WORLD COLLEGE OF TECHNOLOGY & MANAGEMENT(WCTM)

FACULTY NAME	**MS. SONAM TIWARI**
DEPARTMENT	**COMPUTER SCIENCE AND ENGINEERING**
SUBJECT	**DISTRIBUTED SYSTEM**
CLASS	**B.TECH-AIDS**
SEMESTER	**6th SEMESTER**
SESSION	**EVEN (JAN 2024 - MAY 2024)**

Contents

LESSON PLAN			
Name of the Faculty:		**MS. SONAM TIWARI**	
Discipline		**B.TECH-AIDS**	
Semester		**6TH SEM**	
Subject		**DISTRIBUTED SYSTEM**	
Lesson Plan Duration:		**15weeks (FROM JAN 2024 to MAY 2024)**	
Work Load (Lecture/Practical) per week (in hours):Lectures-03 Practical: 00			
WEEK	**DATE**	**THEORY**	
		LECTURE	**TOPIC(Including Assignments and Tests)**
1st		**1st**	Introduction to DS, Types of DS, Characteristics, advantages and disadvantages of distributed system
		2nd	Architecture of distributed system
		3rd	Goals of distributed system, Hardware & Software concepts
2nd		**4th**	Design Issues, Communication in distributed system
		5th	Layered Protocols, ATM networks
		6th	Client/Server model, Peer to Peer model, Remote Procedure calls
3rd		**7th**	Group Communication, Middleware and distributed operating system.
		8th	**Revision/Test**
		9th	**Assignment 1**
4th		**10th**	Clock Synchronization(logical, physical clocks, algorithms, use of synchronized clocks)
		11th	Mutual Exclusion(centralized algorithm, distributed algorithm, a token ring algorithm)
		12th	Election Algorithm, the bully algorithm
5th		**13th**	Ring Algorithm, Atomic Transactions(Introduction, Transaction model, implementation and concurrency control)
		14th	Deadlock in distributed systems
		15th	Distributed deadlock prevention
		16th	Distributed deadlock detection

6th		**17th**	**Revision/ Test**
		18th	**Assignment 2**
7th		**19th**	Processes and Processors in distributed system
		20th	Threads(Introduction to threads, thread usage, design issues for thread packages, implementing a threads package, threads and RPC)
		21st	System models(the workstation model,) System models(the workstation model, using idle workstations, the processor pool model, a hybrid model)
8th		**22nd**	Processors allocation(allocation models, design issues for processor allocation algorithm, implementation issues, example)
		23rd	Scheduling in distributed systems
		24th	Real-time DS(introduction, design issues, real time communication, real time scheduling)
9th		**25th**	Distributed file systems
		26th	Distributed file system design (The file service interface, the directory server interface, semantics of file sharing)
		27th	Distributed file system implementation (file usage, system structure, caching, replication)
10th		**28th**	Trends in distributed file systems (new hardware, scalability, wide area networking, mobile users, fault tolerance, multimedia)
		29th	**Revision/ Test**
		30th	**Assignment 3**
11th		**31st**	Distributed shared memory (introduction)
		32nd	What is shared memory? (on chip memory, bus-based multiprocessors, ring based, switch based, NUMA multiprocessors)
		33rd	Consistency models (strict consistency, sequential consistency, casual consistency, PRAM and processor, weak consistency, release and entry consistency)
12th		**34th**	Page based distributed shared memory (basic design, replication, granularity)
		35th	Achieving sequential consistency, finding the owner, finding the copies, page replacement
		36th	Shared-variable distributed shared memory (munin, midway)
		37th	Case study MACH

13th		**38th**	Introduction to MACH (history of MACH, goals of MACH, the MACH microkernel)
		39th	Process management in MACH (Processes, threads, scheduling)
14th		**40th**	Memory management in MACH (Virtual memory, memory sharing, external memory managers, distributed shared memory in MACH)
		41th	Communication in MACH (Ports, sending and receiving messages, the network message server)
		42nd	UNIX emulation in MACH
15th		**43rd**	**Assignment 4**
16th		**PREVIOUS YEAR PAPER DISCUSSION**	
17th		**REVISION**	

Distributed System

Course code	PCC-ADS-308G				
Category	Professional Core Course				
Course title	Distributed System				
Scheme and Credits	L	T	P	Credits	Semester 6th
	3	0	0	3	
Class work	25 Marks				
Exam	75 Marks				
Total	100 Marks				
Duration of Exam	3 Hours				

Note: Examiner will be required to set NINE questions in all. Question Number 1 willconsist of total 8 parts (short-answer type questions) covering the entire syllabus and will carry 20marks. In addition to the compulsory question there will be four units i.e. Unit-I to Unit-IV. Examiner will set two questions from each Unit of the syllabus and each question will carry20 marks. Student will be required to attempt FIVE questions in all. Question Number 1 will be compulsory. In addition to compulsory question, student will have to attempt four more questions selecting one quest ion from each Unit.

Course Outcomes:

1. To analyze the current popular distributed systems such as peer-to-peer (P2P) systemswill also be analyzed.
2. To know about Shared Memory Techniques.
3. Have Sufficient knowledge about file access.
4. Have knowledge of Synchronization and Deadlock.

UNIT-I

Introduction : Introduction to Distributed System, Goals of Distributed system, Hardware and Software concepts , Design issues. Communication in distributed system: Layered protocols, ATM networks, Client – Server model, Remote Procedure Calls and Group Communication. Middleware and Distributed Operating Systems.

UNIT-II

Synchronization in Distributed System: Clock synchronization, Mutual Exclusion, Election algorithm, the Bully algorithm, a Ring algorithm, Atomic Transactions, Deadlock in Distributed Systems, Distributed Deadlock Prevention, Distributed Deadlock Detection .

UNIT-III

Processes and Processors in distributed systems: Threads, System models, Processors Allocation, Scheduling in Distributed System, Real Time Distributed

Systems.

Distributed file systems: Distributed file system Design, Distributed file system Implementation, Trends in Distributed file systems.

UNIT-IV

Distributed Shared Memory: What is shared memory, Consistency models, Page baseddistributed shared memory, shared variables distributed shared memory.

Case study MACH: Introduction to MACH, process management in MACH,communication in MACH, UNIX emulation in MACH.

Text Book:

1 Distributed Operating System – Andrew S. Tanenbaum, PHI. 2 Operating System Concepts ,P.S.Gill, Firewall Media

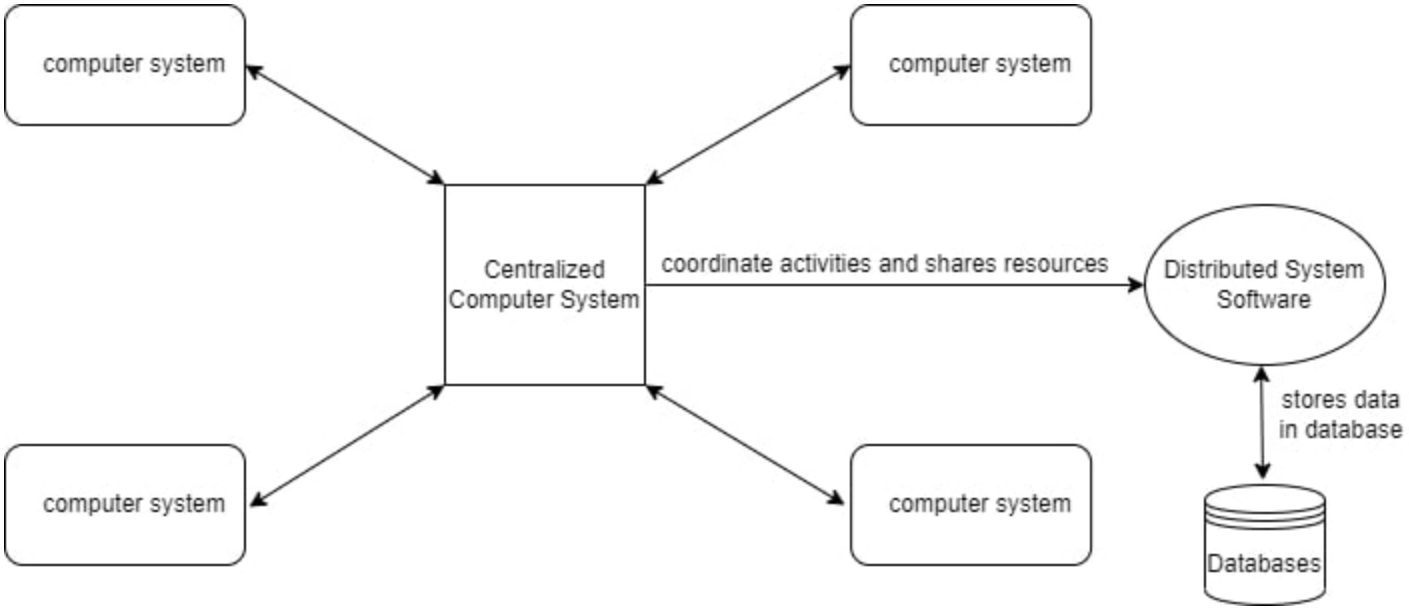
UNIT-1

Introduction to distributed system

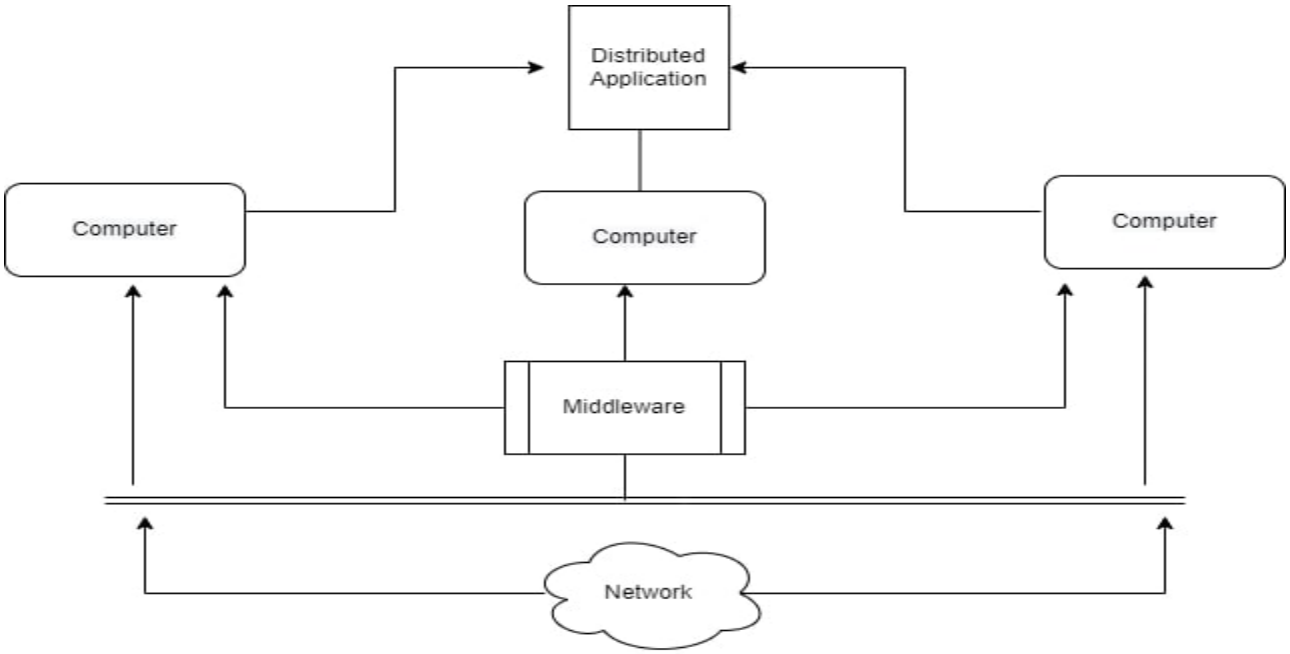
Distributed System is a collection of autonomous computer systems that are physically separated but are connected by a centralized computer network that is equipped with distributed system software. The autonomous computers will communicate among each system by sharing resources and files and performing the tasks assigned to them.

Example of Distributed System:

Any Social Media can have its Centralized Computer Network as its Headquarters and computer systems that can be accessed by any user and using their services will be the Autonomous Systems in the Distributed System Architecture.



- **Distributed System Software:** This Software enables computers to coordinate their activities and to share the resources such as Hardware, Software, Data, etc.
- **Database:** It is used to store the processed data that are processed by each Node/System of the Distributed systems that are connected to the Centralized network.



- As we can see that each Autonomous System has a common Application that can have its own data that is shared by the Centralized Database System.
- To Transfer the Data to Autonomous Systems, Centralized System should be having a Middleware Service and should be connected to a Network.
- Middleware Services enable some services which are not present in the local systems or centralized system default by acting as an interface between the Centralized System and the local systems. By using components of Middleware Services systems communicate and manage data.
- The Data which is been transferred through the database will be divided into segments or modules and shared with Autonomous systems for processing.
- The Data will be processed and then will be transferred to the Centralized system through the network and will be stored in the database.

Characteristics of Distributed System:

- **Resource Sharing:** It is the ability to use any Hardware, Software, or Data anywhere in the System.
- **Openness:** It is concerned with Extensions and improvements in the system (i.e., How openly the software is developed and shared with others)
- **Concurrency:** It is naturally present in Distributed Systems, that deal with the same activity or functionality that can be performed by separate users who are in remote locations. Every local system has its independent Operating Systems and Resources.
- **Scalability:** It increases the scale of the system as a number of processors communicate with more users by accommodating to improve the responsiveness of the system.
- **Fault tolerance:** It cares about the reliability of the system if there is a failure in Hardware or Software, the system continues to operate properly without degrading the performance the system.
- **Transparency:** It hides the complexity of the Distributed Systems to the Users and Application programs as there should be privacy in every system.
- **Heterogeneity:** Networks, computer hardware, operating systems, programming languages, and developer implementations can all vary and differ among dispersed system components.

Advantages of Distributed System:

- Applications in Distributed Systems are Inherently Distributed Applications.
- Information in Distributed Systems is shared among geographically distributed users.
- Resource Sharing (Autonomous systems can share resources from remote locations).
- It has a better price performance ratio and flexibility.
- It has shorter response time and higher throughput.
- It has higher reliability and availability against component failure.
- It has extensibility so that systems can be extended in more remote locations and also incremental growth.

Disadvantages of Distributed System:

- Relevant Software for Distributed systems does not exist currently.
- Security possess a problem due to easy access to data as the resources are shared to multiple systems.
- Networking Saturation may cause a hurdle in data transfer i.e., if there is a lag in the network then the user will face a problem accessing data.
- In comparison to a single user system, the database associated with distributed systems is much more complex and challenging to manage.
- If every node in a distributed system tries to send data at once, the network may become overloaded.

Applications Area of Distributed System:

- **Finance and Commerce:** Amazon, eBay, Online Banking, E-Commerce websites.
- **Information Society:** Search Engines, Wikipedia, Social Networking, Cloud Computing.
- **Cloud Technologies:** AWS, Salesforce, Microsoft Azure, SAP.
- **Entertainment:** Online Gaming, Music, youtube.
- **Healthcare:** Online patient records, Health Informatics.
- **Education:** E-learning.
- **Transport and logistics:** GPS, Google Maps.
- **Environment Management:** Sensor technologies.

Challenges of Distributed Systems:

While distributed systems offer many advantages, they also present some challenges that must be addressed. These challenges include:

- Network latency: The communication network in a distributed system can introduce latency, which can affect the performance of the system.
- Distributed coordination: Distributed systems require coordination among the nodes, which can be challenging due to the distributed nature of the system.
- Security: Distributed systems are more vulnerable to security threats than centralized systems due to the distributed nature of the system.
- Data consistency: Maintaining data consistency across multiple nodes in a distributed system can be challenging.

Goals of distributed system

1. Boosting Performance

The distributed system tries to make things faster by dividing a bigger task into small chunks and finally processing them simultaneously in different computers. It's just like a group of people working together on a project. For example, when we try to search for anything on the internet the search engine distributes the work among several servers and then retrieve the result and display the webpage in a few seconds.

2. Enhancing Reliability

Distributed system ensures reliability by minimizing the load of individual computer failure. If one computer gets some failure then other computers try to keep the system running smoothly. For Example, when we search for something in social media if one server gets an issue then also we are able to access photos, and posts because they switch the server quickly.

3. Scaling for the Future

Distributed systems are experts at handling increased demands. They manage the demands by incorporating more and more computers into the system. This way they run everything smoothly and can handle more users.

4. Resourceful Utilization

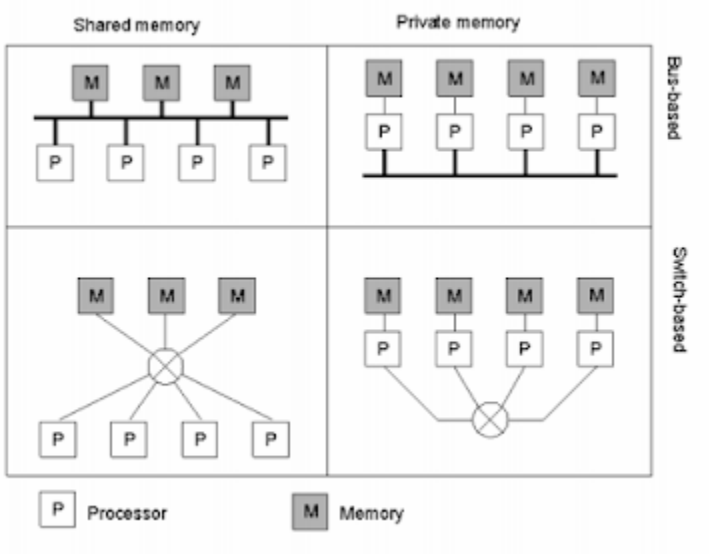
Resource Utilization is one of the most prominent features of a Distributed system. Instead of putting a load on one computer, they distribute the task among the other available resource. This ensures that work will be done by utilizing every resource.

Hardware & Software concept

Hardware concepts :-

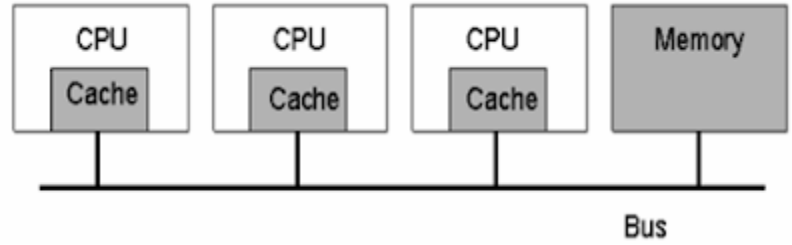
Hardware in Distributed Systems can be organized in several different ways:

- Shared Memory (Multiprocessors , which have a single address space).
- Private Memory (Multicomputers, each CPU has a direct connection to its local memory).



Multiprocessors – Bus Based

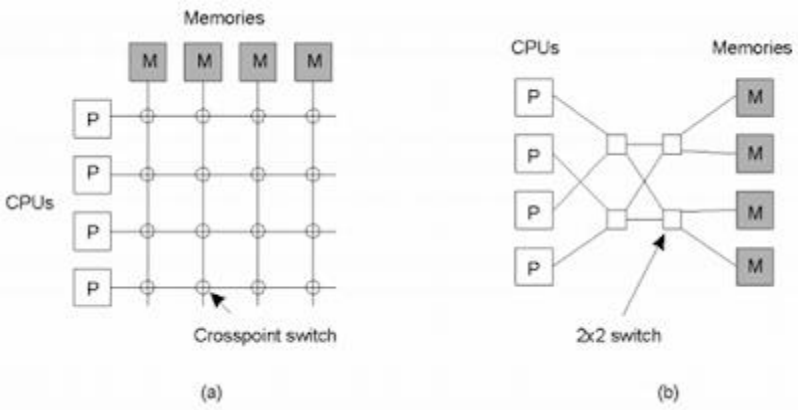
- Have limited scalability
- Cache Memory help avoid bus overloading.



Multiprocessors – Switch Based

- Different CPUs can access different memories simultaneously
- The number of switches limits the number of CPUs that can access memory simultaneously

a) A crossbar switch
b) An omega switching network



Multicomputers :-

Homogeneous:

- All CPUs and memory are identical;
- Connected through a broadcast shared multi access network (like Ethernet) in bus based systems;
- Messages routed through an interconnectionn network in switch-based multicomputers (e.g., grids, hipercubes…).

Heterogeneous:

- The most usual topology;
- Computers may vary widely with respect to processor type, memory size, I/O bandwidth;
- Connections are also diverse (a single multicomputer can simultaneously use LANs, Wide Area ATM, and frame relay networks);
- Sophisticated software is needed to build applications due to the inherent heterogeneity;
- Examples: SETI@home, WWW…

Software Concepts :-

Uniprocessor Operating Systems

- An OS acts as a resource manager or an arbitrator : Manages CPU, I/O devices, memory
- OS provides a virtual interface that is easier to use than hardware
- Structure of uniprocessor operating systems: Monolithic (e.g., MS-DOS, early UNIX)
- One large kernel that handles everything: Layered design
- Functionality is decomposed into N layers
- Each layer uses services of layer N-1 and implements new service(s) for layer N+1

Design issues in distributed system

1. **Heterogeneity**: Heterogeneity is applied to the network, computer hardware, operating system, and implementation of different developers. A key component of the heterogeneous distributed system client-server environment is middleware. Middleware is a set of services that enables applications and end-user to interact with each other across a heterogeneous distributed system.
2. **Openness**: The openness of the distributed system is determined primarily by the degree to which new resource-sharing services can be made available to the users. Open systems are characterized by the fact that their key interfaces are published. It is based on a uniform communication mechanism and published interface for access to shared resources. It can be constructed from heterogeneous hardware and software.
3. **Scalability**: The scalability of the system should remain efficient even with a significant increase in the number of users and resources connected. It shouldn't matter if a program has 10 or 100 nodes; performance shouldn't vary. A distributed system's scaling requires consideration of a number of elements, including size, geography, and management.
4. **Security**: The security of an information system has three components Confidentially, integrity, and availability. Encryption protects shared resources and keeps sensitive information secrets when transmitted.
5. **Failure Handling**: When some faults occur in hardware and the software program, it may produce incorrect results or they may stop before they have completed the intended computation so corrective measures should to implemented to handle this case. Failure handling is difficult in distributed systems because the failure is partial i, e, some components fail while others continue to function.
6. **Concurrency**: There is a possibility that several clients will attempt to access a shared resource at the same time. Multiple users make requests on the same resources, i.e. read, write, and update. Each resource must be safe in a concurrent environment. Any object that represents a shared resource in a distributed system must ensure that it operates correctly in a concurrent environment.
7. **Transparency**: Transparency ensures that the distributed system should be perceived as a single entity by the users or the application programmers rather than a collection of autonomous systems, which is cooperating. The user should be unaware of where the services are located and the transfer from a local machine to a remote one should be transparent.

Layered Protocols:

Due to the absence of shared memory, all communication in distributed systems is based on message passing. When process A wants to communicate with process B,PYit first builds a message in its own address space. Then it executes a system call that causes the operating system to fetch the message and send it over the network to B. Although this basic idea sounds simple enough, in order to prevent chaos, A and B have to agree on the meaning of the bits being sent.

The telephone is a connection oriented communication system. With connectionless protocols, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of connectionless communication. With computers, both connection-oriented and connectionless communication are common. In the OSI model, communication is divided up into seven levels or layers, as shown in Fig. 2-1. Each layer deals with one specific aspect of the communication.

In the OSI model, when process A on machine 1 wants to communicate with process B on machine 2, it builds a message and passes the message to the application layer on its machine. This layer might be a library procedure, for example, but it could also be implemented in some other way (e.g., inside the operating system, on an external coprocessor chip, etc.). The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer in turn adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits bottom, the physical layer actually transmits the message, which by now might look as

shown in Fig. 2-2. When the message arrives at machine 2, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process B, which may reply to it using the reverse path. The information in the layer n header is used for the layer n protocol.

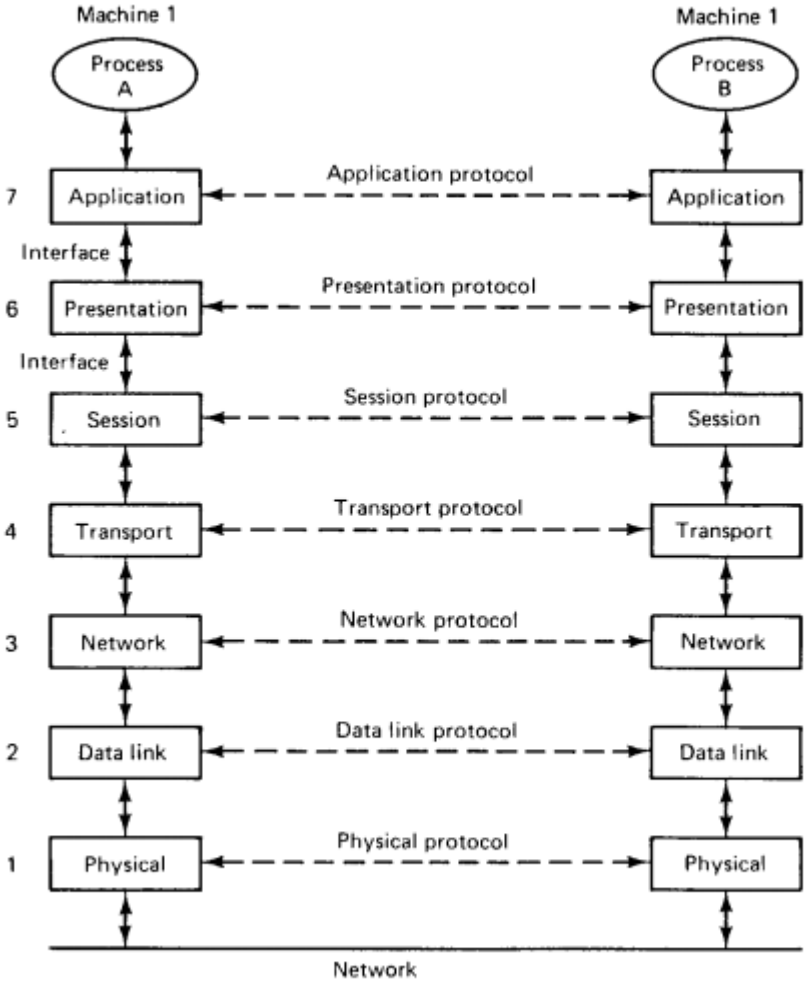


Fig. 2-1. Layers, interfaces, and protocols in the OSI model.

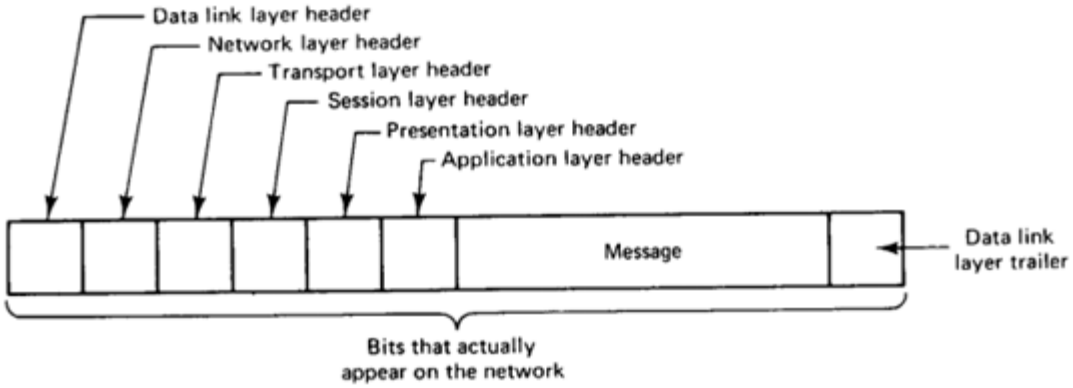


Fig. 2-2. A typical message as it appears on the network.

The Physical Layer:-

The physical layer is concerned with transmitting the Os and 1 s. How many volts to use for 0 and 1, how many bits per second can be sent, and whether transmission can take place in both directions simultaneously are key issues in the physical layer.

The Data Link Layer:-

The physical layer just sends bits. As long as no errors occur, all is well. However, real communication networks are subject to errors, so some mechanism is needed to detect and correct them. This mechanism is the main task of the data link layer. What it does is to group the bits into units, sometimes called frames, and see that each frame is correctly received. The data link layer does its work by putting a special bit pattern on the start and end of each frame, to mark them, as well as computing a checksum by adding up all the bytes in the frame in a certain way. The data link layer appends the checksum to the frame. When the frame arrives, the receiver recomputes the checksum from the data and compares the result to the checksum following the frame. If they agree, the frame is considered correct and is accepted. It they disagree, the receiver asks the sender to retransmit it.

The Network Layer:

On a LAN, there is usually no need for the sender to locate the receiver. It just puts the message out on the network and the receiver takes it off. A widearea network, however, consists of a large number of machines, each with some number of lines to other machines, rather like a large-scale map showing major cities and roads connecting them. For a message to get from the sender to the receiver it may have to make a number of hops, at each one choosing an outgoing line to use. The question of how to choose the best path is called routing, and is the primary task of the network layer. The problem is

complicated by the fact that the shortest route is not always the best route**.**

Why ATM networks?

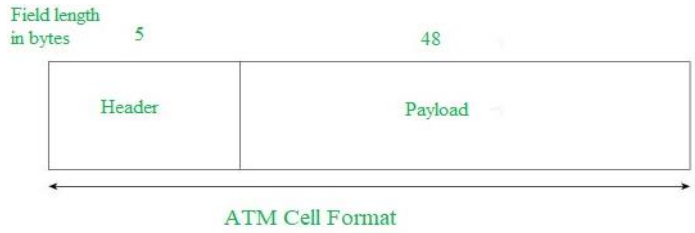
1. Driven by the integration of services and performance requirements of both telephony and data networking: "broadband integrated service vision" (B-ISON).

2. Telephone networks support a single quality of service and are expensive to boot.

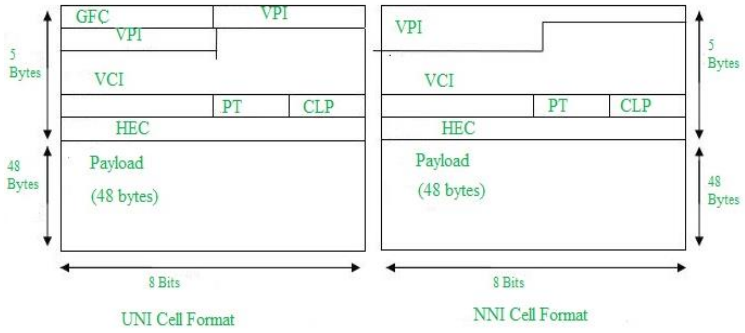
3. Internet supports no quality of service but is flexible and cheap.

4. ATM networks were meant to support a range of service qualities at a reasonable cost- intended to subsume both the telephone network and the Internet.

Asynchronous Transfer Mode (ATM):
It is an International Telecommunication Union- Telecommunications Standards Section (ITU-T) efficient for call relay and it transmits all information including multiple service types such as data, video, or voice which is conveyed in small fixed-size packets called cells. Cells are transmitted asynchronously and the network is connection-oriented. ATM is a technology that has some event in the development of broadband ISDN in the 1970s and 1980s, which can be considered an evolution of packet switching. *Each cell is 53 bytes long* – 5 bytes header and 48 bytes payload. Making an ATM call requires first sending a message to set up a connection.
Subsequently, all cells follow the same path to the destination. It can handle both constant rate traffic and variable rate traffic. Thus it can carry multiple types of traffic with **end-to-end** quality of service. ATM is independent of a transmission medium, they may be sent on a wire or fiber by themselves or they may also be packaged inside the payload of other carrier systems. ATM networks use "Packet" or "cell" Switching with virtual circuits. Its design helps in the implementation of high-performance multimedia networking.
ATM Cell Format –
As information is transmitted in ATM in the form of fixed-size units called **cells**. As known already each cell is 53 bytes long which consists of a 5 bytes header and 48 bytes payload.



Asynchronous Transfer Mode can be of two format types which are as follows:



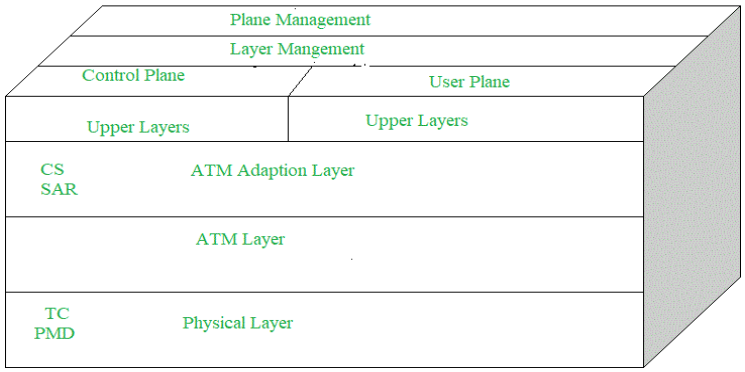
1. **UNI Header:** This is used within private networks of ATMs for communication between ATM endpoints and ATM switches. It includes the Generic Flow Control (GFC) field.

2. **NNI Header:** is used for communication between ATM switches, and it does not include the Generic Flow Control(GFC) instead it includes a Virtual Path Identifier (VPI) which occupies the first 12 bits.

Working of ATM:
ATM standard uses two types of connections. i.e., Virtual path connections (VPCs) which consist of Virtual channel connections (VCCs) bundled together which is a basic unit carrying a single stream of cells from user to user. A virtual path can be created end-to-end across an ATM network, as it does not rout the cells to a particular virtual circuit. In case of major failure, all cells belonging to a particular virtual path are routed the same way through the ATM network, thus helping in faster recovery.
Switches connected to subscribers use both VPIs and VCIs to switch the cells which are Virtual Path and Virtual Connection switches that can have different virtual channel connections between them, serving the purpose of creating a *virtual trunk* between the switches which can be handled as a single entity. Its basic operation is straightforward by looking up the connection value in the local translation table determining the outgoing port of the connection and the new VPI/VCI value of connection on that link.
ATM vs DATA Networks (Internet) –

- ATM is a “virtual circuit” based: the path is reserved before transmission. While Internet Protocol (IP) is connectionless and end-to-end resource reservations are not possible. RSVP is a new signaling protocol on the internet.
- ATM Cells: Fixed or small size and Tradeoff is between voice or data. While IP packets are of variable size.
- Addressing: ATM uses 20-byte global NSAP addresses for signaling and 32-bit locally assigned labels in cells. While IP uses 32-bit global addresses in all packets.

ATM Layers:



1. **ATM Adaption Layer (AAL) –**
   It is meant for isolating higher-layer protocols from details of ATM processes and prepares for conversion of user data into cells and segments it into 48-byte cell payloads. AAL protocol excepts transmission from upper-layer services and helps them in mapping applications, e.g., voice, data to ATM cells.

2. **Physical Layer –**
   It manages the medium-dependent transmission and is divided into two parts physical medium-dependent sublayer and transmission convergence sublayer. The main functions are as follows:
   - It converts cells into a bitstream.
   - It controls the transmission and receipt of bits in the physical medium.
   - It can track the ATM cell boundaries.
   - Look for the packaging of cells into the appropriate type of frames.
3. **ATM Layer –**
   It handles transmission, switching, congestion control, cell header processing, sequential delivery, etc., and is responsible for simultaneously sharing the virtual circuits over the physical link known as cell multiplexing and passing cells through an ATM network known as cell relay making use of the VPI and VCI information in the cell header.

ATM Applications:

1. **ATM WANs –**
   It can be used as a WAN to send cells over long distances, a router serving as an end-point between ATM network and other networks, which has two stacks of the protocol.

2. **Multimedia virtual private networks and managed services –**
   It helps in managing ATM, LAN, voice, and video services and is capable of full-service virtual private networking, which includes integrated access to multimedia.

3. **Frame relay backbone –**
   Frame relay services are used as a networking infrastructure for a range of data services and enabling frame-relay ATM service to Internetworking services.

4. **Residential broadband networks –**
   ATM is by choice provides the networking infrastructure for the establishment of residential broadband services in the search of highly scalable solutions.

5. **Carrier infrastructure for telephone and private line networks –**
   To make more effective use of SONET/SDH fiber infrastructures by building the ATM infrastructure for carrying the telephonic and private-line traffic.

Client-Server Model

The Client-server model is a distributed application structure that partitions task or workload between the providers of a

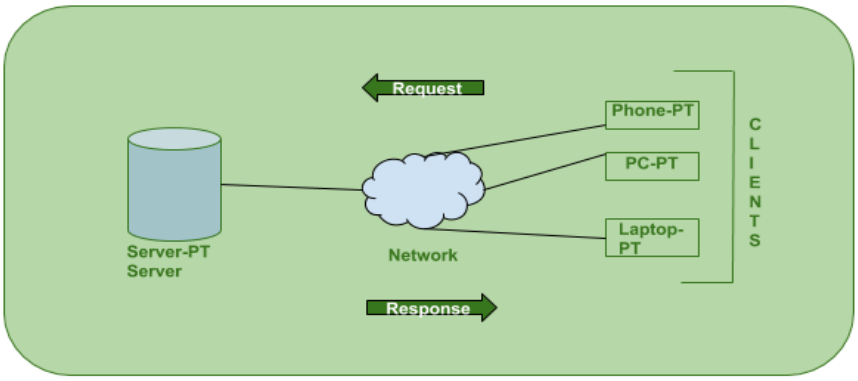
resource or service, called servers, and service requesters called clients. In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested process and deliver the data packets requested back to the client. Clients do not share any of their resources. Examples of Client-Server Model are Email, World Wide Web, etc.

How the Client-Server Model works ?

In this article we are going to take a dive into the **Client-Server** model and have a look at how the **Internet** works via, web browsers. This article will help us in having a solid foundation of the WEB and help in working with WEB technologies with ease.

- **Client:** When we talk the word **Client**, it mean to talk of a person or an organization using a particular service. Similarly in the digital world a **Client** is a computer (**Host**) i.e. capable of receiving information or using a particular service from the service providers (**Servers**).
- **Servers:** Similarly, when we talk the word **Servers**, It mean a person or medium that serves something. Similarly in this digital world a **Server** is a remote computer which provides information (data) or access to particular services.

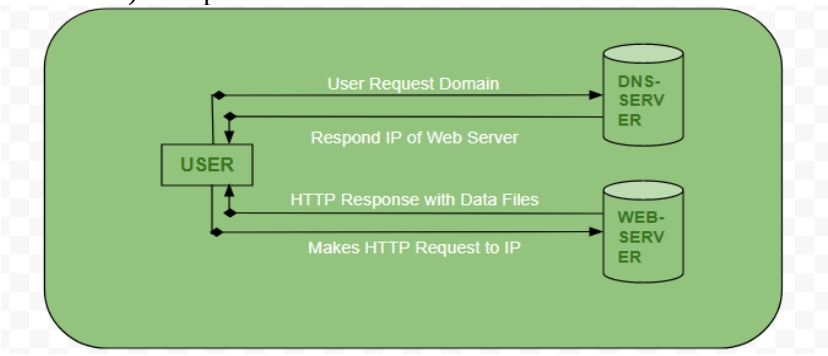
So, its basically the **Client** requesting something and the **Server** serving it as long as its present in the database.



How the browser interacts with the servers ?

There are few steps to follow to interacts with the servers a client.

- User enters the **URL**(Uniform Resource Locator) of the website or file. The Browser then requests the **DNS**(DOMAIN NAME SYSTEM) Server.
- **DNS Server** lookup for the address of the **WEB Server**.
- **DNS Server** responds with the **IP address** of the **WEB Server**.
- Browser sends over an **HTTP/HTTPS** request to **WEB Server's IP** (provided by **DNS server**).
- Server sends over the necessary files of the website.
- Browser then renders the files and the website is displayed. This rendering is done with the help of **DOM** (Document Object Model) interpreter, **CSS** interpreter and **JS Engine** collectively known as the **JIT** or (Just in Time) Compilers.



Advantages of Client-Server model:

- Centralized system with all data in a single place.
- Cost efficient requires less maintenance cost and Data recovery is possible.
- The capacity of the Client and Servers can be changed separately.

Disadvantages of Client-Server model:

- Clients are prone to viruses, Trojans and worms if present in the Server or uploaded into the Server.

- Server are prone to Denial of Service (DOS) attacks.
- Data packets may be spoofed or modified during transmission.
- Phishing or capturing login credentials or other useful information of the user are common and MITM(Man in the Middle) attacks are common.

Remote Procedure Call (RPC) is an interprocess communication technique. The Full form of RPC is Remote Procedure Call. It is used for client-server applications. RPC mechanisms are used when a computer program causes a procedure or subroutine to execute in a different address space, which is coded as a normal procedure call without the programmer specifically coding the details for the remote interaction.

This procedure call also manages low-level transport protocol, such as User Datagram Protocol, Transmission Control Protocol/Internet Protocol etc. It is used for carrying the message data between programs.

Types of RPC

Three types of RPC are:

- Callback RPC
- Broadcast RPC
- Batch-mode RPC

Callback RPC

This type of RPC enables a P2P paradigm between participating processes. It helps a process to be both client and server services.

Functions of Callback RPC:

- Remotely processed interactive application problems
- Offers server with clients handle
- Callback makes the client process wait
- Manage callback deadlocks
- It facilitates a peer-to-Peer paradigm among participating processes.

Broadcast RPC

Broadcast RPC is a client's request, that is broadcast on the network, processed by all servers which have the method for processing that request.

Functions of Broadcast RPC:

- Allows you to specify that the client's request message has to be broadcasted.
- You can declare broadcast ports.
- It helps to reduce the load on the physical network

Batch-mode RPC

Batch-mode RPC helps to queue, separate RPC requests, in a transmission buffer, on the client-side, and then send them on a network in one batch to the server.

Functions of Batch-mode RPC:

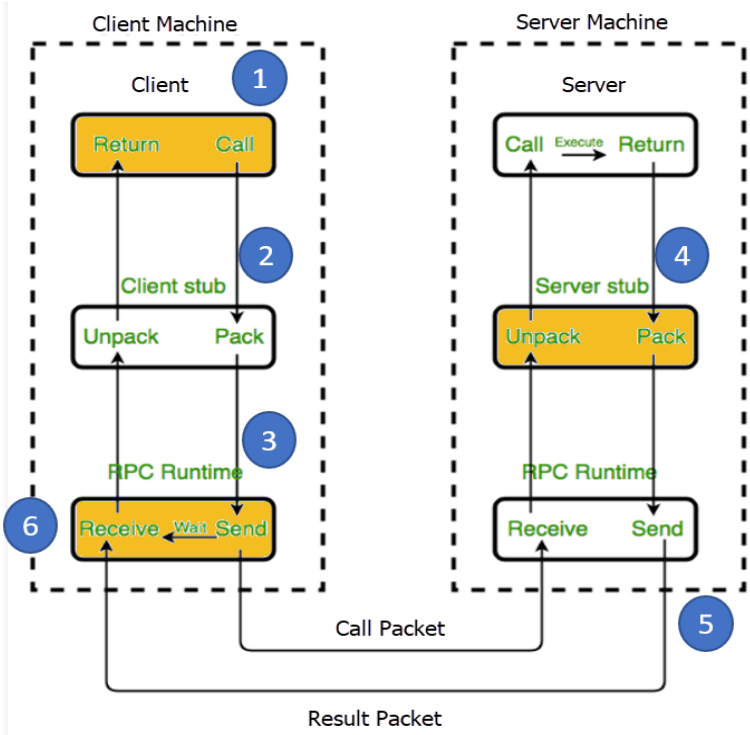
- It minimizes overhead involved in sending a request as it sends them over the network in one batch to the server.

- This type of RPC protocol is only efficient for the application that needs lower call rates.
- It needs a reliable transmission protocol.

RPC Architecture

RPC architecture has mainly five components of the program:

1. **Client**
2. **Client Stub**
3. **RPC Runtime**
4. **Server Stub**
5. **Server**



RPC Architecture

How RPC Works?

Following steps take place during the RPC process:

Step 1) The client, the client stub, and one instance of RPC run time execute on the client machine.

Step 2) A client starts a client stub process by passing parameters in the usual way. The client stub stores within the client's own address space. It also asks the local RPC Runtime to send back to the server stub.

Step 3) In this stage, RPC accessed by the user by making regular Local Procedural Cal. RPC Runtime manages the transmission of messages between the network across client and server. It also performs the job of retransmission, acknowledgment, routing, and encryption.

Step 4) After completing the server procedure, it returns to the server stub, which packs (marshalls) the return values into a message. The server stub then sends a message back to the transport layer.

Step 5) In this step, the transport layer sends back the result message to the client transport layer, which returns back a message to the client stub.

Step 6) In this stage, the client stub demarshalls (unpack) the return parameters, in the resulting packet, and the execution process returns to the caller.

Characteristics of RPC

Here are the essential characteristics of RPC:

- The called procedure is in another process, which is likely to reside in another machine.
- The processes do not share address space.
- Parameters are passed only by values.
- RPC executes within the environment of the server process.
- It doesn't offer access to the calling procedure's environment.

Features of RPC

Here are the important features of RPC:

- Simple call syntax
- Offers known semantics
- Provide a well-defined interface
- It can communicate between processes on the same or different machines

Advantages of RPC

Here are Pros/benefits of RPC:

- RPC method helps clients to communicate with servers by the conventional use of procedure calls in high-level languages.
- RPC method is modeled on the local procedure call, but the called procedure is most likely to be executed in a different process and usually a different computer.
- RPC supports process and thread-oriented models.
- RPC makes the internal message passing mechanism hidden from the user.
- The effort needs to re-write and re-develop the code is minimum.
- Remote procedure calls can be used for the purpose of distributed and the local environment.
- It commits many of the protocol layers to improve performance.
- RPC provides abstraction. For example, the message-passing nature of network communication remains hidden from the user.
- RPC allows the usage of the applications in a distributed environment that is not only in the local environment.
- With RPC code, re-writing and re-developing effort is minimized.
- Process-oriented and thread-oriented models support by RPC.

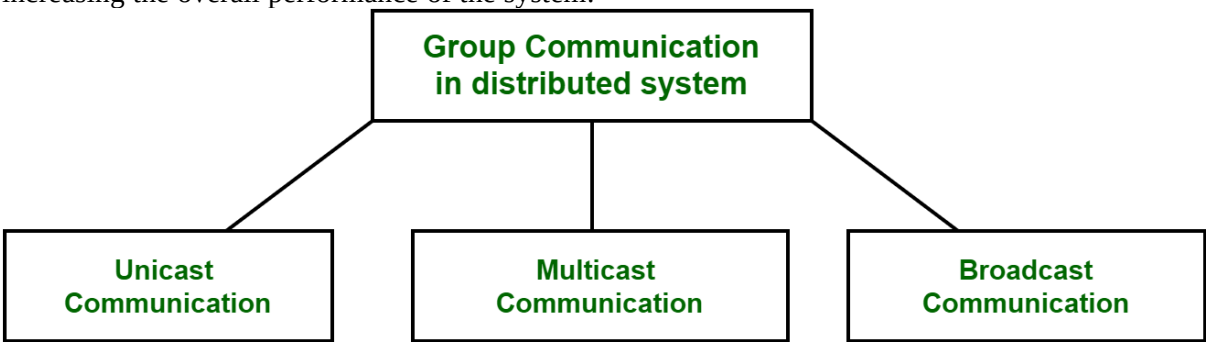
Disadvantages of RPC

Here are the cons/drawbacks of using RPC:

- Remote Procedure Call Passes Parameters by values only and pointer values, which is not allowed.
- Remote procedure calling (and return) time (i.e., overheads) can be significantly lower than that for a local procedure.
- This mechanism is highly vulnerable to failure as it involves a communication system, another machine, and another process.
- RPC concept can be implemented in different ways, which is can't standard.
- Not offers any flexibility in RPC for hardware architecture as It is mostly interaction-based.

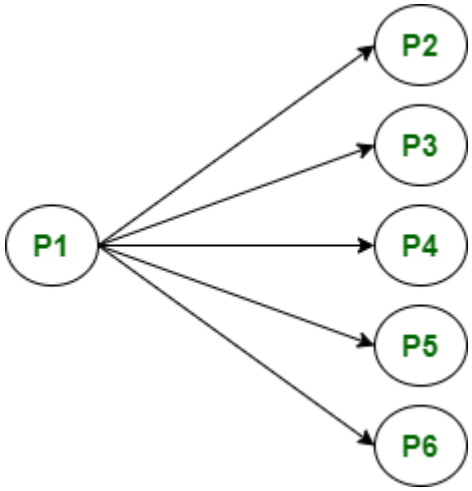
- The cost of the process is increased because of a remote procedure call.

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.



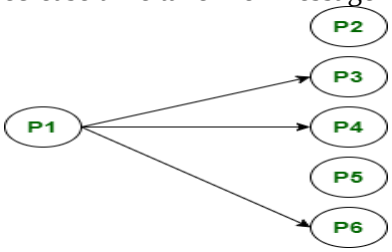
Types of Group Communication in a Distributed System:

- **Broadcast Communication :** When the host process tries to communicate with every process in a distributed system at same time. Broadcast communication comes in handy when a common stream of information is to be delivered to each and every process in 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 a large number of processes and cannot treat a specific process individually.



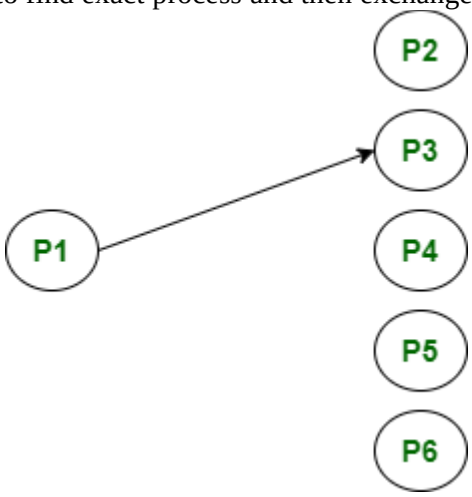
A broadcast Communication: P1 process communicating with every process in the system

- **Multicast Communication :** 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.



A multicast Communication: P1 process communicating with only a group of the process in the system

- **Unicast Communication :** 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 has to treat a specific process only. However, it leads to overheads as it has to find exact process and then exchange information/data.



A unicast Communication: P1 process communicating with only P3 process

What is middleware?

middleware is software that extends the operating system's services. It allows the many components of a distributed approach to interact and handle data.

It comprises web servers, application servers, messaging, and other technologies that aid in creating and delivering applications.

Middleware is used in distributed systems to simplify software development by doing the following:

- To hide the complexities of distributed applications.
- To hide the differences between hardware, operating systems, and protocols.
- Provide a standardized, high-level interface for creating interoperable, reusable, and portable programs.
- Provide a collection of standard services that reduce duplication of effort and improve application cooperation.

An illustration of how middleware works in distributed systems

How does middleware work?

Requests sent over the network try to interact with back-end data. This data might be as essential as a display image or a movie to play or as sophisticated as a history of banking activities.

The requested data can take several forms and be stored in several ways, including arriving from a file server, being pulled from a message queue, or being saved in a database. Middleware's function is to facilitate and simplify access to such back-end resources. Middleware programs often provide a messaging service via which applications may send data, such as SOAP, REST, or JavaScript object notation (JSON).

Types of middleware

There are several instances of middleware, each designed to perform specific duties in integrating applications, online, and cloud services. Here are some types of middleware:

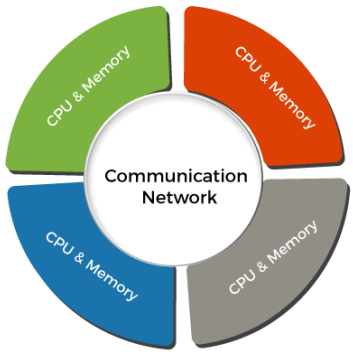
- **Messaging middleware** allows dispersed programs and services to communicate with one another.

- **Object or ORB middleware** allows software components or objects, such as containers, to communicate and interact with a program across distributed networks.
- **RPC middleware** is a protocol that allows one application to request a service from another program on a different machine or network.
- **Middleware for data or databases** allows direct access to and interaction with databases; it often includes SQL database software.
- **Transactional middleware** guarantees that transactions go from one step to the next by monitoring transaction processes.
- **Content-centric middleware** delivers client-side requests for specific content, which is analogous to publish/subscribe middleware such as Apache Kafka.
- **Embedded middleware** allows for communication and integration between embedded applications and real-time operating systems

Distributed Operating System

A distributed operating system **(DOS)** is an essential type of operating system. Distributed systems use many central processors to serve multiple real-time applications and users. As a result, data processing jobs are distributed between the processors.

It connects multiple computers via a single communication channel. Furthermore, each of these systems has its own processor and memory. Additionally, these **CPUs** communicate via high-speed buses or telephone lines. Individual systems that communicate via a single channel are regarded as a single entity. They're also known as **loosely coupled systems**.



This operating system consists of numerous computers, nodes, and sites joined together via **LAN/WAN** lines. It enables the distribution of full systems on a couple of center processors, and it supports many real-time products and different users. Distributed operating systems can share their computing resources and I/O files while providing users with virtual machine abstraction.

Types of Distributed Operating System

There are various types of Distributed Operating systems. Some of them are as follows:

1. **Client-Server Systems**
2. **Peer-to-Peer Systems**
3. **Middleware**
4. **Three-tier**
5. **N-tier**

Client-Server System

This type of system requires the client to request a resource, after which the server gives the requested resource. When a client connects to a server, the server may serve multiple clients at the same time.

Client-Server Systems are also referred to as "Tightly Coupled Operating Systems". This system is primarily intended for multiprocessors and homogenous multicomputer. Client-Server Systems function as a centralized server since they approve all requests issued by client systems.

Server systems can be divided into two parts:

1. Computer Server System

This system allows the interface, and the client then sends its own requests to be executed as an action. After completing the activity, it sends a back response and transfers the result to the client.

2. File Server System

It provides a file system interface for clients, allowing them to execute actions like file creation, updating, deletion, and more.

Peer-to-Peer System

The nodes play an important role in this system. The task is evenly distributed among the nodes. Additionally, these nodes can share data and resources as needed. Once again, they require a network to connect.

The Peer-to-Peer System is known as a "Loosely Couple System". This concept is used in computer network applications since they contain a large number of processors that do not share memory or clocks. Each processor has its own local memory, and they interact with one another via a variety of communication methods like telephone lines or high-speed buses.

Middleware

Middleware enables the interoperability of all applications running on different operating systems. Those programs are capable of transferring all data to one other by using these services.

Three-tier

The information about the client is saved in the intermediate tier rather than in the client, which simplifies development. This type of architecture is most commonly used in online applications.

N-tier

When a server or application has to transmit requests to other enterprise services on the network, n-tier systems are used.

Features of Distributed Operating System

There are various features of the distributed operating system. Some of them are as follows:

Openness

It means that the system's services are freely displayed through interfaces. Furthermore, these interfaces only give the service syntax. For example, the type of function, its return type, parameters, and so on. Interface Definition Languages are used to create these interfaces (IDL).

Scalability

It refers to the fact that the system's efficiency should not vary as new nodes are added to the system. Furthermore, the performance of a system with 100 nodes should be the same as that of a system with 1000 nodes.

Resource Sharing

Its most essential feature is that it allows users to share resources. They can also share resources in a secure and controlled manner. Printers, files, data, storage, web pages, etc., are examples of shared resources.

Flexibility

A DOS's flexibility is enhanced by modular qualities and delivers a more advanced range of high-level services. The kernel/ microkernel's quality and completeness simplify the implementation of such services.

Transparency

It is the most important feature of the distributed operating system. The primary purpose of a distributed operating system is to hide the fact that resources are shared. Transparency also implies that the user should be unaware that the resources he is accessing are shared. Furthermore, the system should be a separate independent unit for the user.

Heterogeneity

The components of distributed systems may differ and vary in operating systems, networks, programming languages, computer hardware, and implementations by different developers.

Fault Tolerance

Fault tolerance is that process in which user may continue their work if the software or hardware fails.

Examples of Distributed Operating System

There are various examples of the distributed operating system. Some of them are as follows:

Solaris

It is designed for the SUN multiprocessor workstations

OSF/1

It's compatible with Unix and was designed by the Open Foundation Software Company.

Micros

The MICROS operating system ensures a balanced data load while allocating jobs to all nodes in the system.

DYNIX

It is developed for the Symmetry multiprocessor computers.

Locus

It may be accessed local and remote files at the same time without any location hindrance.

Mach

It allows the multithreading and multitasking features.

Applications of Distributed Operating System

There are various applications of the distributed operating system. Some of them are as follows:

Network Applications

DOS is used by many network applications, including the Web, peer-to-peer networks, multiplayer web-based games, and virtual communities.

Telecommunication Networks

DOS is useful in phones and cellular networks. A DOS can be found in networks like the Internet, wireless sensor networks, and routing algorithms.

Parallel Computation

DOS is the basis of systematic computing, which includes cluster computing and grid computing, and a variety of volunteer computing projects.

Real-Time Process Control

The real-time process control system operates with a deadline, and such examples include aircraft control systems.

Advantages and Disadvantages of Distributed Operating System

There are various advantages and disadvantages of the distributed operating system. Some of them are as follows:

Advantages

There are various advantages of the distributed operating system. Some of them are as follow:

1. It may share all resources (CPU, disk, network interface, nodes, computers, and so on) from one site to another, increasing data availability across the entire system.
2. It increases the speed of data exchange from one site to another site.
3. It is an open system since it may be accessed from both local and remote locations.
4. It helps in the reduction of data processing time.
5. Most distributed systems are made up of several nodes that interact to make them fault-tolerant. If a single machine fails, the system remains operational.

Disadvantages

There are various disadvantages of the distributed operating system. Some of them are as follows:

1. The system must decide which jobs must be executed when they must be executed, and where they must be executed. A scheduler has limitations, which can lead to underutilized hardware and unpredictable runtimes.
2. It is hard to implement adequate security in DOS since the nodes and connections must be secured.
3. The database connected to a DOS is relatively complicated and hard to manage in contrast to a single-user system.
4. These systems aren't widely available because they're thought to be too expensive.
5. Gathering, processing, presenting, and monitoring hardware use metrics for big clusters can be a real issue.

UNIT-2

Synchronization in Distributed System

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. The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system. Clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

1. **External clock synchronization** is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
2. **Internal clock synchronization** is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

There are 2 types of clock synchronization algorithms: Centralized and Distributed.

1. **Centralized** is the one in which a time server is used as a reference. The single time-server propagates it's time to the nodes, and all the nodes adjust the time accordingly. It is dependent on a single time-server, so if that node fails, the whole system will lose synchronization. Examples of centralized are-Berkeley the Algorithm, Passive Time Server, Active Time Server etc.
2. **Distributed** is the one in which there is no centralized time-server present. Instead, the nodes adjust their time by using their local time and then, taking the average of the differences in time with other nodes. Distributed algorithms overcome the issue of centralized algorithms like scalability and single point failure. Examples of Distributed algorithms are – Global Averaging Algorithm, Localized Averaging Algorithm, NTP (Network time protocol), etc.

Centralized clock synchronization algorithms suffer from two major drawbacks:

1. They are subject to a single-point failure. If the time-server node fails, the clock synchronization operation cannot be performed. This makes the system unreliable. Ideally, a distributed system should be more reliable than its individual nodes. If one goes down, the rest should continue to function correctly.
2. From a scalability point of view, it is generally not acceptable to get all the time requests serviced by a single-time server. In a large system, such a solution puts a heavy burden on that one process.

Distributed algorithms overcome these drawbacks as there is no centralized time-server present. Instead, a simple method for clock synchronization may be to equip each node of the system with a real-time receiver so that each node's clock can be independently synchronized in real-time. Multiple real-time clocks (one for each node) are normally used for this purpose.

Clock Synchronization

Clock synchronization is the mechanism to synchronize the time of all the computers in the distributed environments or system.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 with the same time in synchronized manner you need a clock that has to be synchronized. This process to synchronize data is known as Clock Synchronization.

Synchronization in distributed system is more complicated than in centralized system because of the use of distributed algorithms.

Properties of Distributed algorithms to maintain Clock synchronization:

- Relevant and correct information will be scattered among multiple machines.
- The processes make the decision only on local information.

- Failure of the single point in the system must be avoided.
- No common clock or the other precise global time exists.
- In the distributed systems, the time is ambiguous.

As the distributed systems has its own clocks. The time among the clocks may also vary. So, it is possible to synchronize all the clocks in distributed environment.

Types of Clock Synchronization

- Physical clock synchronization
- Logical clock synchronization
- Mutual exclusion synchronization

Physical Synchronization:

- In physical clock synchronization, All the computers will have their own clocks.
- The physical clocks are needed to adjust the time of nodes. All the nodes in the system can share their local time with all other nodes in the system.
- The time will be set based on UTC (Universal Coordinate Timer).
- The time difference between the two computers is known as “Time drift”. Clock drifts over the time is known as “Skew”. Synchronization is necessary here.

Physical clocks: In physical synchronization, physical clocks are used to time stamp an event on that computer.
If two events, E1 and E2, having different time stamps t1 and t2, the order of the event occurring will be considered and not on the exact time or the day at which they are occur.

Several methods are used to attempt the synchronization of the physical clocks in Distributed synchronization:

1. UTC (Universal coordinate timer)
2. Christian’s algorithm
3. Berkely’s algorithm

Universal Coordinate Time (UTC)

- All the computers are generally synchronized to a standard time called Universal Coordinate Time (UTC).
- UTC is the primary time standard by which the time and the clock are regulated in the world. It is available via radio signals, telephone line and satellites (GPS).

- UTC is broadcasted via the satellites.
- Computer servers and online services with the UTC resources can be synchronized by the satellite broadcast.
- Kept within 0.9 seconds of UTI.

UTO - Mean solar time on Greenwich meridian, Obtained from astronomical observation.

UT1 - UTO corrected for polar motion.

UT2 - UT1 corrected for seasonal variations in earth's rotation.

UTC - Civil time will be measured on an atomic scale.

Christian's Algorithm:

- The simplest algorithm for setting time, it issues a remote procedure call (RPC) to the time sever and obtains the time.
- The machine which send requests to the time server is "d/z" seconds, where d is the maximum difference between the clock and the UTC.
- The time server sends the reply with current UTC when receives the request from the receiver.

Logical clock synchronization

Within the tapestry of distributed systems absolute time often takes a backseat to clock synchronization. Think of clocks as storytellers that prioritize the order of events than their exact timing. These clocks enable the establishment of connections between events like weaving threads of cause and effect. By bringing order and structure into play, task coordination within distributed systems becomes akin to a choreographed dance where steps are sequenced for execution.

- **Event Order Over Absolute Time:** In the realm of distributed systems logical clock synchronization focuses on establishing the order of events than relying on absolute time. Its primary objective is to establish connections between events.
- **Approach towards Understanding Behavior:** Logical clocks serve as storytellers weaving together a narrative of events. This narrative enhances comprehension and facilitates coordination within the distributed system.

3. Mutual exclusion synchronization

In the bustling symphony of distributed systems one major challenge is managing shared resources. Imagine multiple processes competing for access, to the resource simultaneously. To address this issue mutual exclusion synchronization comes into play as an expert technique that reduces chaos and promotes resource harmony. This approach relies on creating a system where different processes take turns accessing shared resources. This helps avoid conflicts and collisions to synchronized swimmers gracefully performing in a water ballet. It ensures that resources are used efficiently and without any conflicts.

- **Managing Resource Conflicts:** In the ecosystem of distributed systems multiple processes often compete for shared resources simultaneously. To address this issue mutual exclusion synchronization enforces a mechanism for accessing resources.
- **Enhancing Efficiency through Sequential Access:** This synchronization approach ensures that resources are accessed sequentially minimizing conflicts and collisions. By orchestrating access, in this manner resource utilization and overall system efficiency are optimized.

Mutual Exclusion

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not 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 Vs. distributed 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. In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not 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.

Requirements of Mutual exclusion Algorithm:

- **No Deadlock:** Two or more site should not endlessly wait for any message that will never arrive.
- **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 site are repeatedly executing critical section
- **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.
- **Fault Tolerance:** In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Some points are need to be taken in consideration to understand mutual exclusion fully :

1) It is an issue/problem which frequently arises when concurrent access to shared resources by several sites is involved. For example, directory management where updates and reads to a directory must be done atomically to ensure correctness.
2) It is a fundamental issue in the design of distributed systems.
3) Mutual exclusion for a single computer is not applicable for the shared resources since it involves resource distribution, transmission delays, and lack of global information.

Solution to distributed mutual exclusion: As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

1. Token Based Algorithm:

- A unique **token** is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence number to order requests for the critical section.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique

2. Non-token based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme

3. Quorum based approach:

- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion

Election Algorithms:

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 other 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 processors 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 send to every active process in the distributed system.

We have two election algorithms for two different configurations of distributed system.

1. **The Bully Algorithm**
2. **The Ring Algorithm**

NOTE:-

- **Distributed algorithms require one process to act as a coordinator or initiator. To decide which process becomes the coordinator different types of algorithms are used.**
- **Election algorithms are meant for electing a coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.**
- **The goal of an election algorithm is to ensure that when an election starts it concludes with all the processes agreeing on who the coordinator should be.**
- **Therefore, whenever initiated, an election algorithm basically finds out which of the currently active processes has the highest priority number and then informs this to all other active processes.**

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-ups answers, it takes over. P's job is done.

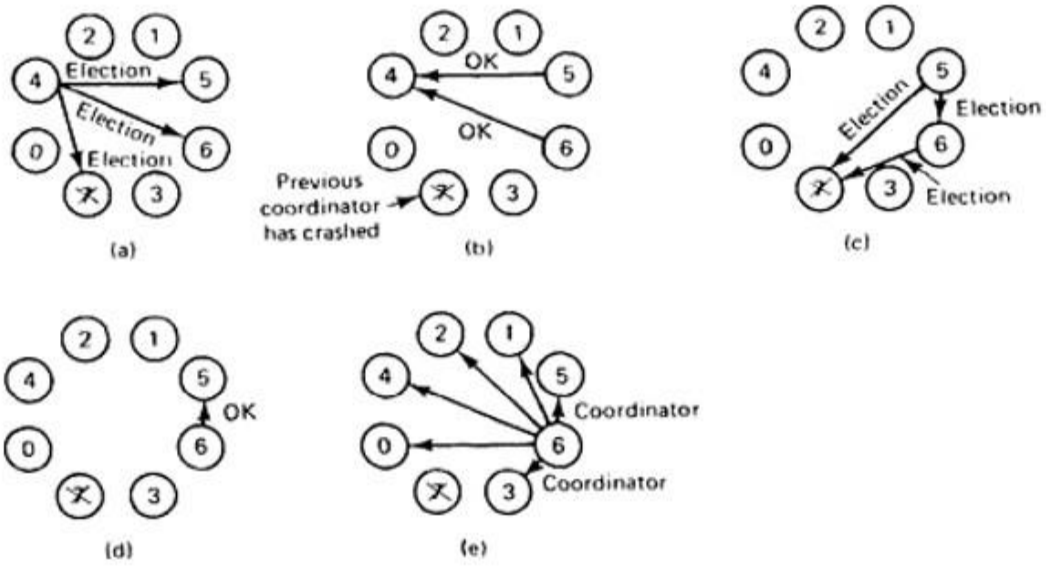
a. A process can get an ELECTION message at any time from one of its lower numbered colleagues.

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:

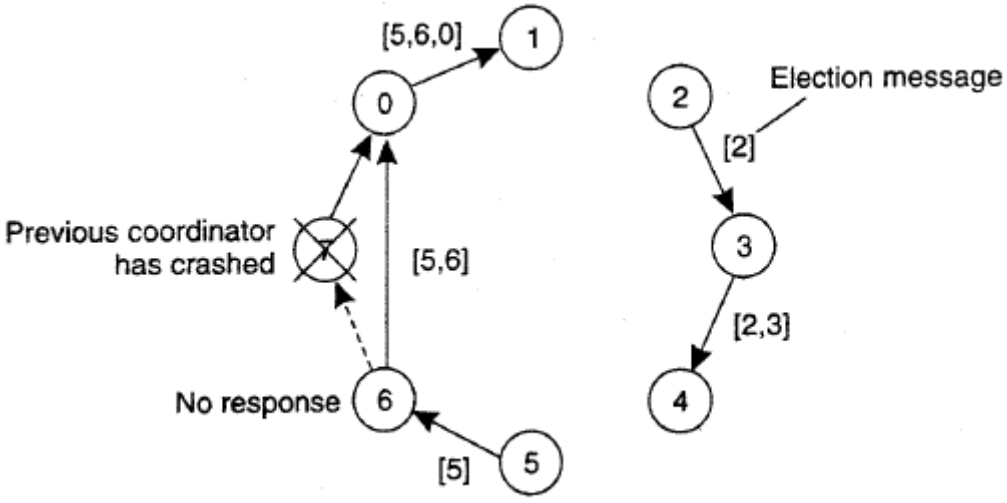


- In fig(a) a group of eight processes taken is numbered from 0 to 7. Assume that previously process 7 was the coordinator, but it has just crashed. Process 4 notices if first and sends ELECTION messages to all the processes higher than it that is 5, 6 and 7.
- In fig (b) processes 5 and 6 both respond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

- In fig(c), both 5 and 6 hold elections by each sending messages to those processes higher than itself.
- In fig(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. It 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.
- If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

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.
  1. 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.
  2. At each step the sender adds its own process number to the list in the message making itself a candidate to elected as coordinator
  3. The message gets back to the process that started it and recognizes this event as the message consists its own process number.
  4. 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. The coordinator is selected with the process having highest number.
  5. When this message is circulated once it is removed and normal work is preceded.



Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, in many cases we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all.

An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency either that both the credit and debit occur or that neither occur. Consistency of data, along with storage and retrieval of data, is a concern often associated with database systems. Recently, there has been an upsurge of interest in using database-systems techniques in operating systems.

System Model

A collection of instructions (or operations) that performs a single logical function is called a transaction. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system. We can think of a transaction as a program unit that accesses and perhaps updates various data items that reside on a disk within some files. From our point of view, such a transaction is simply a sequence of read and write operations terminated by either a commit operation or an abort operation.

A commit operation signifies that the transaction has terminated its execution successfully, whereas an abort operation signifies that the transaction has ended its normal execution due to some logical error or a system failure. If a terminated transaction has completed its execution successfully, it is committed; otherwise, it is aborted. Since an aborted transaction may already have modified the data that it has accessed, the state of these data may not be the same as it would have been if the transaction had executed atomically. So that atomicity is ensured an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing.

We say that such a transaction has been rolled back. It is part of the responsibility of the system to ensure this property. To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

• Volatile storage. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.

• Nonvolatile storage. Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disks and magnetic tapes. Disks are more reliable than main memory but less reliable than magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.

• Stable storage. Information residing in stable storage is never lost (never should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate

information in several nonvolatile storage caches (usually disk) with independent failure modes and to update the information in a controlled manner (Section 12.8). Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accesses. The most widely used method for achieving this form of recording is write-ahead logging. Here, the system maintains, on stable storage, a data structure called the log. Each log record describes a single operation of a transaction write and has the following fields:

• Transaction name. The unique name of the transaction that performed the write operation

• Data item name. The unique name of the data item written

• Old value. The value of the data item prior to the write operation

• New value.

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach: 6.9 Atomic Transactions 225 1. The searching process is time consuming. » 2. Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need to modify. Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of checkpoints. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.

2. Output all modified data residing in volatile storage to the stable storage.

3. Output a log record onto stable storage. The presence of a record in the log allows the system to streamline its recovery procedure. Consider a transaction Tj that committed prior to the checkpoint.

The < T, commit s> record appears in the log before the record, and then finding the subsequent < Ti start > record. Once transaction Tj has been identified, the redo and undo operations need be applied only to transaction Tj and all transactions Tj that started executing after transaction Tj-. We'll call these transactions set T. The remainder of the log can thus be ignored.

The recovery operations that are required are as follows: a For all transactions Tjt in T such that the record < Tj;- commits> appears in the log, execute redo(T)t). • For all transactions Tj- in T that have no < Ti- commits> record in the log, execute undo(TO- 6.9.4 Concurrent Atomic Transactions We have been considering an environment in which only

one transaction can be executing at a time. We now turn to the case where multiple transactions are active simultaneously. Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions are executed serially in some arbitrary order. This property, called serializability, can be maintained by simply executing each transaction within 226 Chapter 6 Process Synchronization a critical section. That is, all transactions share a common semaphore mutex, which is initialized to 1.

When a transaction starts executing, its first action is to execute wa.±t(mutex). After the transaction either commits or aborts, it executes signal(/?z«ta')- Although this scheme ensures the atomicity of all concurrently executing transactions, it is nevertheless too restrictive. As we shall see, in many cases we can allow transactions to overlap their execution while maintaining serializability. A number of different concurrency-control algorithms ensure serializability. These algorithms are described below.

Serializability

Consider a system with two data items, A and B, that are both read and written by two transactions, To and T\. Suppose that these transactions are executed atomically in the order To followed by T\. This execution sequence, which is called a schedule, is represented in Figure 6.22. In schedule 1 of Figure 6.22, the sequence of instruction steps is in chronological order from top to bottom, with instructions of To appearing in the left column and instructions of T\ appearing in the right column. A schedule in which each transaction is executed atomically is called a serial schedule. A serial schedule consists of a sequence of instructions from various transactions wherein the instructions belonging to a particular transaction appear together.

Thus, for a set of n transactions, there exist n\ different valid serial schedules. Each serial schedule is correct, because it is equivalent to the atomic execution of the various participating transactions in some arbitrary order. If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A non-serial schedule does not necessarily imply an incorrect execution (that is, an execution that is not equivalent to one represented by a serial schedule).

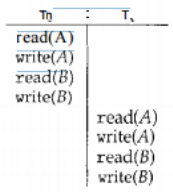


Figure 6.22 Schedule 1: A serial schedule in which T_0 is followed by T_1.

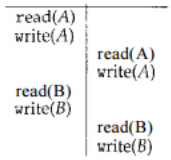


Figure 6.23 Schedule 2: A concurrent serializable schedule.

To see that this is the case, we need to define the notion of conflicting operations. Consider a schedule S in which there are two consecutive operations O,- and Oj of transactions T, and Tj, respectively. We say that O, and Oj conflict if they access the same data item and at least one of them is a write operation. To illustrate the concept of conflicting operations, we consider the nonserial schedule 2 of Figure 6.23. The write(A) operation of To conflicts with the read(A) operation of Ti.

However, the write(A) operation of T\ does not conflict with the read(B) operation of To, because the two operations access different data items. Let Oj and Oj be consecutive operations of a schedule S. If O, and O; are operations of different transactions and O-, and Oj do not conflict, then we can swap the order of O, and 0/ to produce a new schedule S'. We expect S to be equivalent to S', as all operations appear in the same order in both schedules, except for O, and Oj, whose order does not matter. We can illustrate the swapping idea by considering again schedule 2 of Figure 6.23.

As the write(A) operation of T\ does not conflict with the read(B) operation of To, we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping nonconflicting operations, we get:

• Swap the read(B) operation of TQ with the read(A) operation of T\.

• Swap the write(B) operation of To with the write(A) operation of T\.

• Swap the write(B) operation of To with the read(A) operation of T\. The final result of these swaps is schedule 1 in Figure 6.22, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule. If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is conflict serializable. Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1

Locking Protocol

One way to ensure serializability is to associate with each data item a lock and to require that each transaction follow a locking protocol that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes: 228 Chapter 6 Process Synchronization

• Shared. If a transaction X, has obtained a shared-mode lock (denoted by S) on data item Q, then 7] can read this item but cannot write Q

• Exclusive. If a transaction T, has obtained an exclusive-mode lock (denoted by X) on data item Q, then 7} can both read and write Q. We require that every transaction request a lock in an appropriate mode on data item Q, depending on the type of operations it will perform on Q. To access data item Q, transaction 7} must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and T; can now access it. However, if the data item Q is currently locked by some other transaction, then 7) may have to wait. More specifically, suppose that 7} requests an exclusive lock on Q. In this case, 7] must wait until the lock on Q is released. If T, requests a shared lock on Q, then 7) must wait if Q is locked in exclusive mode.

Otherwise, it can obtain the lock and access Q. Notice that this scheme is quite similar to the readers-writers algorithm discussed in Section 6.6.2. A transaction may unlock a data item that it locked at an earlier point. It must, however, hold

a lock on a data item as long as it accesses that item. Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured.

One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

• Growing phase. A transaction may obtain locks but may not release any lock.

• Shrinking phase. A transaction may release locks but may not obtain any new locks. Initially, a transaction is in the growing phase. The transaction acquires locks as needed.

Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued. The two-phase locking protocol ensures conflict serializability (Exercise 6.25). It does not, however, ensure freedom from deadlock. In addition, it is possible that, for a given set of transactions, there are conflict-serializable schedules that cannot be obtained by use of the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

Timestamp-Based Protocols

In the locking protocols described above, the order followed by pairs of conflicting transactions is determined at execution time by the first lock that both request and that involves incompatible modes. Another method for determining the serializability order is to select an order in advance. The most common method for doing so is to use a timestamp ordering scheme. With each transaction T, in the system, we associate a unique fixed timestamp, denoted by TS(T/). This timestamp is assigned by the system 6.9 Atomic Transactions 229 before the transaction T, starts execution. If a transaction 7} has been assigned timestamp TS(Tj-), and later a new transaction 7) enters the system, then TS(7}) < TS(Tj). There are two simple methods for implementing this scheme:

• Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.

• Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned. The timestamps of the transactions determine the serializability order. Thus, if TS(T,) < TS(T,), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T, appears before transaction T,. To implement this scheme, we associate with each data item Q two timestamp values:

• W-timestamp(Q) denotes the largest timestamp of any transaction that successfully executed write(Q).

• R-timestamp(Q) denotes the largest timestamp of any transaction that successfully executed read(Q). These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed. The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

• Suppose that transaction T,- issues read(Q): o If TS(T,) < W-timestamp(), then T, needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Tj- is rolled back. o If TS(TJ) > W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T,).

• Suppose that transaction 7} issues write(Q): o If TS(T,) < R-timestamp(Q), then the value of Q that 7} is producing was needed previously and T,- assumed that this value would never be produced. Hence, the write operation is rejected, and 7} is rolled back. => If TS(T,) < W-timestamp(Q), then T, is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and T, is rolled back. o Otherwise, the write operation is executed. A transaction T, that is rolled back as a result of the issuing of either a read or write operation is assigned a new timestamp and is restarted.

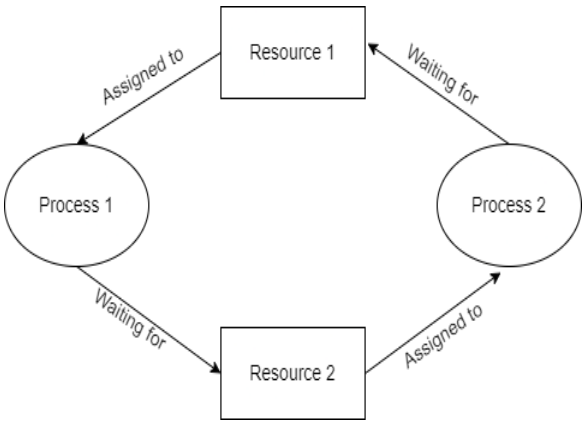
T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Figure 6.24 Schedule 3: A schedule possible under the timestamp protocol.

To illustrate this protocol, consider schedule 3 of Figure 6.24, which includes transactions % and T3. We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, TS(T2) < TS(T3), and the schedule is possible under the timestamp protocol. This execution can also be produced by the two-phase locking protocol. However, some schedules are possible under the two-phase locking protocol but not under the timestamp protocol, and vice versa. The timestamp protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol also ensures freedom from deadlock, because no transaction ever waits.

Deadlock in Distributed System

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. When this situation arises, it is known as Deadlock.



Deadlock

A Distributed System is a Network of Machines that can exchange information with each other through Message-passing. It can be very useful as it helps in resource sharing. In such an 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:

- **Avoidance:** Resources are carefully allocated to avoid deadlocks.
- **Prevention:** Constraints are imposed on the ways in which processes request resources in order to prevent deadlocks.
- **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.

Types of Distributed Deadlock:

There are two types of Deadlocks in Distributed System:

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.

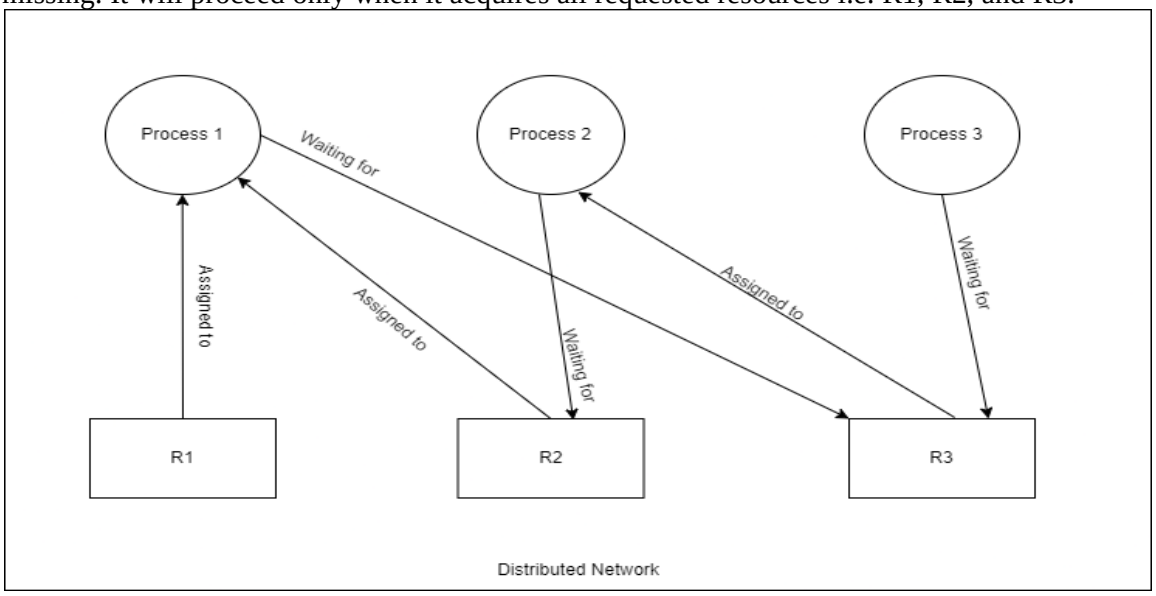


figure 1: Resource Deadlock

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 modeled by using 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.

- 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.

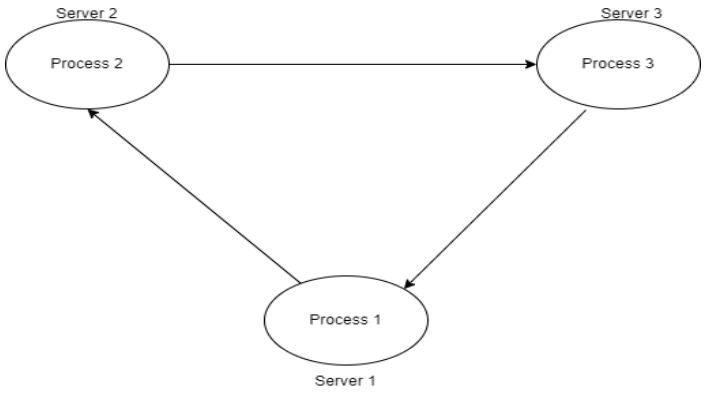


figure 2: Communication Deadlock

Deadlock Prevention

A Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for a resource that is held by some other process.

There are four necessary conditions for a Deadlock to ha**ppen which are:**

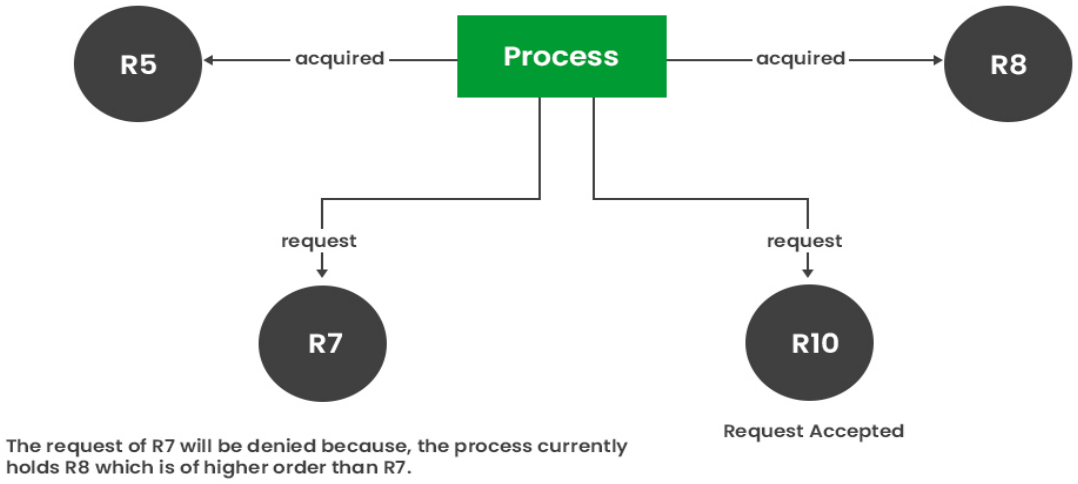
- **Mutual Exclusion:** There is at least one resource that is non-sharable and can be used by only one Process at a time.
- **Hold and Wait:** A process is holding at least one Resource and waiting for another.
- **No Preemption:** A resource cannot be taken from a process until it releases the resource.
- **Circular Wait:** At least two processes should form a circular chain by holding a resource and waiting for a resource that is held by the next process in the chain.

So the above four conditions are necessary for a deadlock to occur, if any one of the above four conditions is prevented, we can prevent a Deadlock to occur. There are 2 ways to prevent deadlock in a distributed system.

- Ordered Request
- Collective Request

Ordered Request

As the name suggests, in this Deadlock Prevention method, each resource type is assigned a certain level to maintain a resource request policy for a process. This is known as the Resource Allocation policy. For each Resource, a global level number is assigned to impose ordering of all resource types. While requesting for a resource, a Process has to make sure that it does not request for a resource whose level order is lower than the highest-level order resource it currently holds. It can only request resources higher than the highest level resources, held by the process. Refer to the below example for a much better understanding. Suppose there are 10 resources from level 1 to 10, and 10 is the highest level order resource. If a Process currently has resources 5 and 8, it cannot request a resource below 8, it can only request resources 9 and 10. Like, the process cannot make a request for Resource 7, while holding resource 8. This method does not mean that requests should be made in increasing order of sequence. Before sending a request for resource 7, it has to release the held resource 8. After releasing 8, it can acquire 7. It is allowed because currently, it does not hold a resource higher than 7.



The Method makes sure that the Circular Wait condition is not reached and if one of the deadlock conditions is denied, the deadlock will be prevented.

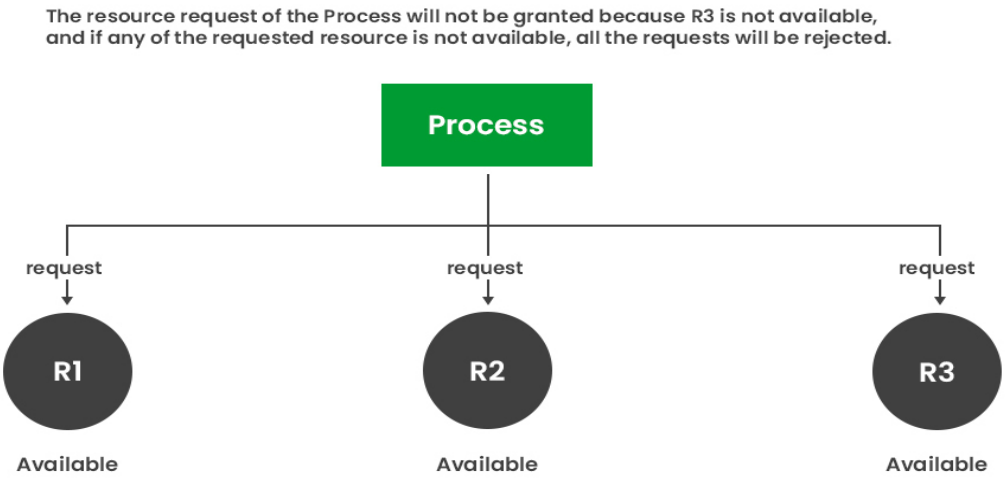
Disadvantages:

- A Process that has resource request orders in increasing order levels with respect to Resource Allocation Policy, will utilize all the resources and will waste the resources.
- For Example: Take the reference of the above example, if a process has a resource request order from 1 to 10. It will acquire all resources and this degrades resource utilization.

Collective Request

This method prevents the Hold and Wait for condition by using any of the following Resource Allocation Policies :

- This Resource Allocation Policy ensures that a Process requests for all the required resources before the execution of the process. If any of the required resources are not available, the request is not granted. Example: A Process requires 3 resources for its execution, if all the 3 resources are available, the request is granted and all 3 resources are allocated. But if any one of the 3 resources is not available, then none of the 3 resources will be allocated and the request is rejected.



- In this Resource Allocation Policy, the Processes have to ensure that before requesting any resource, it should not hold any resource. That is, it should release all of its current resources before making any request for new resources. While requesting any resource, the process should not hold any resources.

- In both of the above Resource Allocation Policies, the Hold and Wait condition of Deadlock can't be reached and thus the Deadlock is prevented.

Disadvantages:

- It leads to low resource utilization.
- Process Starvation is also possible because, if there is a process with high resource needs, its request acceptance will get delayed.

Deadlock Detection in Distributed Systems

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.

Approaches to detect deadlock in the distributed system

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.

Deadlock Handling Strategies

Various deadlock handling strategies in the distributed system are as follows:

1. There are mainly three approaches to handling deadlocks: deadlock prevention, deadlock avoidance, and deadlock detection.
2. Handling deadlock becomes more complex in distributed systems since no site has complete knowledge of the system's present state and every inter-site communication entails a limited and unpredictable latency.
3. The operating system uses the deadlock Avoidance method to determine whether the system is in a safe or unsafe state. The process must inform the operating system of the maximum number of resources, and a process may request to complete its execution.
4. Deadlocks prevention are commonly accomplished by implementing a process to acquire all of the essential resources at the same time before starting execution or by preempting a process that already has the resource.
5. In distributed systems, this method is highly inefficient and impractical.
6. The presence of cyclical wait needs an examination of the status of process resource interactions to detect deadlock.
7. The best way to dealing with deadlocks in distributed systems appears to be deadlock detection.

Issues of Deadlock Detection

Various issues of deadlock detection in the distributed system are as follows:

1. Deadlock detection-based deadlock handling requires addressing two fundamental issues: first, detecting existing deadlocks, and second, resolving detected deadlocks.
2. Detecting deadlocks entails tackling two issues: WFG maintenance and searching the WFG for the presence of cycles.
3. In a distributed system, a cycle may include multiple sites. The search for cycles is highly dependent on the system's WFG as represented across the system.

Resolution of Deadlock Detection

Various resolutions of deadlock detection in the distributed system are as follows:

1. Deadlock resolution includes the braking existing wait-for dependencies in the system WFG.
2. It includes rolling multiple deadlocked processes and giving their resources to the blocked processes in the deadlock so that they may resume execution.

Deadlock detection algorithms in Distributed System

Various deadlock detection algorithms in the distributed system are as follows:

1. **Path-Pushing Algorithms**
2. **Edge-chasing Algorithms**
3. **Diffusing Computations Based Algorithms**
4. **Global State Detection Based Algorithms**

Path-Pushing Algorithms

Path-pushing algorithms detect distributed deadlocks by keeping an explicit global WFG. The main concept is to create a global WFG for each distributed system site. When a site in this class of algorithms performs a deadlock computation, it sends its local WFG to all neighboring sites. The term path-pushing algorithm was led to feature the sending around the paths of global WFG.

Edge-Chasing Algorithms

An edge-chasing method verifies a cycle in a distributed graph structure by sending special messages called probes along the graph's edges. These probing messages are distinct from request and response messages. If a site receives the matching probe that it previously transmitted, it can cancel the formation of the cycle.

Diffusing Computations Based Algorithms

In this algorithm, deadlock detection computation is diffused over the system's WFG. These techniques use echo algorithms to detect deadlocks, and the underlying distributed computation is superimposed on this computation. If this computation fails, the initiator reports a deadlock global state detection.

Global State Detection Based Algorithms

Deadlock detection algorithms based on global state detection take advantage of the following facts:

1. A consistent snapshot of a distributed system may be taken without freezing the underlying computation.
2. If a stable property exists in the system before the snapshot collection starts, it will be preserved.

UNIT-3

Processes and Processors in Distributed Systems

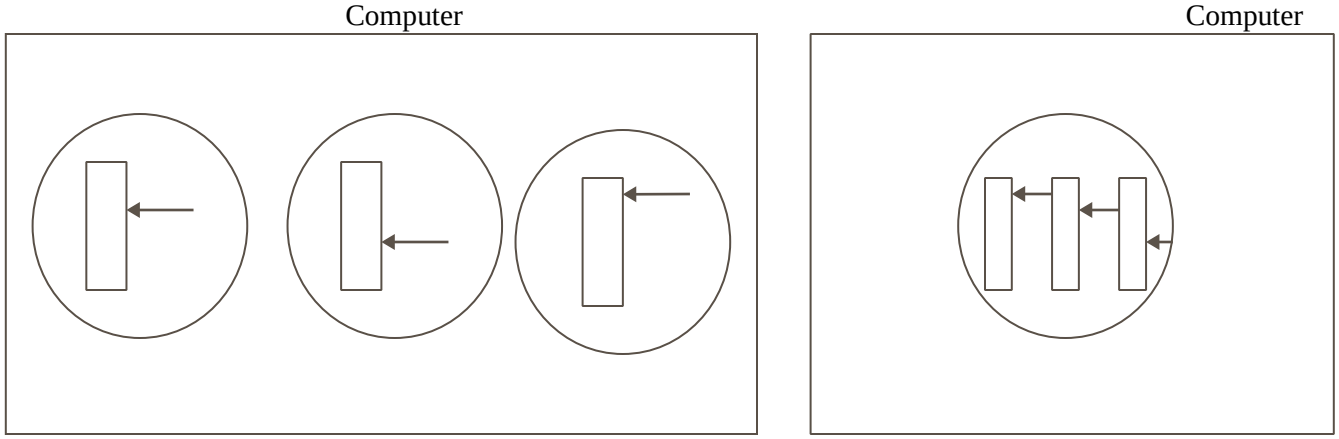
- In most traditional OS, each process has an address space and a single thread of control.
- It is desirable to have multiple threads of control sharing one address space but running in quasi-parallel.

Introduction to threads

- Thread is a lightweighted process.
- The analogy: thread is to process as process is to machine.
- Each thread runs strictly sequentially and has its own program counter and stack to keep track of where it is.
- Threads share the CPU just as processes do: first one thread runs, then another does.
- Threads can create child threads and can block waiting for system calls to complete.
- All threads have exactly the same address space. They share code section, data section, and OS resources (open files & signals). They share the same global variables. One thread can read, write, or even completely wipe out another thread's stack.
- Threads can be in any one of several states:
  running, blocked, ready, or terminated.
- There is no protection between threads:

(1)it is not necessary (2) it should not be necessary: a process is always owned by a single user, who has created multiple threads so that they can cooperate, not fight.

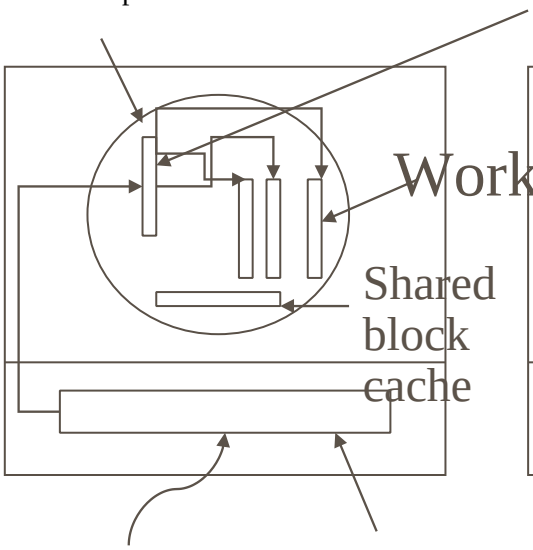
Threads



Thread usage

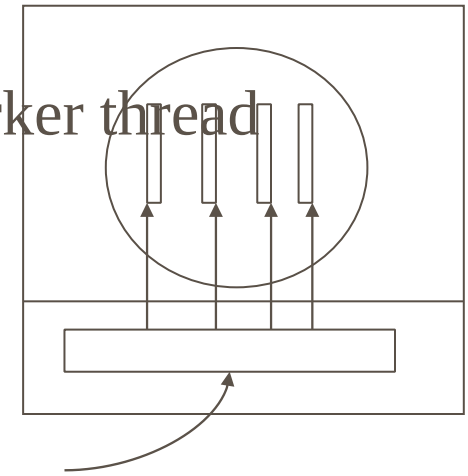
Dispatcher/worker model

File server process



Request for Mailbox

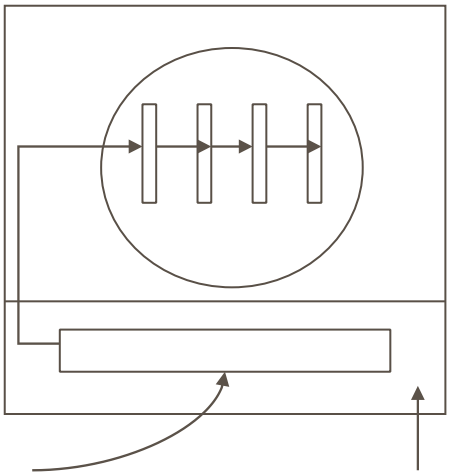
Dispatcher thread



Team model

Kernel work arrives

Pipeline mo



Advantages of using threads

1. Useful for clients: if a client wants a file to be replicated on multiple servers, it can have one thread talk to each server.
2. Handle signals, such as interrupts from the keyboard. Instead of letting the signal interrupt the process, one thread is dedicated full time to waiting for signals.
3. Producer-consumer problems are easier to implement using threads because threads can share a common buffer.
4. It is possible for threads in a single address space to run in parallel, on different CPUs.

Design Issues for Threads Packages

- A set of primitives (e.g. library calls) available to the user relating to threads is called a **thread package**.
- **Static thread**: the choice of how many threads there will be is made when the program is written or when it is compiled. Each thread is allocated a fixed stack. This approach is simple, but inflexible.
- **Dynamic thread**: allow threads to be created and destroyed on-the-fly during execution.

Mutex

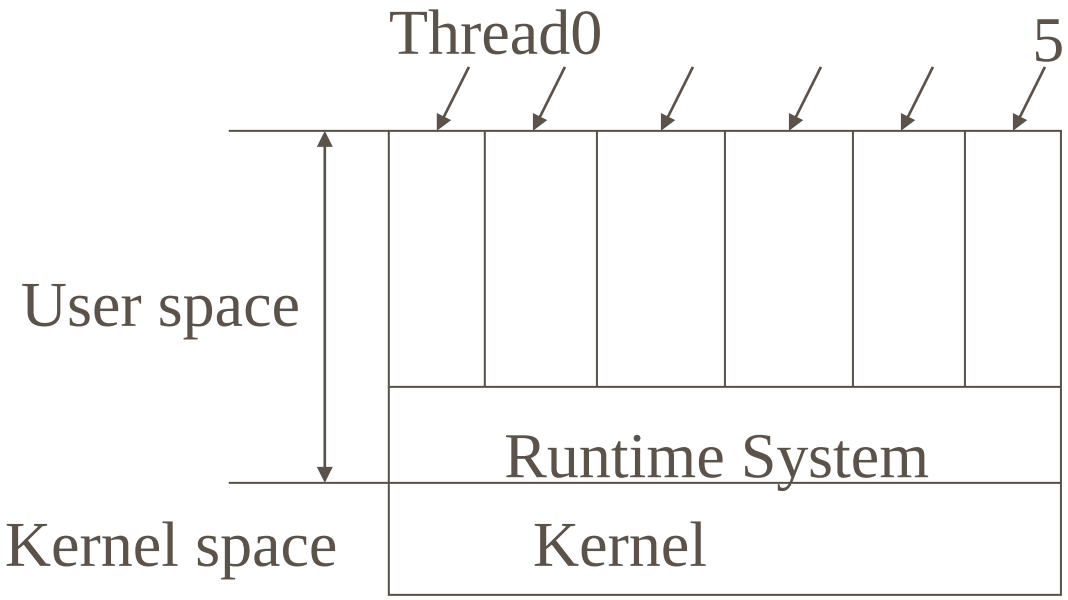
- If multiple threads want to access the shared buffer, a mutex is used. A mutex can be locked or unlocked.
- Mutexes are like binary semaphores: 0 or 1.

- Lock: if a mutex is already locked, the thread will be blocked.
- Unlock: unlocks a mutex. If one or more threads are waiting on the mutex, exactly one of them is released. The rest continue to wait.
- Trylock: if the mutex is locked, Trylock does not block the thread.

  Instead, it returns a status code indicating failure.

Implementing a threads package

■ **Implementing threads in user space**



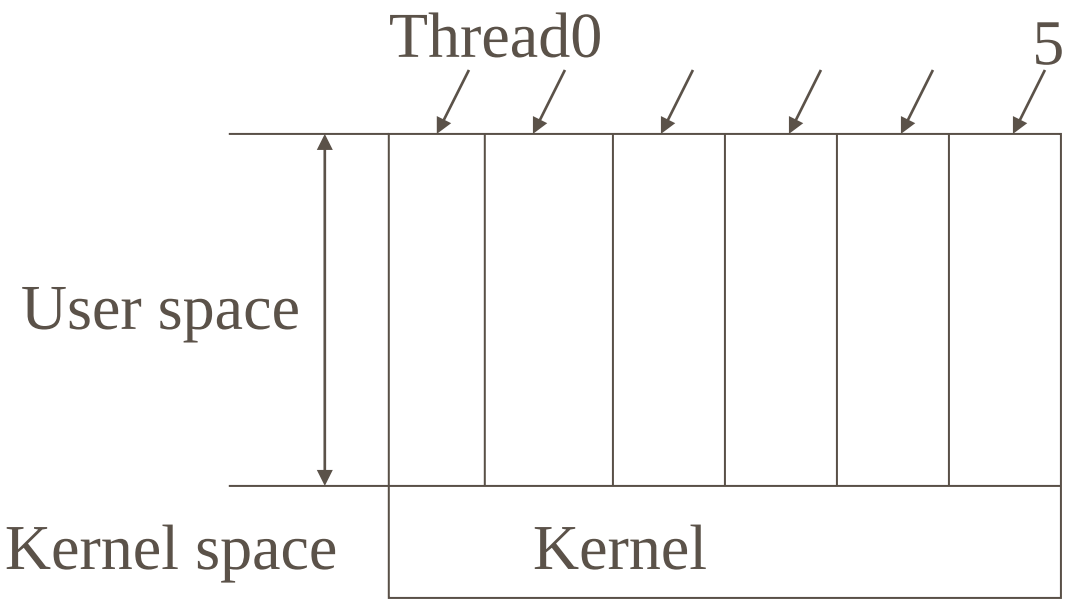
Advantage

- User-level threads package can be implemented on an operating system that does not support threads. For example, the UNIX system.
- The threads run on top of a runtime system, which is a collection of procedures that manage threads. The runtime system does the thread switch. Store the old environment and load the new one. It is much faster than trapping to the kernel.
- User-level threads scale well. Kernel threads require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Disadvantage

- Blocking system calls are difficult to implement. Letting one thread make a system call that will block the thread will stop all the threads.
- Page faults. If a thread causes a page faults, the kernel does not know about the threads. It will block the entire process until the page has been fetched, even though other threads might be runnable.
- If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- For the applications that are essentially CPU bound and rarely block, there is no point of using threads. Because threads are most useful if one thread is blocked, then another thread can be used.

Implementing threads in the kernel



- The kernel knows about and manages the threads. No runtime system is needed. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation and destruction.
- To manage all the threads, the kernel has one table per process with one entry per thread.
- When a thread blocks, the kernel can run either another thread from the same process or a thread from a different process.

Scheduler Activations

- Scheduler activations combine the advantage of user threads (good performance) and kernel threads.
- The goals of the scheduler activation are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space.
- Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks, the user-space runtime system can schedule a new one by itself.
- Disadvantage:

Upcall from the kernel to the runtime system violates the structure in the layered system.

System Models

- **The workstation model:**

the system consists of workstations scattered throughout a building or campus and connected by a high-speed LAN.

The systems in which workstations have local disks are called **diskful workstations**. Otherwise, **diskless workstations**.

Why diskless workstation?

- If the workstations are diskless, the file system must be implemented by one or more remote file servers. Diskless workstations are cheaper.

- Ease of installing new release of program on several servers than on hundreds of machines. Backup and hardware maintenance is also simpler.
- Diskless does not have fans and noises.
- Diskless provides symmetry and flexibility. You can use any machine and access your files because all the files are in the server.
- Advantage: low cost, easy hardware and software maintenance, symmetry and flexibility.
- Disadvantage: heavy network usage; file servers may become bottlenecks.

Diskful workstations

- The disks in the diskful workstation are used in one of the four ways:
- 1. Paging and temporary files (temporary files generated by the compiler passes).
- ■

Advantage: reduces network load over diskless case

Disadvantage: higher cost due to large number of disks needed

- ■
- 2. Paging, temporary files, and system binaries (binary executable programs such as the compilers, text editors, and electronic mail handlers).
- ■

Advantage: reduces network load even more

Disadvantage: higher cost; additional complexity of updating the binaries

- 3.Paging, temporary files, system binaries, and file caching (download the file from the server and cache it in the local disk. Can make modifications and write back. Problem is cache coherence).

Advantage: still lower network load; reduces load on file servers as well Disadvantage: higher cost; cache consistency problems

- ■ 4.Complete local file system (low network traffic but sharing is difficult).

Advantage: hardly any network load; eliminates need for file servers Disadvantage: loss of transparency

Using Idle Workstation

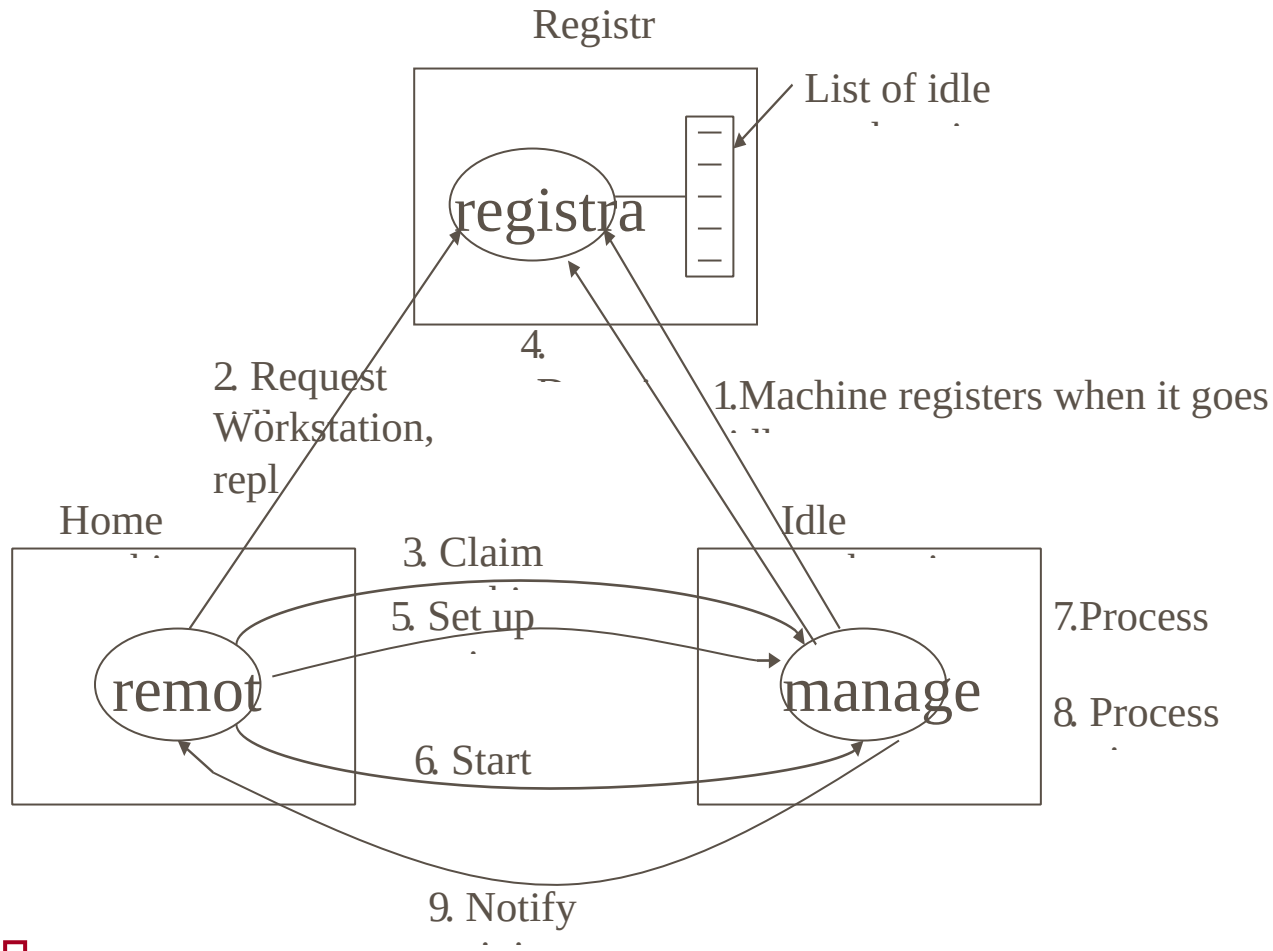
- The earliest attempt to use idle workstations is command:
- rsh machine command
- The first argument names a machine and the second names a command to run.

Flaws

- 1) User has to tell which machine to use.

- 2) The program executes in a different environment than the local one.
- 3) Maybe log in to an idle machine with many processes.

What is an idle workstation?



- If no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle.
- The algorithms used to locate idle workstations can be divided into two categories:
- **server driven**--if a server is idle, it registers in registry file or broadcasts to every machine.
- **client driven** --the client broadcasts a request asking for the specific machine that it needs and wait for the reply.

A registry-based algorithm for finding & using idle workstation

How to run the process remotely?

- To start with, it needs the same view of the file system, the same working directory, and the same environment variables.
- Some system calls can be done remotely but some can not. For example, read from keyboard and write to the screen. Some must be done remotely, such as the UNIX system calls SBRK (adjust the size of the data segment), NICE (set CPU scheduling priority), and PROFIL (enable profiling of the program counter).

The processor pool model

- A processor pool is **a rack full of CPUs in the machine room**, which can be dynamically allocated to users on demand.
- Why processor pool?
- Input rate v, process rate u. mean response time T=1/(u-v).
- If there are n processors, each with input rate v and process rate u.
- If we put them together, input rate will be nv and process rate will be nu. Mean response time will be T=1/(nu-nv)=(1/n)T.

A hybrid model

- A possible compromise is to provide each user with a personal workstation and to have a processor pool in addition.
- For the hybrid model, even if you can not get any processor from the processor pool, at least you have the workstation to do the work.

Processor Allocation

- determine which process is assigned to which processor. Also called load distribution.
- Two categories:
- Static load distribution-nonmigratory, once allocated, can not move, no matter how overloaded the machine is.
- Dynamic load distribution-migratory, can move even if the execution started. But algorithm is complex.

The goals of allocation

- 1 Maximize CPU utilization
- 2 Minimize mean response time/
  Minimize response ratio

  Response ratio-the amount of time it takes to run a process on some machine, divided by how long it would take on some unloaded benchmark processor. E.g. a 1sec job that takes 5 sec. The ratio is 5/1.

Design issues for processor allocation algorithms

- Deterministic versus heuristic algorithms
- Centralized versus distributed algorithms
- Optimal versus suboptimal algorithms

- Local versus global algorithms
- Sender-initiated versus receiver-initiated algorithms

How to measure a processor is overloaded or underloaded?

1 Count the processes in the machine? Not accurate because even the machine is idle there are some daemons running.

2 Count only the running or ready to run processes? Not accurate because some daemons just wake up and check to see if there is anything to run, after that, they go to sleep. That puts a small load on the system.

3 Check the fraction of time the CPU is busy using time interrupts. Not accurate because when CPU is busy it sometimes disable interrupts.

How to deal with overhead?

A proper algorithm should take into account the CPU time, memory usage, and network bandwidth consumed by the processor allocation algorithm itself.

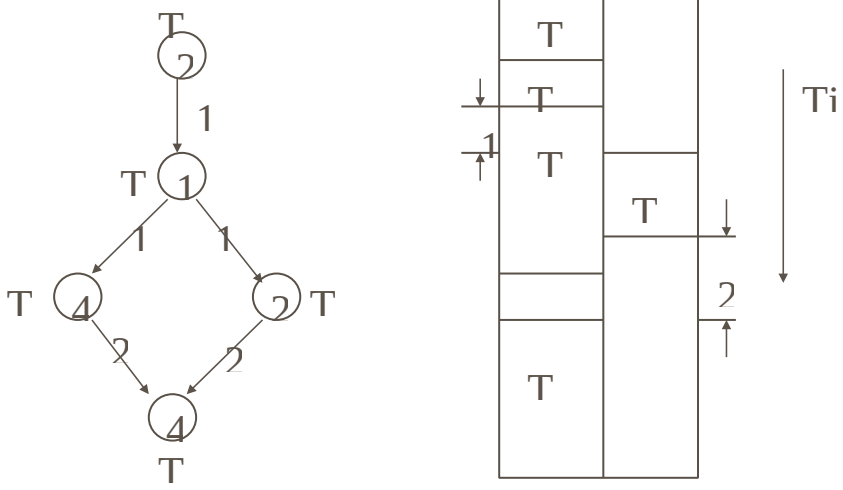
How to calculate complexity?

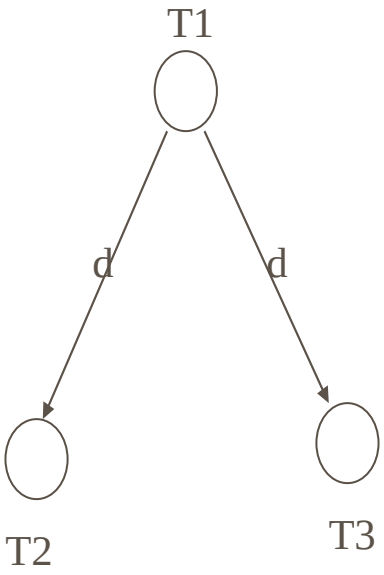
If an algorithm performs a little better than others but requires much complex implementation, better use the simple one.

How to deal with stability?

Problems can arise if the state of the machine is not stable yet, still in the process of updating.

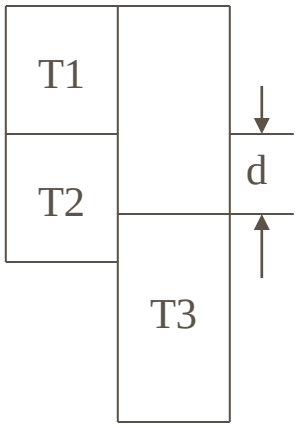
Load distribution based on precedence graph





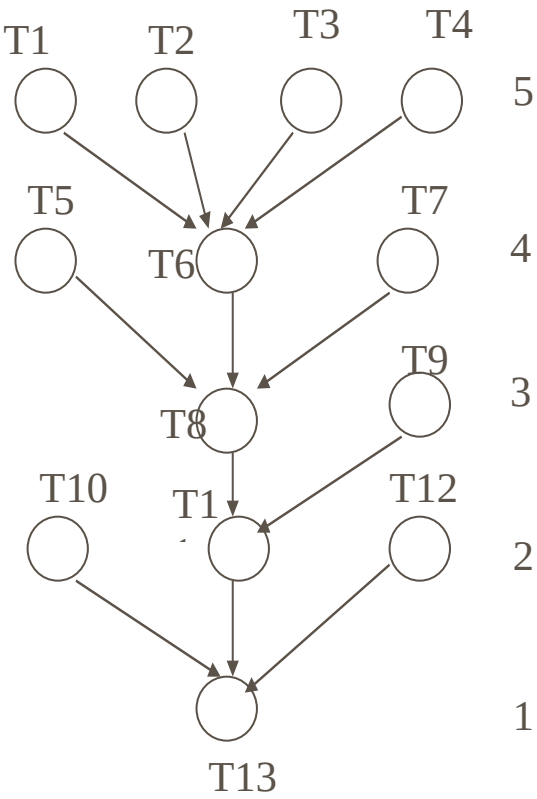
T1	T1
T2	T3

T1
T2
T3



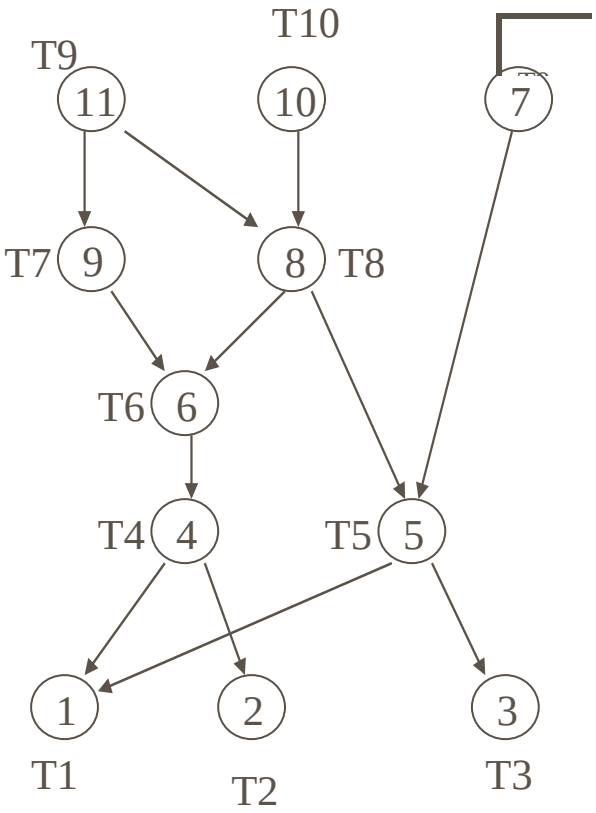
Two Optimal Scheduling Algorithms

- The precedence graph is a tree



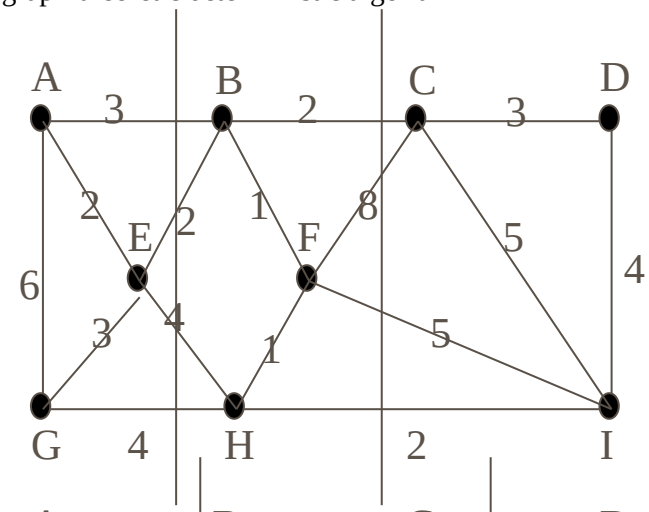
T1	T2	T3
T4	T5	T7
T6	T9	T10
T8	T12	
T11		
T13		

- There are only two processors

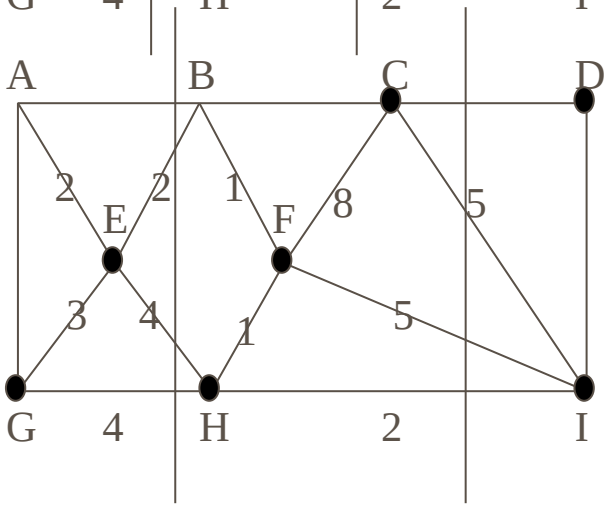


T9	T10
T7	T8
T11	T6
T5	T4
T3	T2
T1	

A graph-theoretic deterministic algorithm



Total network traffic: 2+4+3+4+2+8+5+2 =30



Total network traffic: 3+2+4+4+3+5+5+2 =28

Dynamic Load Distribution

■ **Components of dynamic load distribution**

- Initiation policy
- Transfer policy
- Selection policy
- Profitability policy
- Location policy
- Information policy

Dynamic load distribution algorithms

■ Load balancing algorithms can be classified as follows:

- Global vs. Local
- Centralized vs. decentralized
- Noncooperative vs. cooperative
- Adaptive vs. nonadaptive

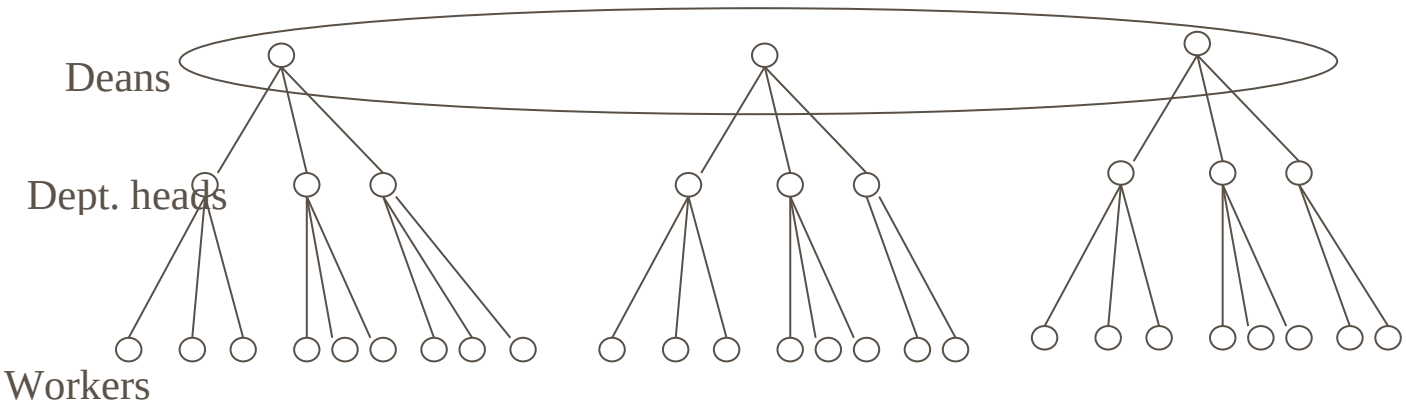
A centralized algorithm

■ **Up-down algorithm**: a coordinator maintains a **usage table** with one entry per personal workstation.

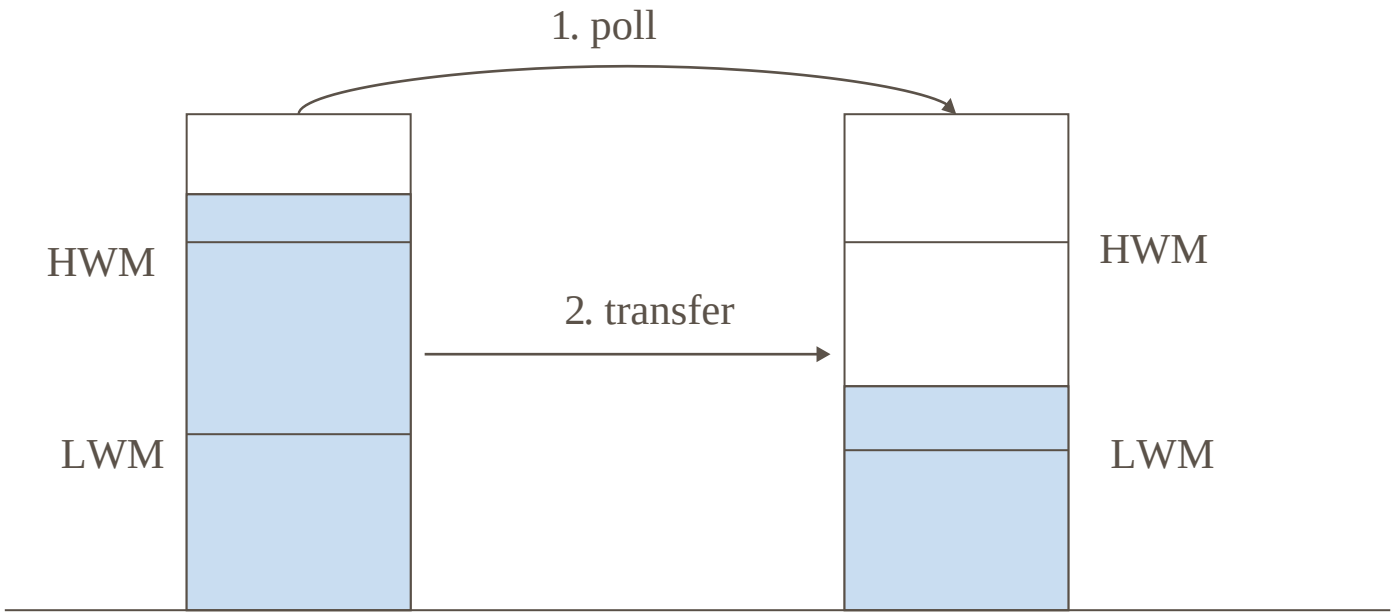
1. When a workstation owner is running processes on other people's machines, it accumulates penalty points, a fixed number per second. These points are added to its usage table entry.
2. When it has unsatisfied requests pending, penalty points are subtracted from its usage table entry.

- A positive score indicates that the workstation is a net user of system resources, whereas a negative score means that it needs resources. A zero score is neutral.
- When a processor becomes free, the pending request whose owner has the lowest score wins.

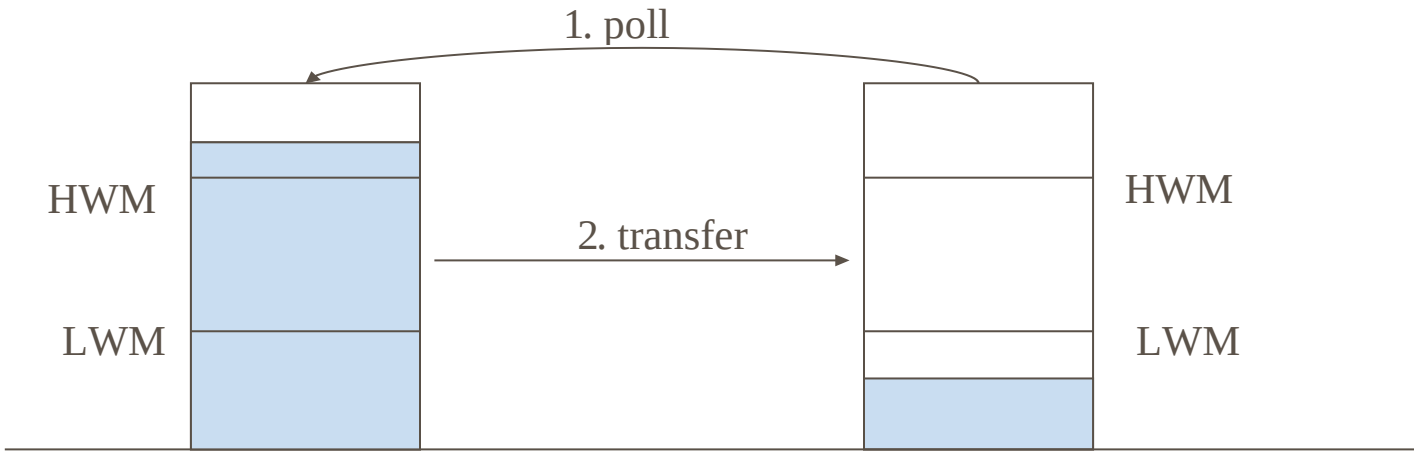
A hierarchical algorithm



A sender-initiated algorithm



A receiver-initiated algorithm

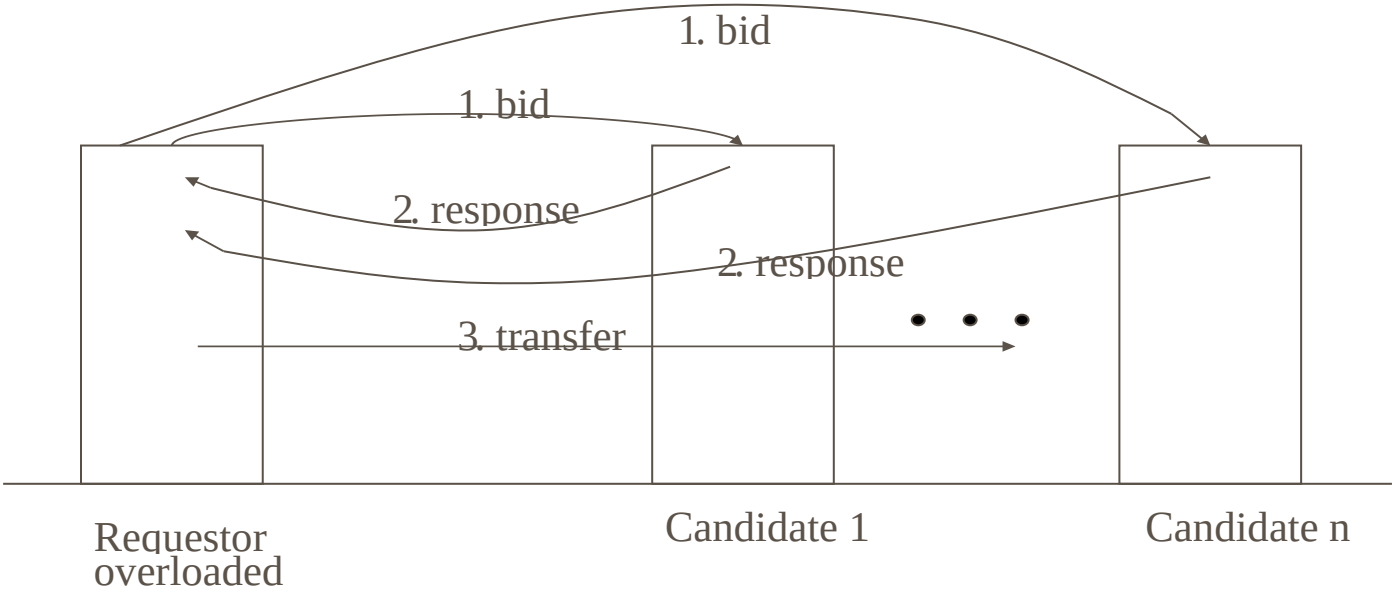


A bidding algorithm

■ This acts like an economy. Processes want CPU time. Processors give the price. Processes pick up the process that can do the work and at a reasonable price and processors pick up the process that gives the highest price.

Bidding algorithm

- *Iterative* (also called *nearest neighbor*) *algorithm*: rely on successive approximation through load exchanging among neighboring nodes to reach a global load distribution.
- *Direct algorithm:* determine senders and receivers first and then load exchanges follow.



Direct algorithm

- the average system load is determined first. Then it is broadcast to all the nodes in the system and each node determines its status: overloaded or underloaded. We can call an overloaded node a *peg* and an underloaded node a *hole*.
- the next step is to fill holes with pegs preferably with minimum data movements.

Nearest neighbor algorithms: diffusion

$L_u(t+1)=L_u(t)+\sum_{v \in A(u)} (\alpha_{u,v} (L_v(t)-L_u(t))+\beta_u(t))$ A(u) is the neighbor set of u.

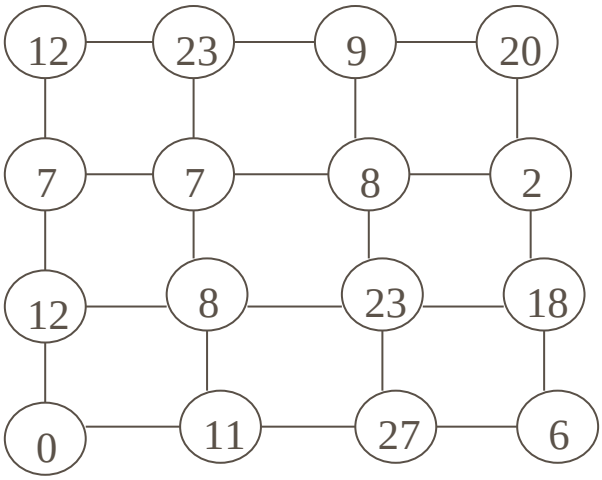
$0<=\alpha_{u,v}<=1$ is the diffusion parameter which determines the amount of load exchanged between two neighboring nodes u and v.

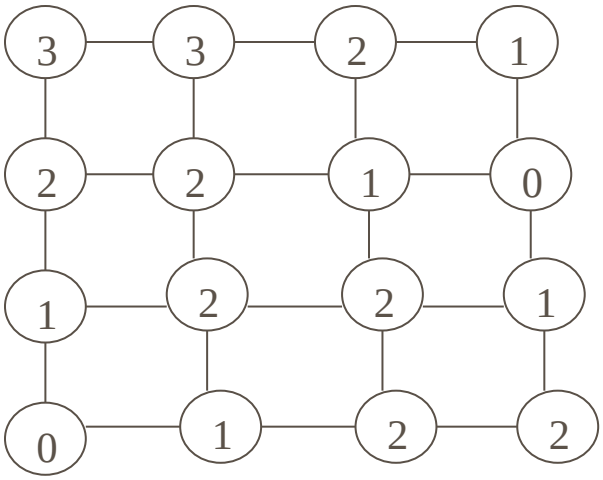
$\beta_u(t))$ is the new incoming load between t and t+1.

Nearest neighbor algorithm: gradient

- One of the major issues is to define a reasonable contour of gradients. The following is one model. The *propagated pressure* of a processor u, p(u), is defined asIf u is lightly loaded, p(u) = 0
- Otherwise, $p(u) = 1 + \min\{p(v)|v \in A(u)\}$

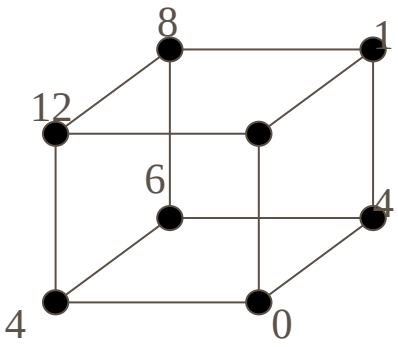
Nearest neighbor algorithm: gradient

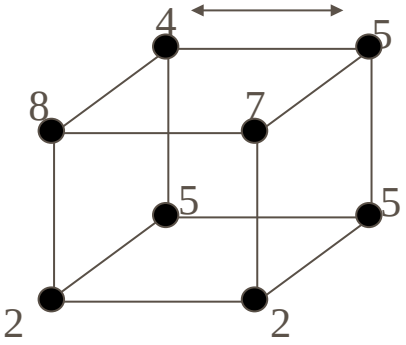


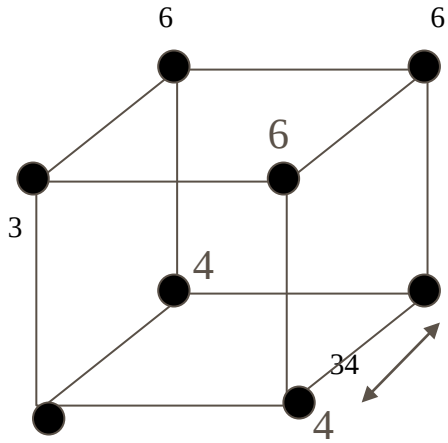


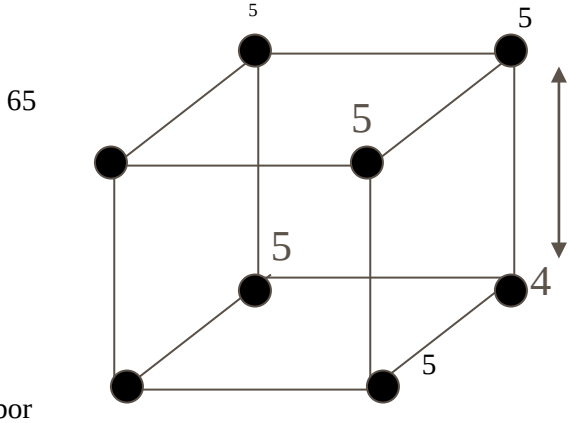
A node is lightly loaded if its load is < 3.

Nearest neighbor algorithm: dimension exchange

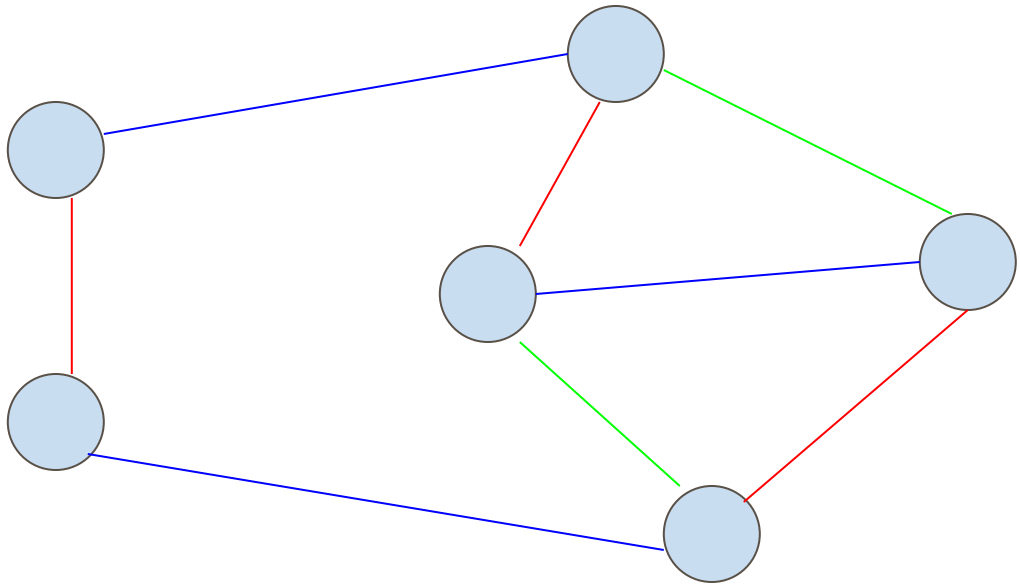








Nearest neighbor algorithm: dimension exchange extension



Fault tolerance

- **component faults**
- Transient faults: occur once and then disappear. E.g. a bird flying through the beam of a microwave transmitter may cause lost bits on some network. If retry, may work.
- Intermittent faults: occurs, then vanishes, then reappears, and so on. E.g. A loose contact on a connector.
- Permanent faults: continue to exist until the fault is repaired. E.g. burnt-out chips, software bugs, and disk head crashes.

System failures

- There are two types of processor faults:
- 1 Fail-silent faults: a faulty processor just stops and does not respond
- 2 Byzantine faults: continue to run but give wrong answers

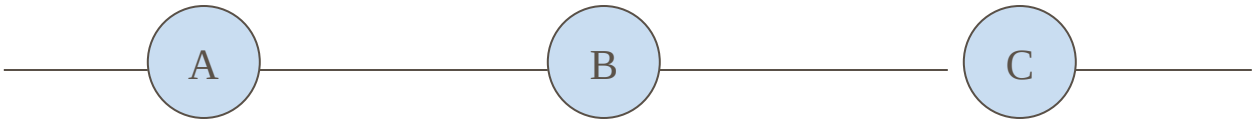
Synchronous versus Asynchronous systems

■ Synchronous systems: a system that has the property of always responding to a message within a known finite bound if it is working is said to be **synchronous**.
Otherwise, it is **asynchronous**.

Use of redundancy

- There are three kinds of fault tolerance approaches:
- 1 Information redundancy: extra bit to recover from garbled bits.
- 2 Time redundancy: do again
- 3 Physical redundancy: add extra components. There are two ways to organize extra physical equipment: **active replication** (use the components at the same time) and **primary backup** (use the backup if one fails).

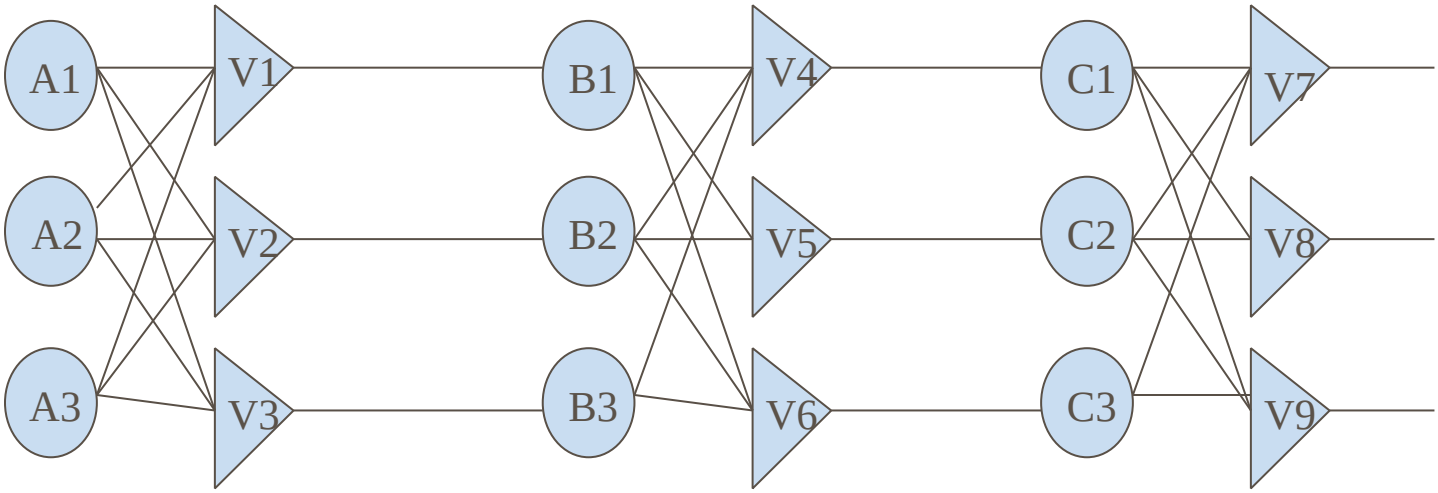
Fault Tolerance using active replication



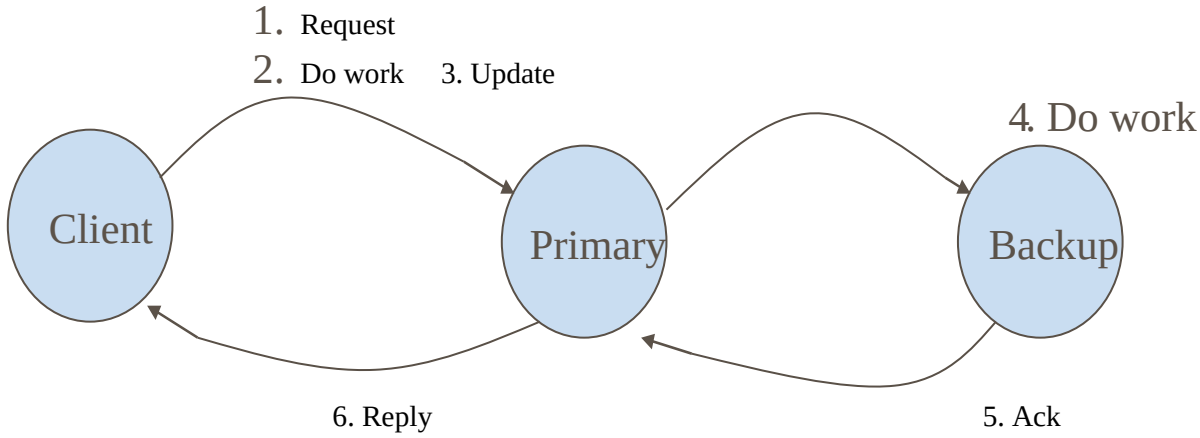
voter

How much replication is needed?

- A system is said to be **k fault tolerant** if it can survive faults in *k* components and still meet its specifications.
- *K+1* processors can fault tolerant *k* fail-stop faults. If *k* of them fail, the one left can work. But need *2k+1* to tolerate *k* Byzantine faults because if *k* processors send out wrong replies, but there are still *k+1* processors giving the correct answer. By majority vote, a correct answer can still be obtained.



Fault Tolerance using primary backup



Handling of Processor Faults

- **Backward recovery** – checkpoints.
- In the checkpointing method, two undesirable situations can occur:

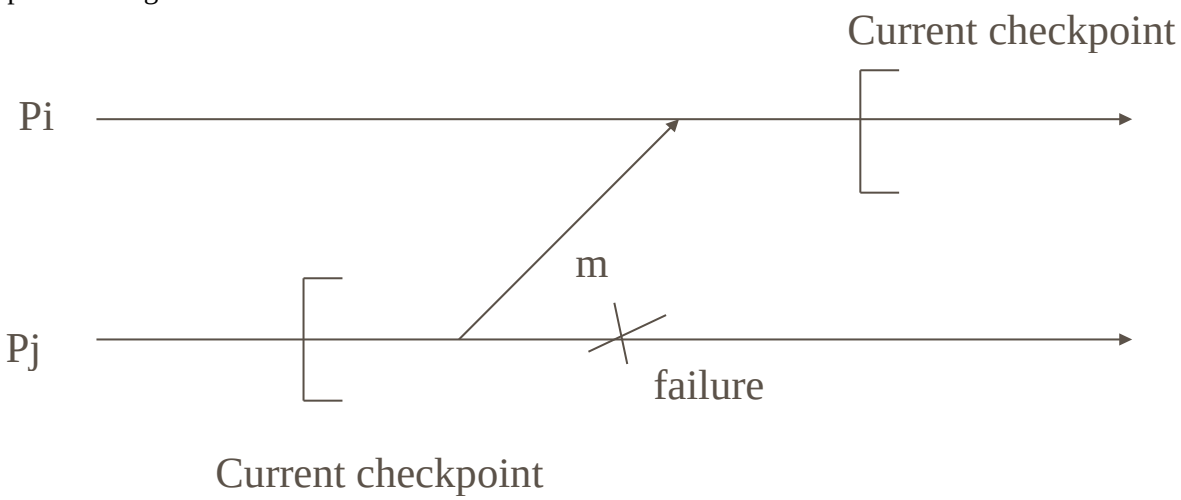
- *Lost message*. The state of process Pi indicates that it has sent a message *m* to process Pj. Pj has no record of receiving this message.

- *Orphan message*. The state of process Pj is such that it has received a message *m* from the process Pi but the state of the process Pi is such that it has never sent the message *m* to Pj.

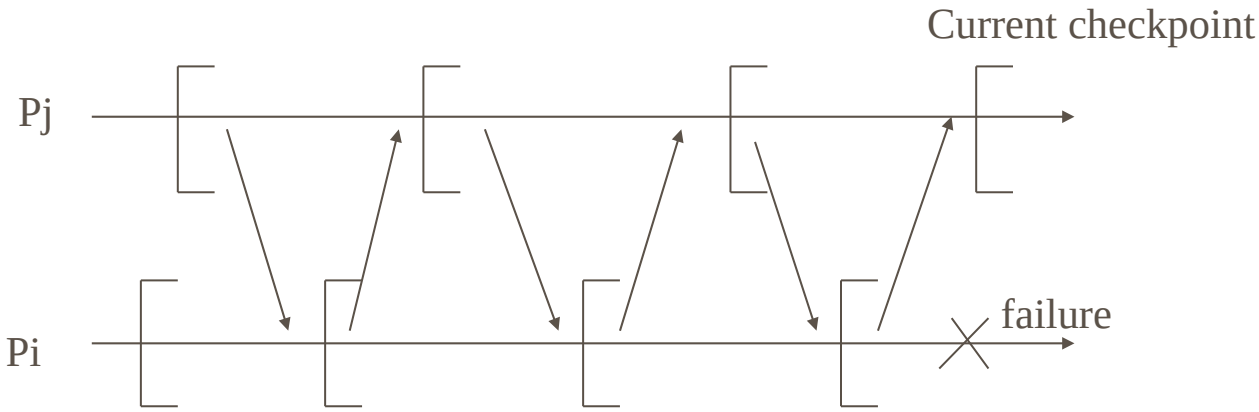
- A **strongly consistent set** of checkpoints consist of a set of local checkpoints such that there is no orphan or lost message.

- A **consistent set** of checkpoints consists of a set of local checkpoints such that there is no orphan message.

Orphan message



Domino effect



Synchronous checkpointing

- a processor Pi needs to take a checkpoint only if there is another process Pj that has taken a checkpoint that includes the receipt of a message from Pi and Pi has not recorded the sending of this message.

- In this way no orphan message will be generated.

Asynchronous checkpointing

■ Each process takes its checkpoints independently without any coordination.

Hybrid checkpointing

■ Synchronous checkpoints are established in a longer period while asynchronous checkpoints are used in a shorter period. That is, within a synchronous period there are several asynchronous periods.

Agreement in Faulty Systems

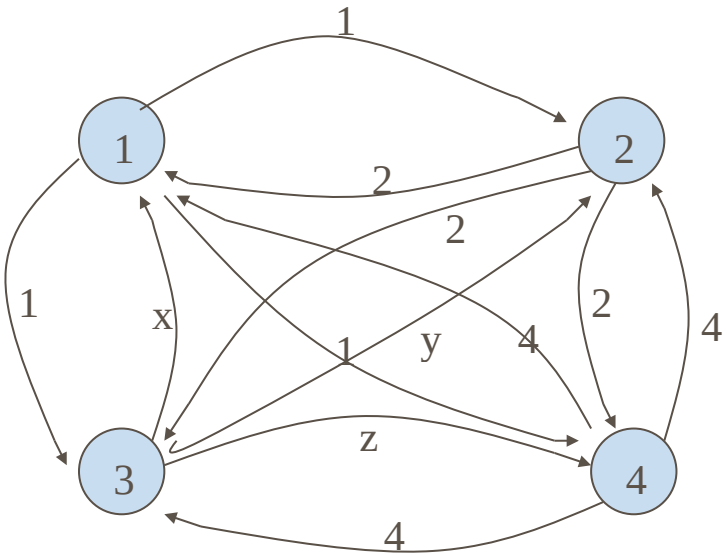
- **two-army problem**

- Two blue armies must reach agreement to attack a red army. If one blue army attacks by itself it will be slaughtered. They can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.

- They can never reach an agreement on attacking.

- Now assume the communication is perfect but the processors are not. The classical problem is called the **Byzantine generals problem.** **N** generals and **M** of them are traitors. Can they reach an agreement?

Lamport's algorithm



N = 4
M = 1

■ After the first round

1 got (1,2,x,4); 2 got (1,2,y,4);
3 got (1,2,3,4); 4 got (1,2,z,4)

■ After the second round

1 got (1,2,y,4), (a,b,c,d), (1,2,z,4)

2 got (1,2,x,4), (e,f,g,h), (1,2,z,4)

4 got (1,2,x,4), (1,2,y,4), (i,j,k,l)

■ Majority

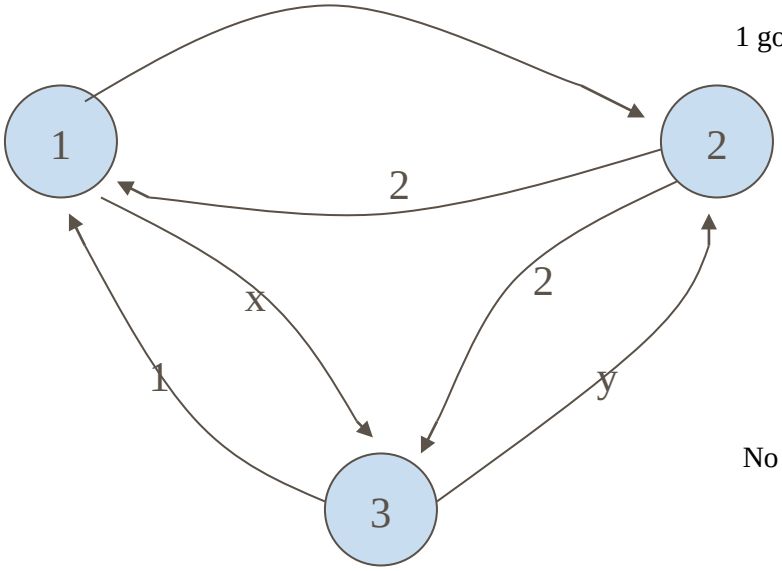
1 got (1,2,_,4); 2 got (1,2,_,4); 4 got (1,2,_,4)

■ So all the good generals know that 3 is the bad guy.

Result

- If there are m faulty processors, agreement can be achieved only if 2m+1 correct processors are present, for a total of 3m+1.

- Suppose n=3, m=1. Agreement cannot be reached.



1 got (1,2,x); 2 got (1,2,y); 3 got (1,2,3)

After the second round

1 got (1,2,y), (a,b,c)
2 got (1,2,x), (d,e,f)

No majority. Cannot reach an agreement.

Agreement under different models

- Turek and Shasha considered the following parameters for the agreement problem.

1. The system can be synchronous (A=1) or asynchronous (A=0).
2. Communication delay can be either bounded (B=1) or unbounded (B=0).
3. Messages can be either ordered (C=1) or unordered (C=0).
4. The transmission mechanism can be either point-to-point (D=0) or broadcast (D=1).

A Karnaugh map for the agreement problem

AB \ CD	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	1	1	1
10	0	0	1	1

- Minimizing the Boolean function we have the following expression for the conditions under which consensus is possible:
  AB+AC+CD = True

- (AB=1): Processors are synchronous and communication delay is bounded.
- (AC=1): Processors are synchronous and messages are ordered.

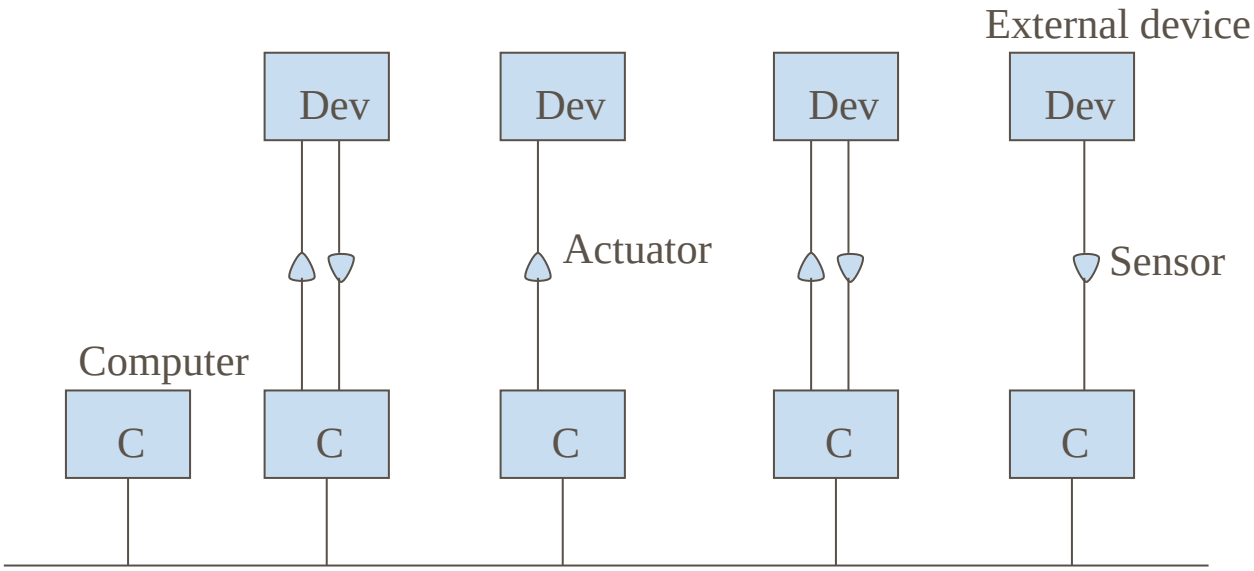
- (CD=1): Messages are ordered and the transmission mechanism is broadcast.

REAL-TIME DISTRIBUTED SYSTEMS

- **What is a real-time system?**

- **Real-time programs** interact with the external world in a way that involves time. When a stimulus appears, the system must respond to it in a certain way and before a certain deadline. E.g. automated factories, telephone switches, robots, automatic stock trading system.

Distributed real-time systems structure



Stimulus

■ An external device generates a stimulus for the computer, which must perform certain actions before a deadline.

1. Periodic: a stimulus occurring regularly every T seconds, such as a computer in a TV set or VCR getting a new frame every 1/60 of a second.
2. Aperiodic: stimulus that are recurrent, but not regular, as in the arrival of an aircraft in an air traffic controller's air space.
3. Sporadic: stimulus that are unexpected, such as a device overheating.

Two types of RTS

- Soft real-time systems: missing an occasional deadline is all right.

- Hard real-time systems: even a single missed deadline in a hard real-time system is unacceptable, as this might lead to loss of life or an environmental catastrophe.

Design issues

- **Clock Synchronization** - Keep the clocks in synchrony is a key issue.

- **Event-Triggered versus Time-Triggered Systems**

- **Predictability**

- **Fault Tolerance**

- **Language Support**

Event-triggered real-time system

- when a significant event in the outside world happens, it is detected by some sensor, which then causes the attached CPU to get an interrupt.
  Event-triggered systems are thus interrupt driven. Most real-time systems work this way.

- Disadvantage: they can fail under conditions of heavy load, that is, when many events are happening at once. This **event shower** may overwhelm the computing system and bring it down, potentially causing problems seriously.

Time-triggered real-time system

- in this kind of system, a clock interrupt occurs every T milliseconds. At each clock tick sensors are sampled and actuators are driven. No interrupts occur other than clock ticks.

- T must be chosen carefully. If it too small, too many clock interrupts. If it is too large, serious events may not be noticed until it is too late.

An example to show the difference between the two

- Consider an elevator controller in a 100-story building. Suppose that the elevator is sitting on the 60^{th} floor. If someone pushes the call button on the first floor, and then someone else pushes the call button on the 100^{th} floor. In an eventtriggered system, the elevator will go down to first floor and then to 100^{th} floor. But in a timetriggered system, if both calls fall within one sampling period, the controller will have to make a decision whether to go up or go down, for example, using the nearest-customer-first rule.

- In summary, event-triggered designs give faster response at low load but more overhead and chance of failure at high load. Time-trigger designs have the opposite properties and are furthermore only suitable in a relatively static environment in which a great deal is known about system behavior in advance.

Predictability

■ One of the most important properties of any real-time system is that its behavior be predictable. Ideally, it should be clear at design time that the system can meet all of its deadlines, even at peak load. It is known when event E is detected, the order of processes running and the worst-case behavior of these processes.

Fault Tolerance

- Many real-time systems control safety-critical devices in vehicles, hospitals, and power plants, so fault tolerance is frequently an issue.

- Primary-backup schemes are less popular because deadlines may be missed during cutover after the primary fails.

- In a safety-critical system, it is especially important that the system be able to handle the worst-case scenario. It is not enough to say that the probability of three components failing at once is so low that it can be ignored. Faulttolerant real-time systems must be able to cope with the maximum number of faults and the maximum load at the same time.

Language Support

- In such a language, it should be easy to express the work as a collection of short tasks that can be scheduled independently.
- The language should be designed so that the maximum execution time of every task can be computed at compile time. This requirement means that the language cannot support general **while** loops and recursions.
- The language needs a way to deal with time itself.
- The language should have a way to express minimum and maximum delays.
- There should be a way to express what to do if an expected event does not occur within a certain interval.
- Because periodic events play an important role, it would be useful to have a statement of the form: every (25 msec){…} that causes the statements within the curly brackets to be executed every 25 msec.

Real-Time Communication

- Cannot use Ethernet because it is not predictable.
- Token ring LAN is predictable. Bounded by kn byte times. K is the machine number. N is a n-byte message .
- An alternative to a token ring is the TDMA (Time Division Multiple Access) protocol. Here traffic is organized in fixed-size frames, each of which contains n slots. Each slot is assigned to one processor, which may use it to transmit a packet when its time comes. In this way collisions are avoided, the delay is bounded, and each processor gets a guaranteed fraction of the bandwidth.

Real-Time Scheduling

- Hard real time versus soft real time
- Preemptive versus nonpreemptive scheduling
- Dynamic versus static
- Centralized versus decentralized

Dynamic Scheduling

- **1. Rate monotonic algorithm:**
- It works like this: in advance, each task is assigned a priority equal to its execution frequency. For example, a task runs every 20 msec is assigned priority 50 and a task run every 100 msec is assigned priority 10. At run time, the scheduler always selects the highest priority task to run, preempting the current task if need be.
- **2.Earliest deadline first algorithm:**
- Whenever an event is detected, the scheduler adds it to the list of waiting tasks. This list is always keep sorted by deadline, closest deadline first.
- **3.Least laxity algorithm:**

- this algorithm first computes for each task the amount of time it has to spare, called the laxity. For a task that must finish in 200 msec but has another 150 msec to run, the laxity is 50 msec. This algorithm chooses the task with the least laxity, that is, the one with the least breathing room.

Static Scheduling

■ The goal is to find an assignment of tasks to processors and for each processor, a static schedule giving the order in which the tasks are to be run.

A comparison of Dynamic versus Static Scheduling

- Static is good for time-triggered design.
- 1. It must be carefully planned in advance, with considerable effort going into choosing the various parameters.
- 2. In a hard real-time system, wasting resources is often the price that must be paid to guarantee that all deadlines will be met.
- 3. An optimal or nearly optimal schedule can be derived in advance.
- Dynamic is good for event-triggered design.
- 1. It does not require as much advance work, since scheduling decisions are made on-the-fly, during execution.
- 2. It can make better use of resources than static scheduling.
- 3. No time to find the best schedule.

A **Distributed File System (DFS)** as the name suggests, is a file system that is distributed on multiple file servers or multiple locations. It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer.

The main purpose of the Distributed File System (DFS) is to allows users of physically distributed systems to share their data and resources by using a Common File System. A collection of workstations and mainframes connected by a Local Area Network (LAN) is a configuration on Distributed File System. A DFS is executed as a part of the operating system. In DFS, a namespace is created and this process is transparent for the clients.

DFS has two components:

- **Location Transparency –**
  Location Transparency achieves through the namespace component.
- **Redundancy –**
  Redundancy is done through a file replication component.

In the case of failure and heavy load, these components together improve data availability by allowing the sharing of data in different locations to be logically grouped under one folder, which is known as the “DFS root”.

It is not necessary to use both the two components of DFS together, it is possible to use the namespace component without using the file replication component and it is perfectly possible to use the file replication component without using the namespace component between servers.

File system replication:
Early iterations of DFS made use of Microsoft’s File Replication Service (FRS), which allowed for straightforward file replication between servers. The most recent iterations of the whole file are distributed to all servers by FRS, which recognises new or updated files.

"DFS Replication" was developed by Windows Server 2003 R2 (DFSR). By only copying the portions of files that have changed and minimising network traffic with data compression, it helps to improve FRS. Additionally, it provides users with flexible configuration options to manage network traffic on a configurable schedule.

Features of DFS :

- **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.
- **User mobility :**
  It will automatically bring the user's home directory to the node where the user logs in.
- **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.
- **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.
- **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.
- **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.
- **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.
- **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.
- **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.
- **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.

History :

The server component of the Distributed File System was initially introduced as an add-on feature. It was added to Windows NT 4.0 Server and was known as "DFS 4.1". Then later on it was included as a standard component for all editions of Windows 2000 Server. Client-side support has been included in Windows NT 4.0 and also in later on version of Windows.

Linux kernels 2.6.14 and versions after it come with an SMB client VFS known as "cifs" which supports DFS. Mac OS X 10.7 (lion) and onwards supports Mac OS X DFS.

Properties:

- File transparency: users can access files without knowing where they are physically stored on the network.
- Load balancing: the file system can distribute file access requests across multiple computers to improve performance and reliability.
- Data replication: the file system can store copies of files on multiple computers to ensure that the files are available even if one of the computers fails.
- Security: the file system can enforce access control policies to ensure that only authorized users can access files.
- Scalability: the file system can support a large number of users and a large number of files.
- Concurrent access: multiple users can access and modify the same file at the same time.
- Fault tolerance: the file system can continue to operate even if one or more of its components fail.
- Data integrity: the file system can ensure that the data stored in the files is accurate and has not been corrupted.
- File migration: the file system can move files from one location to another without interrupting access to the files.
- Data consistency: changes made to a file by one user are immediately visible to all other users.
  Support for different file types: the file system can support a wide range of file types, including text files, image files, and video files.

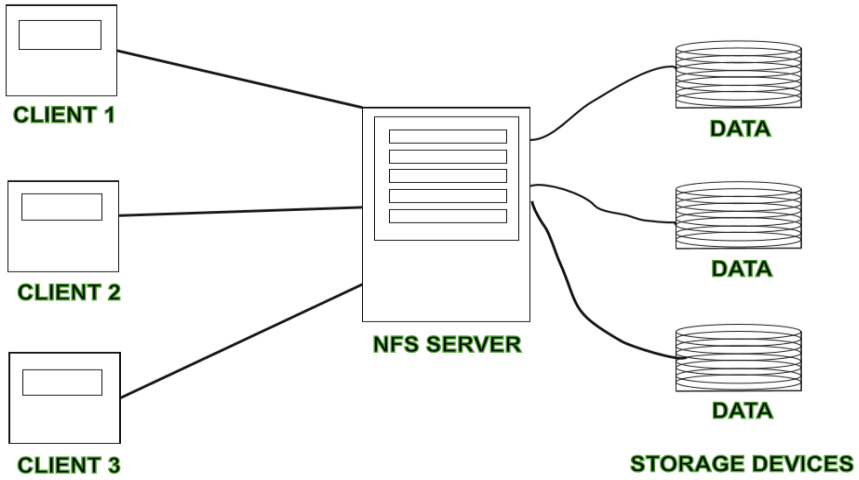
Applications :

- **NFS –**
  NFS stands for Network File System. It is a client-server architecture that allows a computer user to view, store, and update files remotely. The protocol of NFS is one of the several distributed file system standards for Network-Attached Storage (NAS).
- **CIFS –**
  CIFS stands for Common Internet File System. CIFS is an accent of SMB. That is, CIFS is an application of SIMB protocol, designed by Microsoft.
- **SMB –**
  SMB stands for Server Message Block. It is a protocol for sharing a file and was invented by IMB. The SMB protocol was created to allow computers to perform read and write operations on files to a remote host over a Local Area Network (LAN). The directories present in the remote host can be accessed via SMB and are called as "shares".
- **Hadoop –**
  Hadoop is a group of open-source software services. It gives a software framework for distributed storage and operating of big data using the MapReduce programming model. The core of Hadoop contains a storage part, known as Hadoop Distributed File System (HDFS), and an operating part which is a MapReduce programming model.
- **NetWare –**
  NetWare is an abandon computer network operating system developed by Novell, Inc. It primarily used combined multitasking to run different services on a personal computer, using the IPX network protocol.

Working of DFS :

There are two ways in which DFS can be implemented:

- **Standalone DFS namespace –**
  It allows only for those DFS roots that exist on the local computer and are not using Active Directory. A Standalone DFS can only be acquired on those computers on which it is created. It does not provide any fault liberation and cannot be linked to any other DFS. Standalone DFS roots are rarely come across because of their limited advantage.
- **Domain-based DFS namespace –**
  It stores the configuration of DFS in Active Directory, creating the DFS namespace root accessible at \\<**domainname**>\<**dfsroot**> or \\<**FQDN**>\<**dfsroot**>



Advantages :

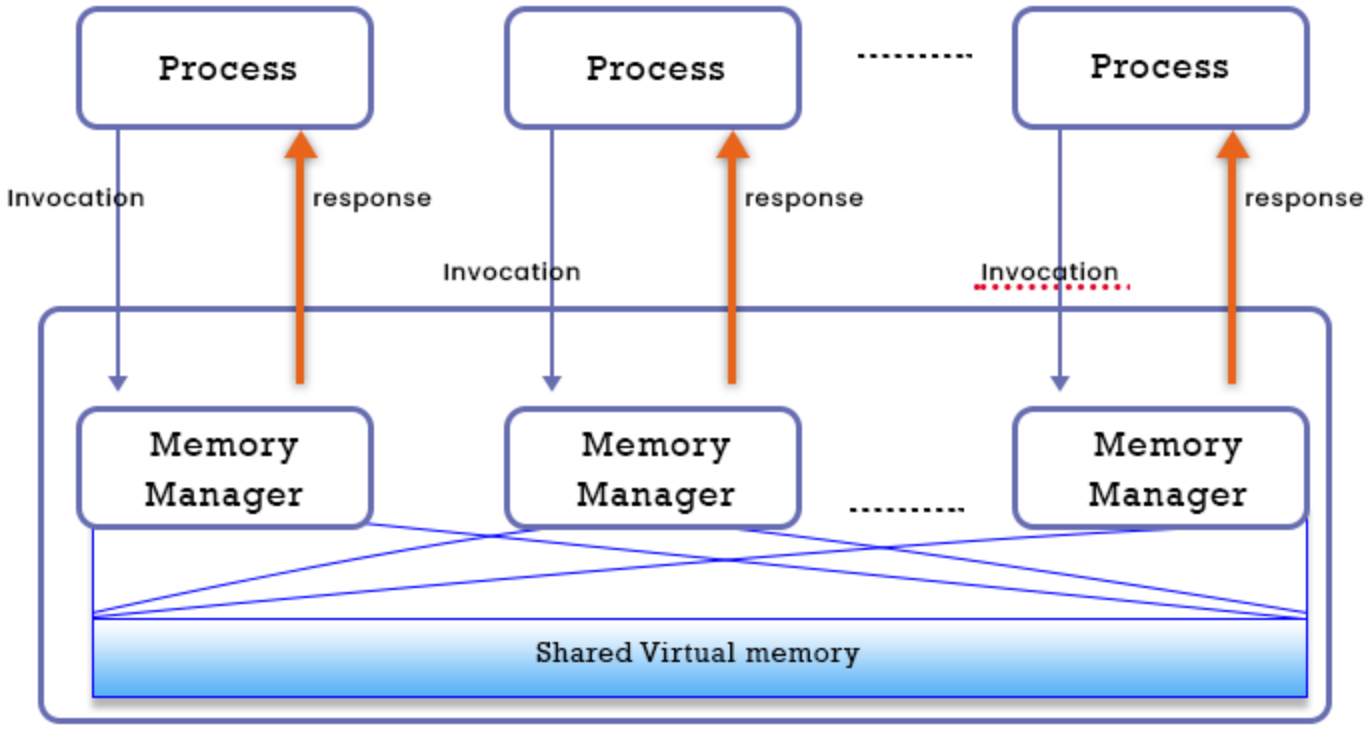
- DFS allows multiple user to access or store the data.
- It allows the data to be share remotely.
- It improved the availability of file, access time, and network efficiency.
- Improved the capacity to change the size of the data and also improves the ability to exchange the data.
- Distributed File System provides transparency of data even if server or disk fails.

Disadvantages :

- In Distributed File System nodes and connections needs to be secured therefore we can say that security is at stake.
- There is a possibility of lose of messages and data in the network while movement from one node to another.
- Database connection in case of Distributed File System is complicated.
- Also handling of the database is not easy in Distributed File System as compared to a single user system.
- There are chances that overloading will take place if all nodes tries to send data at once.

UNIT-4

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. 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.



Distributed shared memory

Let us now discuss some different types of Distributed shared memory:

1. **On-Chip Memory:**
    1. All the data are stored in the CPU's chip.
    2. There is a direct connection between Memory and address lines.
    3. The On-Chip Memory DSM is very costly and complicated.
2. **Bus-Based Microprocessor**
    1. The connection between the memory and the CPU is established through a number of parallel wires that acts as a bus.
    2. 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.
    3. The network traffic is reduced by the cache memory.

3. **Ring-based Microprocessor**
    1. In Ring-based DSM, there is no centralized memory present
    2. All the nodes are connected through some link/network, and accessing of memory is done by the token passing
    3. All the nodes/computers present in the shared area access a single address line in this system

Let us now see some advantages of Distributed Shared Memory system:

1. Easy Abstraction - Since the address space is the same, data migration is not an issue for programmers, making it simpler to build than RPC.
2. Easier Potability - Sequential to distributed system migration is made easy by the access protocols employed in DSM. Because they make use of a common programming interface, DSM programmes are portable.
3. Locality of data - Data that is being fetched in large blocks, or data that is close to the memory address being fetched, may be needed in the future.
4. Larger memory space - Large virtual memory space is provided, paging operations are decreased, and the total memory size is the sum of the memory sizes of all the nodes.
5. Better Performance - It speeds up the data access in order to increase the performance of the system.
6. Flexible communication Environment - There is no requirement for the sender and receiver to be present, thus they can join and leave the DSM system without influencing the others.
7. Simplified process Migration - Since they all share the same address space, moving one process to a different computer is simple

Some more basic advantages of the distributed shared memory (DSM) system are listed below:

1. It is less expensive than using multiprocessing systems
2. Data access is done smoothly
3. It provides better scalability as several nodes can access the memory.

Now let's see the disadvantages of the distributed shared memory:

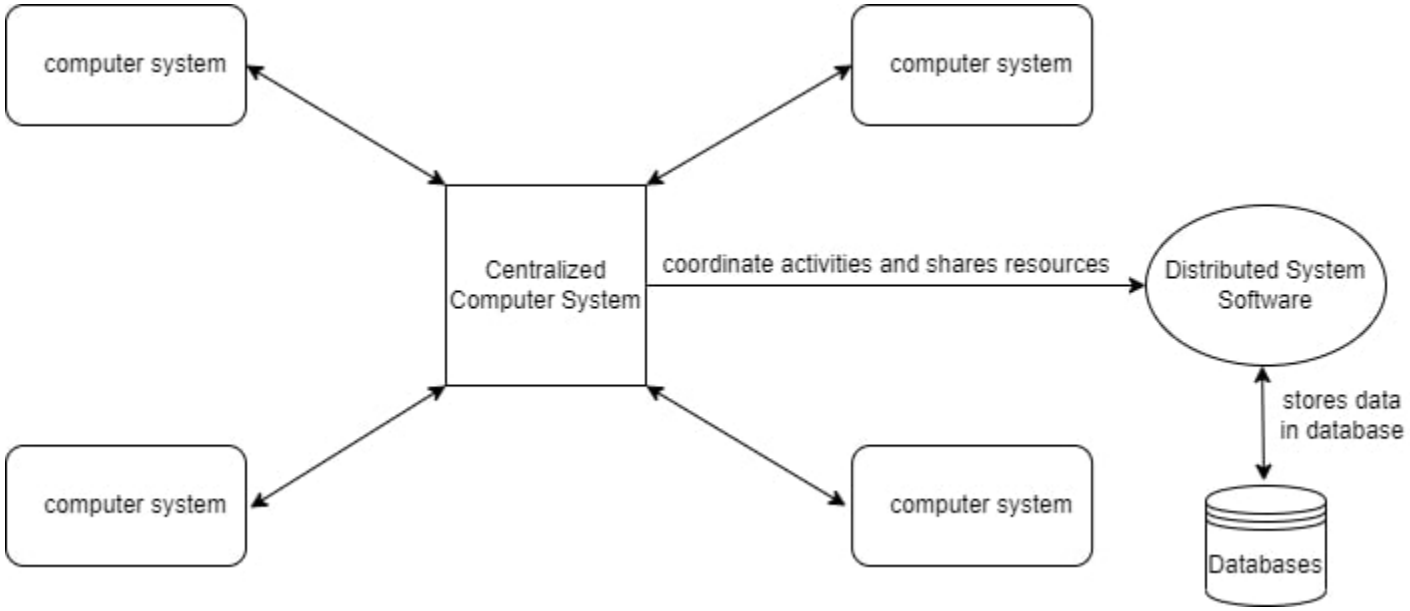
1. Accessing is faster in a non-distributed shared memory system than in a distributed system.
2. Simultaneous access to data has always been a topic of discussion. It should ensure some additional protection to it.
3. It generates some programmer control over the actual message
4. In order to write correct programs, programmers must learn the consistency model
5. It is not much efficient as the message-passing implementation as it uses the asynchronous message-passing implementations.

Difference between Message Passing and Distributed shared memory:

Distributed shared memory	Message Passing
Variables are directly shared	Variables are formalized.
Less cost of communication	More Costs of communication
Errors may occur by altering the data by the processes.	Private address space is associated with the processes which prevents the occurrence of errors.
Processes cannot run simultaneously.	Processes may run at the same time.

Consistency Models

A consistency model is a set of rules that govern the behavior of a distributed system. It establishes the circumstances in which the system's various parts can communicate with one another and decides how the system will react to modifications or errors. A distributed system's consistency model plays a key role in ensuring the system's consistency and dependability in the face of distributed computing difficulties including network delays and partial failures.



Consistency models in distributed systems refer to the guarantees provided by the system about the order in which operations appear to occur to clients. Specifically, it determines how data is accessed and updated across multiple nodes in a distributed system, and how these updates are made available to clients.

There are various types of consistency models available in distributed systems. Each consistency model has its own strengths and weaknesses, and the choice of model depends on the specific requirements of the system.

Types of Consistency Models

Strong Consistency: In a strongly consistent system, all nodes in the system agree on the order in which operations occurred. Reads will always return the most recent version of the data, and writes will be visible to all nodes immediately after they occur. This model provides the highest level of consistency. There are some performance and availability issues.

Process	Write Operation	Read Operation
P1	Write(x)a	

Process	Write Operation	Read Operation
P2		Read(x)a

Pipelined Random Access Memory Or FIFO Consistency Model: PRAM is one of the weak consistency models. Here, all processes view the operations of a single process in the same order that they were issued by that process, while operations issued by different processes can be viewed in a different order from different processes.

P1	P2	P3	P4
Write(x)a			
	Read(x)a		
	Write(x)b		
		Read(x)a	Read(x)b
		Read(x)b	Read(x)a

Sequential Consistency Model: This model was proposed by Lamport. In this sequential consistency model, the result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. The sequential consistency model is weak as compared to the strict consistency.

Process	Write Operation	Read Operation
P1	Write(x)a	
P2		Read(x)a
P3	Write(x)b	

Causal Consistency Model: The causal consistency model was introduced by Hutto and Ahamad in 1990. It makes sure that All processes see causally-related shared accesses in the same order. The causal consistency model is weaker as compared to strict or sequential consistency because the difference between causal consistency and sequential consistency is that causal consistency does not require a total order.

P1	P2	P3	P4
Write(x)a	Read(x)a	Read(x)a	Read(x)a
	Write(x)b		
Write(x)c		Read(x)c	Read(x)b
		Read(x)b	Read(x)c

Weak Consistency Model: A weakly consistent system provides no guarantees about the ordering of operations or the state of the data at any given time. Clients may see different versions of the data depending on which node they connect to. This model provides the highest availability and scalability but at the cost of consistency.

Processor Consistency Model: The processor consistency model was introduced by Goodman in 1989 and is analogous to the PRAM consistency model but there's a small difference i.e there's a restriction of memory coherence. Memory coherence refers to the fact that all processes should reflect in the same order for all write operations to any given memory address.

Page based distributed shared memory

- It is also called NORMA (No Remote Memory Access).
- The project deals with extending the concept of shared memory (an IPC mechanism) for a distributed environment. Hence, the programmer is freed from the task of implicit message passing in the program.
- Processors connected by a specialized message passing network, work station on LAN or similar designs.
- The essential elements here is that no processor can directly access any other processor's memory.

Major design issues in classical DSM system

1. **Granularity**
   - It refers to the block size of a DSM system.
   - It is the amount of data sent with each update.
   - If granularity is too large, a whole page (or more) would be sent for an update to a single byte, reducing efficiency.
2. **Structure of shared memory**
   - It refers to the layout of the shared data in memory.
   - Dependent on the type of applications that the DSM system is intended to support:
   - By datatype
   - By Database
   - No structuring(Simple linear array)
3. **Replacement strategy**
   - If the local memory of a node is full, a cache miss at that node implies not only a fetch of accessed data block from a remote not but also a replacement.
   - Data block must be replaced by the new data block.
4. **Thrashing**
   - It occurs when a network resources are exhausted & more time is spent in validating data & sending updates than is used doing actual work.
   - Based on system specification one should choose write, update or write- invalidate to avoid thrashing.
5. **Memory Coherence & Access Synchronization**
   - In a DSM system, concurrent access to shared data may be generated.
   - Therefore, memory coherence protocol & access synchronization is needed to maintain the consistency of shared data.
6. **Data location & access:** To share data in a DSM system, it must implement sum form of data block locating mechanism in order to service network data block fault & to meet the requirement of the memory coherence semantics solids are being used.

Shared variables distributed shared memory:

- A more structured approach is to share only certain variable & data structure that are needed by more than one processor.
- By this problem changes from how to do paging over the network to how to maintain a potentially replicated distributed database consisting of shared variable

1. **Munin:** It is a DSM system that is fundamentally based on software objects, but which can place each object on a separate page so the hardware MMU can be used for detecting accessed to shared object.

   a) Release consistence: Munin distinguish three classes of variables.
   - Ordinary variables (Read & Write only)
   - Shared variables (visible to multiple processes)
   - Synchronization variables (locks & barriers)

   b) Multiple protocols
   - Read only
   - Migratory (lock must firstly acquired)
   - Write-shared (two or more processor write at same time)
   - Conventional (shared variable that are not anotated)

   c) Directories
   - It uses directories to locate pages containing shared variables.

   d) Synchronization
   - Munin maintains a second directory for synchronization variables.

2. **Midway:** It is a DSM that based on sharing individual data structures. Its goals were to allow existing and new multiprocessor programs to run efficiently on multi computers with only small changes to the code.

a) entry consistency: To make the entry consistency work midway requires the program have 3 characteristics that multiprocessor program do not have:
- Shared variables must be declared using the new keyword shared
- Each shared variable must be associated with a lock or barrier.
- Shared variables may only be accessed inside critical section.