0);
    g.fillPolygon(new int[] {len, len-ARR_SIZE, len-ARR_SIZE, len},
    new int[] {0, -ARR_SIZE, ARR_SIZE, 0}, 4);
}
public void paintComponent(Graphics g)
{
    for (int x = 15; x < 200; x += 16)
    drawArrow(g, x, x, x, 150);
    drawArrow(g, 30, 300, 300, 190);
}
public void paint(Graphics g)
{
    int h1,h11,h12;
    Graphics2D go=(Graphics2D)g;
    go.setPaint(Color.black);
    for(i=1;i<=p;i++)
    {
        go.drawLine(50,100*i,450,100*i);
    }
    for(i=1;i<=p;i++)
    {
        for(j=1;j<=ev[i];j++)
        {
    k=i*10+j;
    go.setPaint(Color.blue);
    go.fillOval(50*j,100*i-3,5,5);
    go.drawString("e"+i+j+"("+en[i][j]+")",50*j,100*i-5);
    h1=hm.get(k);
    if(h1!=0)
    {
        h11=h1/10;
        h12=h1%10;
        go.setPaint(Color.red);
```

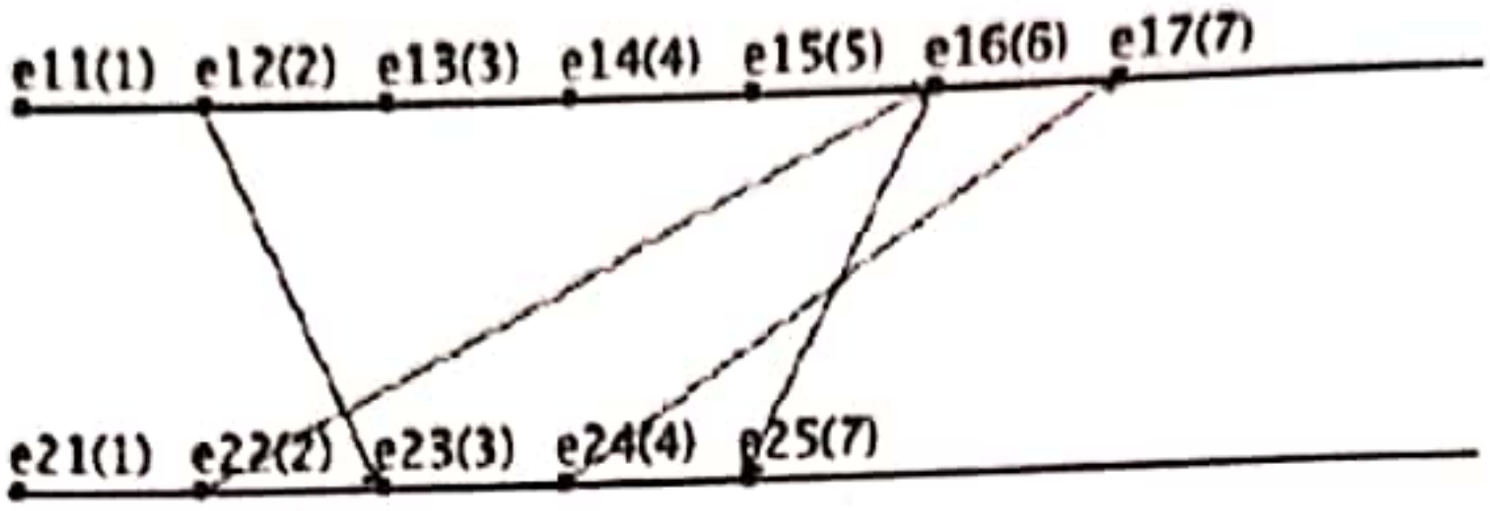
```
                        drawArrow(go,50*h12+2,100*h11,50*j+2,100*i);
                    }
                }
            }
        }
    }
    public void calc()
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the number of process:");
        p=sc.nextInt();
        System.out.println("Enter the no of events per process:");
        for(i=1;i<=p;i++)
        {
            ev[i]=sc.nextInt();
        }
        System.out.println("Enter the relationship:");
        for(i=1;i<=p;i++)
        {
            System.out.println("For process:"+i);
            for(j=1;j<=ev[i];j++)
            {
                System.out.println("For event:"+(j));
                int input=sc.nextInt();
                k=i*10+j;
                hm.put(k,input);
                if(j==1)
                en[i][j]=1;
            }
        }
        for(i=1;i<=p;i++)
        {
            for(j=2;j<=ev[i];j++)
            {
                k=i*10+j;
                if(hm.get(k)==0)
                {
                    en[i][j]=en[i][j-1]+1;
                }
                else
                {
                    int a=hm.get(k);
                    int p1=a/10;
                    int e1=a%10;
                    if(en[p1][e1]>en[i][j-1])
                    en[i][j]=en[p1][e1]+1;
```

```
                else
                en[i][j]=en[i][j-1]+1;
            }
        }
    }
    for(i=1;i<=p;i++)
    {
        for(j=1;j<=ev[i];j++)
        {
            System.out.println(en[i][j]);
        }
    }
    JFramejf=new draw();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(500,500);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        lamport lam=new lamport();
        lam.calc();
    }
}
```

Output

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac lamport.java

C:\Program Files\Java\jdk1.8.0_341\bin>java lamport
Enter the number of process:
2
Enter the no of events per process:
7
5
Enter the relationship:
For process:1
For event:1
0
For event:2
0
For event:3
0
For event:4
0
For event:5
0
For event:6
22
For event:7
24
For process:2
For event:1
0
For event:2
0
For event:3
12
For event:4
0
For event:5
16
```



Experiment No. 5 : Election Algorithm

AIM : To implement Bully Election Algorithm

```
//Bully.java
import java.io.*;
import java.util.*;
public class Bully
{
    static int n;
    static int pro[] = new int[100];
    static int sta[] = new int[100];
    static int co;
    public static void main(String args[])
    {
        System.out.print("Enter the number of process: ");
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        int i, j, c, cl = 1;
        for (i = 0; i< n; i++)
        {
            sta[i] = 1;
            pro[i] = i;
        }
        boolean choice = true;
        int ch;
        do
        {
            System.out.println("Enter Your Choice");
            System.out.println("1. Crash Process");
            System.out.println("2. Recover Process");
            System.out.println("3. Exit");
            System.out.print("> ");
```

```
        ch = sc.nextInt();
        switch (ch)
        {
            case 1:
            System.out.print("Enter the process number: ");
            c = sc.nextInt();
            sta[c - 1] = 0;
            cl = 1;
            break;
            case 2:
            System.out.print("Enter the process number: ");
            c = sc.nextInt();
            sta[c - 1] = 1;
            cl = 1;
            break;
            case 3:
            choice = false;
            cl = 0;
            break;
        }
        if (cl == 1)
        {
            System.out.print("Which process will initiate election? = ");
            int ele = sc.nextInt();
            elect(ele);
        }
        System.out.println("Final coordinator is " + co);
    } while (choice);
}
static void elect(int ele)
{
    ele = ele - 1;
    co = ele + 1;
    for (int i = 0; i< n; i++)
    {
        if (pro[ele] < pro[i])
        {
            System.out.println("Election message is sent from " + (ele + 1) + " to " + (i + 1));
            if (sta[i] == 1)
            System.out.println("Ok message is sent from " + (i + 1) + " to " + (ele + 1));
            if (sta[i] == 1)
            elect(i + 1);
        }
    }
  }
}
```

Output

```
Enter the number of process: 5
Enter Your Choice
1. Crash Process
2. Recover Process
3. Exit
> 1
Enter the process number: 1
Which process will initiate election? = 2
Election message is sent from 2 to 3
Ok message is sent from 3 to 2
Election message is sent from 3 to 4
Ok message is sent from 4 to 3
Election message is sent from 4 to 5
Ok message is sent from 5 to 4
Election message is sent from 3 to 5
Ok message is sent from 5 to 3
Election message is sent from 2 to 4
Ok message is sent from 4 to 2
Election message is sent from 4 to 5
Ok message is sent from 5 to 4
Election message is sent from 2 to 5
Ok message is sent from 5 to 2
Final coordinator is 5
Enter Your Choice
1. Crash Process
2. Recover Process
3. Exit
> 1
Enter the process number: 5
Which process will initiate election? = 3
Election message is sent from 3 to 4
Ok message is sent from 4 to 3
Election message is sent from 4 to 5
Election message is sent from 3 to 5
Final coordinator is 4
Enter Your Choice
1. Crash Process
2. Recover Process
3. Exit
> 2
Enter the process number: 1
Which process will initiate election? = 3
Election message is sent from 3 to 4
Ok message is sent from 4 to 3
Election message is sent from 4 to 5
Election message is sent from 3 to 5
Final coordinator is 4
Enter Your Choice
1. Crash Process
2. Recover Process
3. Exit
>
```

Experiment No. 6 : Mutual Exclusion Algorithm

Aim : To implement program for Mutual Exclusion Algorithm

```
//MutualServer.java
import java.io.*;
import java.net.*;
public class MutualServer implements Runnable
{
    Socket socket = null;
    static ServerSocketss;
    MutualServer(Socket newSocket)
    {
        this.socket = newSocket;
    }
    public static void main(String args []) throws IOException
    {
        ss = new ServerSocket(7000);
        System.out.println("Server Started");
        while(true)
        {
            Socket s = ss.accept();
            MutualServer es = new MutualServer(s);
            Thread t = new Thread(es);
            t.start();
        }
    }
    public void run()
    {
        try
        {
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            while(true)
            {
                System.out.println(in.readLine());
            }
        }
        catch(Exception e){}
    }
}
//ClientOne.java
import java.io.*;
import java.net.*;
```

```
public class ClientOne
{
    public static void main(String args []) throws IOException
    {
        Socket s = new Socket("127.0.0.1",7000);
        PrintStream out = new PrintStream(s.getOutputStream());
        ServerSocket ss = new ServerSocket(7001);
        Socket s1 = ss.accept();
        BufferedReader in1 = new BufferedReader(new InputStreamReader(s1.getInputStream()));
        PrintStream out1 = new PrintStream(s1.getOutputStream());
        BufferedReaderbr = new BufferedReader(new InputStreamReader(System.in));
        String str = "Token";
        while(true)
        {
            if(str.equalsIgnoreCase("Token"))
            {
                System.out.println("Do you want to send some data");
                System.out.println("Enter Yes or No");
                str = br.readLine();
                if(str.equalsIgnoreCase("Yes"))
                {
                    System.out.println("Enter the data");
                    str=br.readLine();
                    out.println(str);
                }
                out1.println("Token");
            }
            System.out.println("Waiting for Token");
            str = in1.readLine();
        }
    }
}
//ClientTwo.java
import java.io.*;
import java.net.*;
public class ClientTwo
{
    public static void main(String args []) throws IOException
    {
        Socket s = new Socket("127.0.0.1",7000);
        PrintStream out = new PrintStream(s.getOutputStream());
        Socket s2 = new Socket("127.0.0.1",7001);
        BufferedReader in2 = new BufferedReader(new InputStreamReader(s2.getInputStream()));
```

```
PrintStream out2 = new PrintStream(s2.getOutputStream());
BufferedReaderbr = new BufferedReader(new InputStreamReader(System.in));
String str = "Token";
while(true)
{
    System.out.println("Waiting for Token");
    str = in2.readLine();
    if(str.equalsIgnoreCase("Token"))
    {
        System.out.println("Do you want to send some data");
        System.out.println("Enter Yes or No");
        str = br.readLine();
        if(str.equalsIgnoreCase("Yes"))
        {
            System.out.println("Enter the data");
            str=br.readLine();
            out.println(str);
        }
        out2.println("Token");
    }
  }
 }
}
```

Output

(1) Compile MutualServer.java, ClientOne.java, ClientTwo.java

(2) Execute MutualServer.java

(3) Open new command prompt and execute ClientOne.java. Keep it running till ClientTwo starts.

(4) Open another command prompt and execute ClientTwo.java. The output allows both the clients to use token and share their messages with each other using Token Ring concept. To send the message, the client has to accept the token by typing Yes followed by the message alternately and has to type No to release the token.

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac MutualServer.java

C:\Program Files\Java\jdk1.8.0_341\bin>javac ClientOne.java

C:\Program Files\Java\jdk1.8.0_341\bin>javac ClientTwo.java

C:\Program Files\Java\jdk1.8.0_341\bin>java MutualServer
Server Started
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java ClientOne
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java ClientTwo
Waiting for Token
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java ClientOne
Do you want to send some data
Enter Yes or No
Yes
Enter the data
Distributed Computing
Waiting for Token
Do you want to send some data
Enter Yes or No
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java ClientTwo
Waiting for Token
Do you want to send some data
Enter Yes or No
Yes
Enter the data
Parallel Computing
Waiting for Token
```

```
C:\Program Files\Java\jdk1.8.0_341\bin>java MutualServer
Server Started
Distributed Computing
Parallel Computing
```

Experiment No. 7 : Deadlock Management in Distributed Systems

Aim : To implement Banker's Algorithm for Deadlock Management

```
//Bankers.java
import java.util.Scanner;
public class Bankers
{
    private int need[][],allocate[][],max[][],avail[][],np,nr;
    private void input()
    {
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter no. of processes and resources : ");
        np=sc.nextInt();            //no. of process
        nr=sc.nextInt();            //no. of resources
        need=new int[np][nr];       //initializing arrays
        max=new int[np][nr];
        allocate=new int[np][nr];
        avail=new int[1][nr];
        System.out.println("Enter allocation matrix -->");
        for(int i=0;i<np;i++)
        for(int j=0;j<nr;j++)
        allocate[i][j]=sc.nextInt(); //allocation matrix
        System.out.println("Enter max matrix -->");
```

```
    for(int i=0;i<np;i++)
    for(int j=0;j<nr;j++)
    max[i][j]=sc.nextInt();          //max matrix
    System.out.println("Enter available matrix -->");
    for(int j=0;j<nr;j++)
    avail[0][j]=sc.nextInt();        //available matrix
    sc.close();
}
private int[][] calc_need()
{
    for(int i=0;i<np;i++)
    for(int j=0;j<nr;j++)  //calculating need matrix
    need[i][j]=max[i][j]-allocate[i][j];
    return need;
}
private booleancheck(int i)
{
    //checking if all resources for ith process can be allocated
    for(int j=0;j<nr;j++)
    if(avail[0][j]<need[i][j])
    return false;
return true;
}
public void isSafe()
{
    input();
    calc_need();
    booleandone[]=new boolean[np];
    int j=0;
    while(j<np)
    {   //until all process allocated
        boolean allocated=false;
        for(int i=0;i<np;i++)
        if(!done[i] && check(i)){  //trying to allocate
        for(int k=0;k<nr;k++)
        avail[0][k]=avail[0][k]-need[i][k]+max[i][k];
        System.out.println("Allocated process : "+i);
        allocated=done[i]=true;
        j++;
    }
    if(!allocated) break;          //if no allocation
}
if(j==np)                          //if all processes are allocated
```

```
    System.out.println("\nSafely allocated");
    else
    System.out.println("All processescant be allocated safely");
}
    public static void main(String[] args)
    {
        new Bankers().isSafe();
    }
}
```

Output

```
C:\Program Files\Java\jdk1.8.0_341\bin>javac Bankers.java

C:\Program Files\Java\jdk1.8.0_341\bin>java Bankers
Enter no. of processes and resources : 3 4
Enter allocation matrix -->
1 2 2 1
1 0 3 3
1 2 1 0
Enter max matrix -->
3 3 2 2
1 1 3 4
1 3 5 0
Enter available matrix -->
3 1 1 2
Allocated process : 0
Allocated process : 1
Allocated process : 2

Safely allocated
```

Experiment No. 8 : Load Balancing

Aim : To implement the program for demonstrating a load-balancing approach in a distributed environment.

```
//LoadBalance.java
import java.util.*;
public class LoadBalance
{
    static void printLoad(int servers, int processes)
    {
        int each = processes / servers;
        int extra = processes % servers;
        int total = 0;
        int i = 0;
        for (i = 0; i< extra; i++)
        {
            System.out.println("Server " + (i + 1) + " has " + (each + 1) + " Processes");
```

```
    }
    for (; i< servers; i++)
    {
        System.out.println("Server " + (i + 1) + " has " + each + " Processes");
    }
}
public static void main(String[] args)
{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter the number of Servers: ");
    int servers = sc.nextInt();
    System.out.print("Enter the number of Processes: ");
    int processes = sc.nextInt();
    while (true)
    {
        printLoad(servers, processes);
        System.out.println("\n 1.Add Servers 2.Remove Servers 3.Add Processes 4.Remove Processes
        5.Exit ");
        System.out.print("> ");
        switch (sc.nextInt())
        {
            case 1:
            System.out.println("How many more servers to add ? ");
            servers += sc.nextInt();
            break;
            case 2:
            System.out.println("How many more servers to remove ? ");
            servers -= sc.nextInt();
            break;
            case 3:
            System.out.println("How many more Processes to add ? ");
            processes += sc.nextInt();
            break;
            case 4:
            System.out.println("How many more Processes to remove ? ");
            processes -= sc.nextInt();
            break;
            case 5:
            return;
        }
    }
  }
}
```

Output

```
~/proj/DC/exp5/ java LoadBalance
Enter the number of Servers: 4
Enter the number of Processes: 17
Server 1 has 5 Processes
Server 2 has 4 Processes
Server 3 has 4 Processes
Server 4 has 4 Processes

 1.Add Servers 2.Remove Servers 3.Add Processes 4.Remove Processes 5.Exit
> 3
How many more Processes to add ?
3
Server 1 has 5 Processes
Server 2 has 5 Processes
Server 3 has 5 Processes
Server 4 has 5 Processes

 1.Add Servers 2.Remove Servers 3.Add Processes 4.Remove Processes 5.Exit
> 4
How many more Processes to remove ?
7
Server 1 has 4 Processes
Server 2 has 3 Processes
Server 3 has 3 Processes
Server 4 has 3 Processes

 1.Add Servers 2.Remove Servers 3.Add Processes 4.Remove Processes 5.Exit
> 1
How many more servers to add ?
1
Server 1 has 3 Processes
Server 2 has 3 Processes
Server 3 has 3 Processes
Server 4 has 2 Processes
Server 5 has 2 Processes
```

...Lab End

❑❑❑