Sardar Patel University,Balaghat(M.P)
School of Engineering & Technology
Department of CSE
Subject Name: Distributed System
Subject Code: CSE701
Semester: 7th

Unit-1

A Distributed System (DS) is one in which

- Hardware and software components, located at remote networked computers, coordinate and communicate their actions only by passing messages. Any distance may separate computers in the network.
- Sharing of resources is the main motivation of distributed systems. Resources may be managed by servers and accessed by clients, or they may be encapsulated as objects and accessed by client objects.

A distributed operating system runs on multiple independent computers, connected through communication network, but appears to its users as a single virtual machine and runs its own OS.

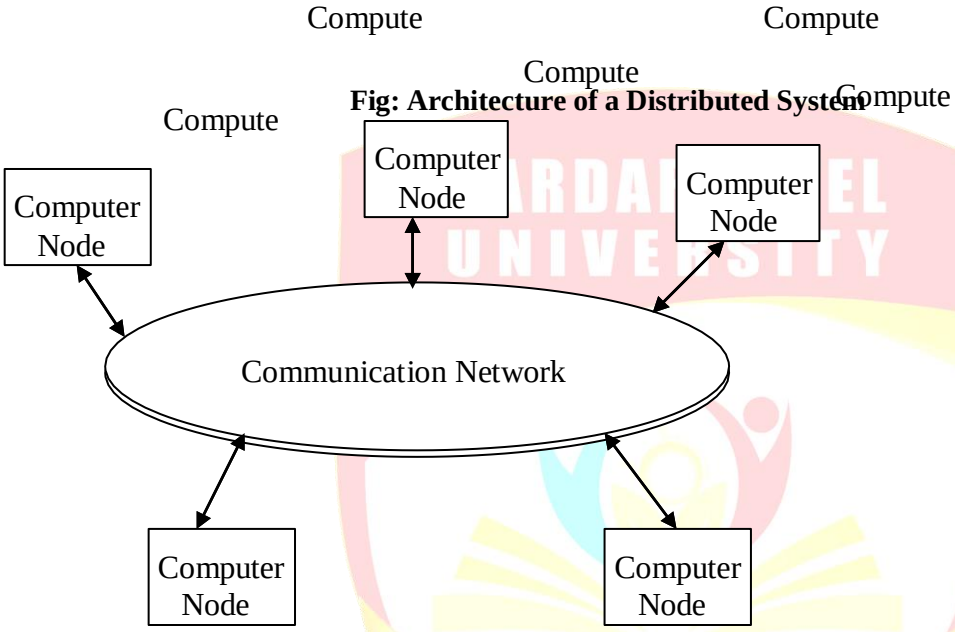


Fig: Architecture of a Distributed System

Architecture of a Distributed System

Characteristics of Distributed Systems

A Distributed System has the following characteristics:

- It consists of several independent computers connected through communication network,
- The computers communicate with each other by exchanging message over a communication network.
- Each computer has its own memory, clock and runs its own operating system.
- Each computer has its own resources, called local resources
- Remote resources are accessed through the network

Goals of Distributed system:

There are four important goals that should be met to make building a distributed system worth the effort. A distributed system should make resources easily accessible; it should reasonably hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

1 Making Resources Accessible: - The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way.

2 Distribution Transparency: - An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent. Types of Transparency:-

Transparency	**Description**
Access	Hide differences in data representation and how a resource is accessed.
Location	Hide where a resource is located.
Migration	Hide that a resource may move to another location while in use.
Relocation	Hide that a resource may be moved to another location while in use.
Replication	Hide that a resource is replicated.
Concurrency	Hide that a resource may be shared by several competitive users.
Failure	Hide the failure and recovery of a resource.

Degree of Transparency: aiming for distribution transparency may be a nice goal when designing and implementing distributed systems, but that it should be considered together with other issues such as performance and comprehensibility. There are situations in which it is not at all obvious that hiding distribution is a good idea. As distributed systems are expanding todevices that people carry around, and where the very notion of location and context awareness is becoming increasingly important, it may be best to actually expose distribution rather than trying to hide it. This distribution exposure will become more evident when we discuss embedded and ubiquitous distributed systems later in this chapter. As a simple example, consider an office worker who wants to print a file from her notebook computer. It is better to send the print job to a busy nearby printer, rather than to an idle one at corporate headquarters in a different country. There are also other arguments against distribution transparency. Recognizing that full distribution transparency is simply impossible, we should ask ourselves whether it is even wise to pretend that we can achieve it. It may be much better to make distribution explicit so that the user and application developer are never tricked into believing that there is such a thing as transparency. The result will be that users will much better understand the (sometimes unexpected) behavior of a distributed system, and are thus much better prepared to deal with this behavior.

3 Openness: - An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. Important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers). Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be extensible.

4 Scalability: - Scalability of a system can be measured along at least three different dimensions. First, a system can be scalable with respect to its **size**, meaning that we can easily add more users and resources to the system. Second, a **geographically scalable** system is one in which the users and resources may lie far apart. Third, a system can be **administratively scalable**, that it can still be easy to manage even if it spans many independent administrative organizations.

Advantages of Distributed Systems are:

- **Resource Sharing**: Due to communication between connected computers resources can be shared among computers.
- **Enhance Performance**: This is due to the fact that many tasks can be executed concurrently at different computers. Load distribution among computers can further improve response time.
- **Improved reliability and availability: I**ncreased reliability is due to the fact that if few computers fail others are available and hence the system continues.
- **Modular expandability**: New hardware and software resources can be added without replacing the existing resources.

Inherent Limitations of Distributed Systems

The lack of common memory and system wide common clock is an inherent problem in distributed systems.

- Without a shared memory, up-to-date information about the state of the system is not available to every process via a simple memory lookup. The state information must therefore be collected through communication.

- In the absence of global time, it becomes difficult to talk about temporal order of events. The combination of unpredictable communication delays and the lack of global time in a distributed system make it difficult to know how up-to-date collected state information really is.

System Architecture Types

Distributed systems can be modeled into several types. Various models are used for building distributed computing systems. These models can be broadly classified into five categories, and they are described below:

1. Mini Computer Model,
2. Workstation Model,
3. Workstation Server Model,
4. Processor Pool Model, and
5. Hybrid Model.

1. Mini Computer Model

In this model, the distributed system consists of several minicomputers. Each computer supports multiple users and provides access to remote resources. The ratio of processors to users is normally less than one.

Minicomputer model is a simple extension of the centralized time-sharing system. As shown in Figure 1, a distributed computing system based on this model consists of a few minicomputers. They may be large supercomputers as well interconnected by a communication network. Each minicomputer usually has multiple users simultaneously logged on to it.

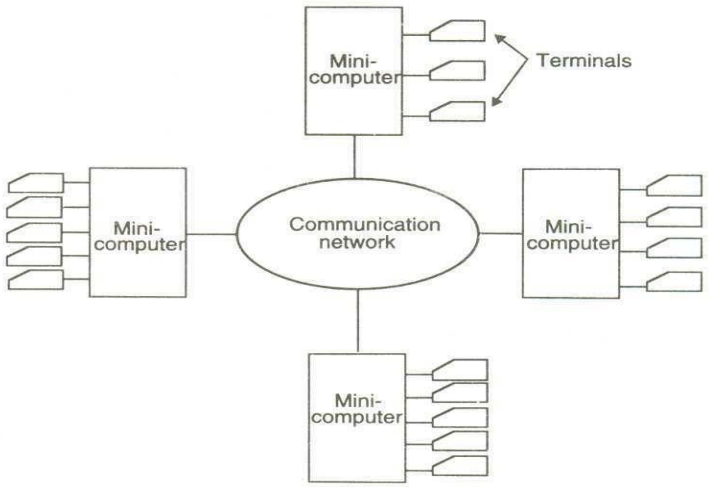


Figure 1: The distributed system based on minicomputer model

Several interactive terminals are connected to each minicomputer. Each user is logged on to one specific minicomputer, with remote access to other minicomputers. The network allows a user to access |remote resources that are available on some machine other that the one on to which the user is currently logged.

The minicomputer model may be used when resource sharing (such as sharing of information databases of different types, with each type of database located on a different machine) with remote users is desired. The early ARPAnet is an example of a distributed computing system based on the minicomputer model.

2. Workstation Model

In this model, the distributed system consists of several workstations; every user has a workstation where user's work is performed. With the help of distributed file system, a user can access data regardless of the location of the data. The ratio of processors to users is normally one. The workstations are independent computers with memory, hard disks, keyboard and console. Workstations are connected with each other through communication network.

As shown in Figure 2, a distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. A company's office or a university department may have several workstations scattered throughout a building or compass

each workstation equipped with its own disk and serving as a single-user computer.

It has been often found that in such an environment at any one time (especially at night), a significant proportion of the workstations are idle, resulting in the waste of large amounts of CPU time. Therefore, the idea of the workstation model is to interconnect all these workstations by a high-speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.

In this model a user logs onto one of the workstations called his or her *home workstation* and submits jobs for execution. When the system finds that the user's workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the processes from the user's workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user's, in. workstation.

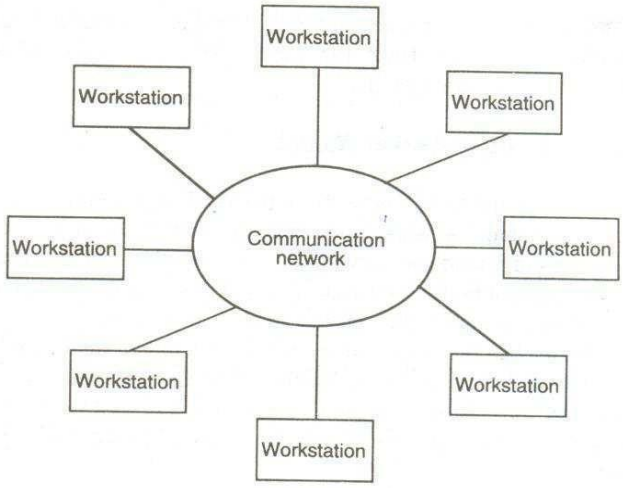


Figure 2: A distributed system based on the workstation model

This model is not so simple to implement because several issues must be resolved. These issues are as follows:

- How does the system find an idle workstation?
- How is a process transferred from one workstation to get it executed on another workstation?
- What happens to a remote process if a user logs onto a workstation that was idle until now and was being used to execute a process of another workstation?

Three commonly used approaches for handling the third issue are as follows:

1. The first approach is to allow the remote process share the resources of the workstation along win its own logged -on .user's processes. This method is easy to implement, but it

defeats the main idea of workstations serving as personal computers, because if remote processes are allowed to execute simultaneously with the logged-on user's own processes, the logged-on user does not get his or her guaranteed response.

2. The second approach is to kill the remote process. The main drawbacks of this method are that all processing done by the remote process gets lost and tie file system may be left in an inconsistent state; making this method unattractive.
3. The third approach is, migrate the remote process back to its home workstation, so that its execution can be continued there. This method is difficult to implement because it requires the system to support preemptive process migration facility.

The Sprite system developed at Xerox is an examples of distributed computing systems based on the workstation model.

3. Workstation-Server Model

A workstation with its own local disk is usually called a *diskfull* workstation and a workstation without a local disk is called a *diskless* workstation. With high-speed networks, diskless workstations have become more popular than diskfull workstations, making the workstation-server model more popular than the workstation model for building distributed computing systems.

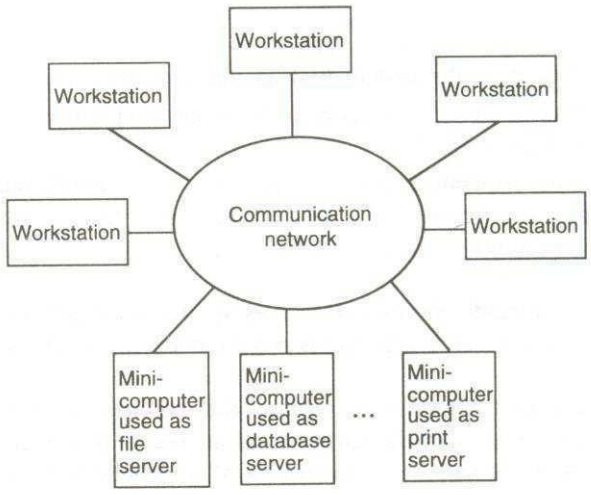


Fig 3: A Distributed System based on the workstation-server model

As shown in Figure 3, a distributed computing system based on the workstation-server model consists of a few minicomputers and several workstations interconnected by a communication network. Most of the workstation may be diskless, but a few of may be disk full.

When diskless workstations are used on a network, the file system to be used by these workstations must be implemented either by a diskfull workstation or by a minicomputer equipped with a disk for file storage.

One or more of the minicomputers are used for implementing the file system. Other minicomputers may be used for providing other types of services, such as database service and print service. Therefore, each minicomputer is used as a server machine to provide one or more types of services. For a number of reasons, such as higher reliability, and better scalability, multiple servers are often used for managing the resources of a particular type in a distributed computing system.

For example, there may be multiple file servers, each running on a separate minicomputer and cooperating via the networks for managing the files of all the users in file system.

In this model, a user logs onto a workstation called his or her home workstation. Normal computation activities required by the user's processes are performed at the user's home workstation, but requests for services provided by special servers (such as a file server or a database server) are sent to a server providing that type of service that performs the user's requested activity and returns the result of request processing to the user's workstation. Therefore, in this model, the user's processes need not be migrated to the server machine for getting the work done by those machines.

For better overall system performance the local disk of a dishful workstation is normally used for such purposes as storage of temporary files, storage of unshared files, storage of shared files that are rarely changed, paging activity in virtual-memory management, and caching of remotely accessed data

As compared to the workstation model, the workstation-server model has several advantages:

1. It is much cheaper to use a few minicomputers equipped with large fast disks that are accessed over the network than a large number of dishful workstations, with each workstation having a small, slow disk.
2. Diskless workstations are also preferred to dishful workstations from a system maintenance point of view. Software installation, backup and hardware maintenance are easier to perform with a few large disks than win many small disks scattered all over a building or campus.
3. In the workstation-server model, since all files are managed by be file servers, users have the flexibility to use any workstation and access the files in the same manner irrespective of which workstation the user is currently logged on. Note that this is not true win the workstation model, in which each workstation has its local file system, because different mechanisms are needed to access local and remote files.

4. In the workstation-server model, the request-response protocol described above is mainly used to access the services of the server machine. Therefore unlike the workstation model, this model does not need a process migration facility which is difficult to implement.

The request-response protocol is known as the client-server model of

communication. In this model, a client process (which in this case resides on a workstation) sends a request to a serve process (which In this case resides on a computer) for getting some service such as reading a block of a file to the server executes the request and sends back a reply to the cheat that contains the result of request processing.

The client-server model provides an effective general-purpose approach to the sharing of information and resources in distributed computing systems It is not only meant for use with the workstation-server model but also can be Implemented in a variety of hardware and software environments. The computers used to run the client and server processes need not necessarily be workstations and minicomputers. They can be of many types and there is no need to distinguish between them It is even possible for both the client and server processes to be run on the same computer. Moreover some processes are both client and server processes That is, a server process may use the services of another server, appearing as a client to the latter

5. A user has guaranteed response time because workstations are not used for executing remote processes. However the model does not utilize the processing capability of idle workstations.

4. Processor Pooled Model

Processor-pool model is based on the observation that most of the time a user does not need any computing power but once m a while he or she may need a very large amount of computing power for a short time. Therefore, in the processor-pooled model the processors are pooled together to be shared by the users as needed. The pool of processors consists of a large number of microcomputer and minicomputers attached to the network. Each processor m the pool has its own memory to load and run a system program or an application program of the distributed computing system

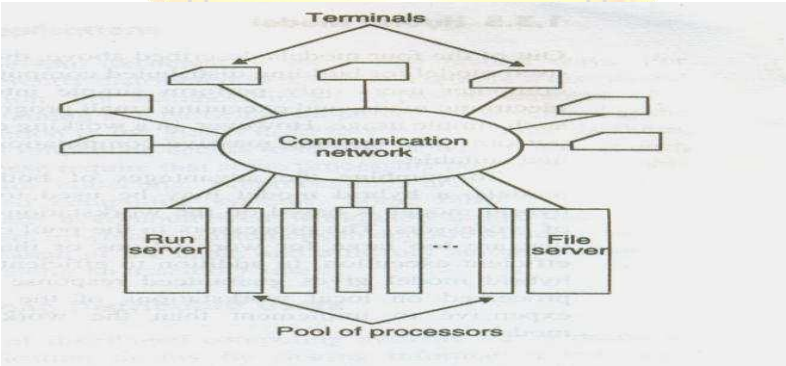


Figure 5: A distributed computing system based on processor-poor model

In the pure processor's model, the processors m the pool have no terminals attached directly to them, and users access the system from terminals that are attached to the network via special devices. These terminals are either small diskless workstations or graphic terminals.

A special server, called a run server, manages and allocates the processors in the pool to

different users on a demand bases. When a user submits a Job for computation, an appropriate number of professors are temporarily assigned to the job by the run server.

For Example

- If the user's computation job is the compilation of a program having n segments,
- Each of the segments can be compiled independently to produce separate releasable object files;
- n processors from the pool can be allocated to this job to compile all the segments in parallel.
- When the computation is completed, the processors are returned to the pool for use by other users

In the processor-pool model there is no concept of a home machine. That is, a user does not log onto a particular machine but to the system as a whole This is in contrast to other models in which each user has a home machine (e.g., a workstation or minicomputer) onto which he or she logs and runs most of his or her programs they by default.

As compared to the workstation-server model, the processor-pool model allows better utilization of the available processing power of a distributed computing system. This is

becauseIn the processor-pool model, the entire processing power of the system is available for use by the currently logged-users, whereas this is not true for the workstation-server

- model in which several workstations may be idle at a particular time but they cannot be used for processing the jobs of other users.
- Furthermore the processor-pool model provides greater flexibility than the workstation-server model in the sense that the system's services can be easily expanded without the need to install any more computers
- The professors in the pool can be allocated to act as extra servers to carry any additional load arising from an increased user population or to provide new services.

However, the processor-pool model is usually considered to be unsuitable for high-performance interactive applications, especially those using graphics or window systems. This is mainly because of the slow speed of communication between the computer on which the application program of a user is being executed and the terminal via which the user is interacting with to system. The workstation-server model is genially considered to be more suitable for such applications.

In the processor pool model, the ration of processors to users is greater than one. The model attempts to allocate one or more processors to a user to complete the task. Once the user's task is completed the assigned processors are returned to the pool.

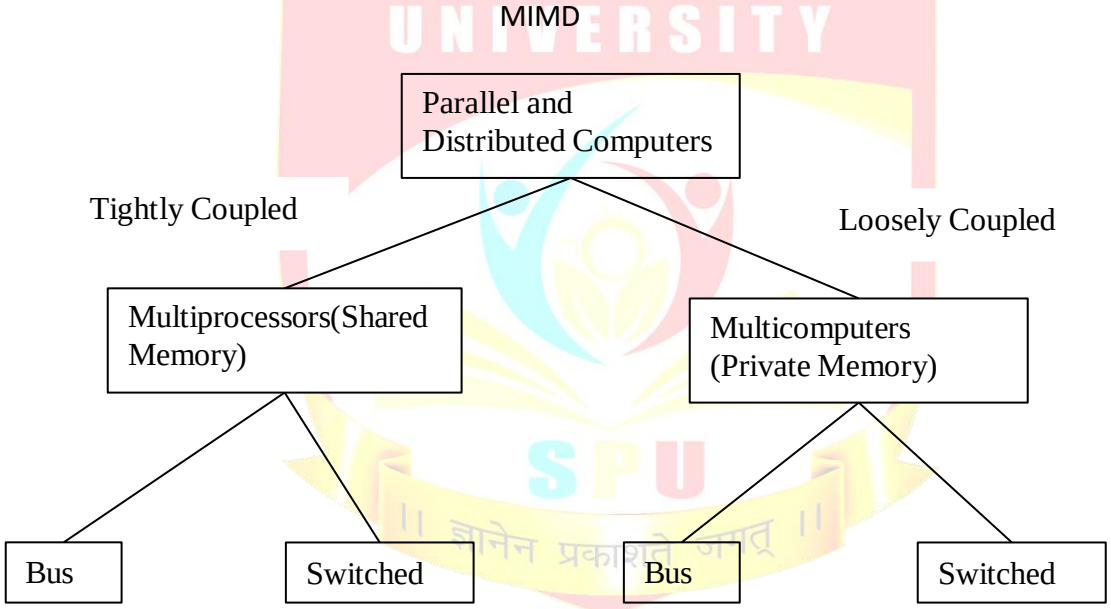
Examples: Amoeba system is combination of workstation and processor pool models. Each user performs quick interactive response type of task on the workstation (such as editing). User can access to pool of processors for executing jobs that need significant numerical computations.

The Hybrid Model

To combine the advantages of both the workstation-server and processor-pool models, a hybrid model may be used to build a distributed computing system. The hybrid model is based on the workstation-server model but with the addition of a pool of processors. The processors in the pool can be allocated dynamically for computations that are too large for workstations or that requires several computers concurrency for efficient execution. In addition to efficient execution of computation-intensive jobs, the hybrid model gives guaranteed response to interactive jobs by allowing them to be processed on local workstations of the users. However, the hybrid model is more expensive to implement than the workstation-server model or the processor-pool model.

Hardware and Software Concepts

Hardware Concept



- Although all Distributed Systems consist of multiple CPUs, there are different ways of interconnecting them and how they communicate
- Flynn (1972) identified two essential characteristics to classify multiple CPU computer systems: the number of instruction streams and the number of data streams
    - Uniprocessors SISD
    - Array processors are SIMD - processors cooperate on a single problem
    - MISD - No known computer fits this model
    - Distributed Systems are MIMD - a group of independent computers each with its own program counter, program and data

- MIMD can be split into two classifications
    - Multiprocessors - CPUs share a common memory
    - Multi computers - CPUs have separate memories
- Can be further sub classified as
    - Bus - All machines connected by single medium (e.g., LAN, bus, backplane, cable)
    - Switched - Single wire from machine to machine, with possibly different wiring patterns (e.g, Internet)
- Further classification is

    Tightly-coupled - short delay in communication between computers, high data rate (e.g., Parallel computers working on related computations)

    Loosely-coupled - Large delay in communications, Low data rate (Distributed Systems working on unrelated computations)

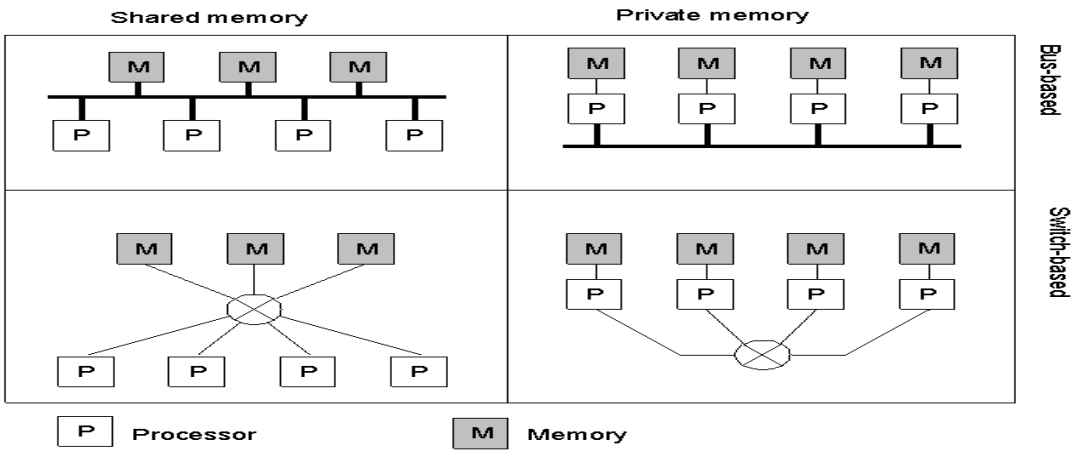
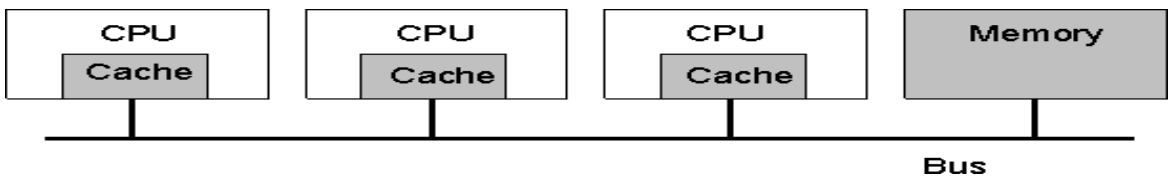
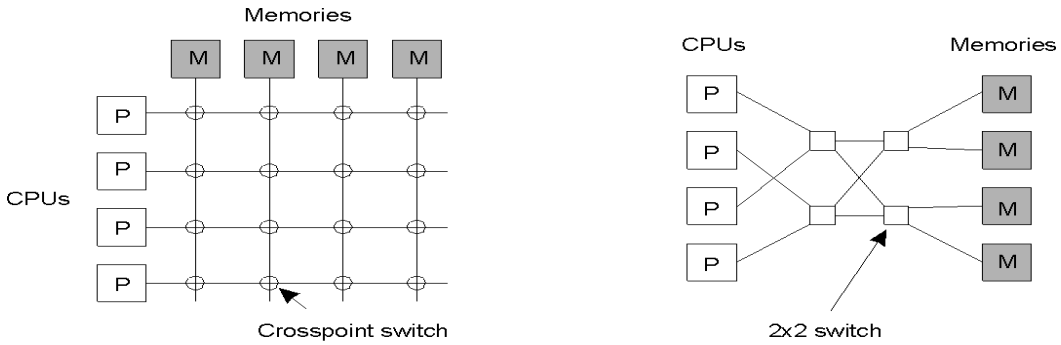


Figure 6 Hardware Concept

Bus Based Multiprocessor:

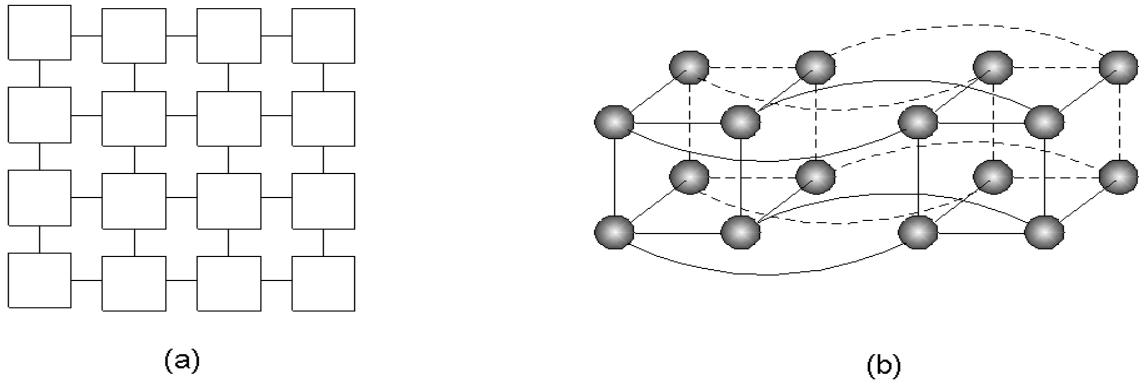


Switched Based Multiprocessor



a) A crossbar switch b) An omega switching network

Homogeneous Multicomputer System



a) Grid b) Hypercube

Software Concept:

A distributed operating system is a software over a collection of independent, networked, communicating, and physically separate computational nodes. Each individual node holds a specific software subset of the global aggregate operating system

- IPC
- Uniform Process Management
- Uniform and Visible File System
- Local Control of Machines
- Scheduling Issues

System	Description	Main Goal
DOS (Distributed Operating System)	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS (Network Operating System)	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

Issues in Distributed Operating Systems

A distributed operating system is a program that manages the resources of a computer system and provides users an easy and friendly interface to operate the system. The typical characteristics of the distributed operating systems are:

- System appears to its users as a centralized operating system, but it runs on multiple independent computers.
- Each computer may have the same or different operating system, but not visible to the users.
- User views the system as a virtual uniprocessor, and not a collection of distinct machines.
- User does not know on what computers the job was executed, on what computers the required files are stored and how the system communicates and synchronies among different computers.

Some important issues that arise in the design of a distributed operating system are:

- Unavailability of up to date Global knowledge
- Naming
- Scalability
- Compatibility

  Binary level,

  Execution level

  Protocol level
- Process Synchronization
- Resource management

  Data migration

Computational

migration Distributed

scheduling

- Security
- Structuring

Monolithic kernel

Collective kernel structure

Object-oriented operating system

- Client Server Computing Model

1. **Global Knowledge**:

Due to the unavailability of a global memory, a global clock and the unpredictability of message delays, it is practically impossible for a computer to collect up-to-date information about the global state of the distributed system. This leads to the two basic problems in designing Distributed Operating Systems

- How to determine efficient techniques to implement *decentralized system control*, where the system does not know the current and complete up to date status of the global state.
- How to order all the events that occur at different times at different computers in the system, due to the absence of a global clock

2. **Naming:**

Names are used to refer objects (e.g. computers, printers, services, files, users, etc.). a named service maps a logical name into a physical address by using a table or directory lookup The directory of all the named objects in the system must be maintained to allow proper access.

In distributed system directories may be replicated and stored at different locations to overcome a single point failure and to increase the availability of the named service. The drawbacks of replication are:

- It require more storage capacity and
- Synchronization requirement are needed when directories are updated, as directory at each location need to be updated.

Alternately directories may be partitioned to overcome the drawbacks of replicated

directories. The drawback of partitioned directories is

- In finding the partition containing the name and address of interest.

Both schemes of replicated directories and partitioned directories have their strengths and weaknesses.

3. **Scalability**

Distributed system generally grows with time. Any mechanisms or approaches adopted in designing a distributed system must not result in badly degraded performance when the system grows.

For example, broadcast based protocol works well for small distributed systems but not for large distributed systems. Consider a distributed file system that locates files by broadcasting queries. Under this file system, every computer in the distributed system is subjected to the message handling overhead. As number of users increases and distributed system gets larger, the file-request queries will increase and the overhead will grow larger.

4. **Compatibility**

The interoperability among the resources in a distributed system must be an integral part of the design of a distributed system. Three levels of compatibility exist:

- Binary level compatibility,
- Execution level compatibility and
- Protocol level compatibility.

Binary Level Compatibility

In Binary level compatibility, all processors execute the same binary instruction set, even though the processors may differ in performance and in input-output.

- Advantage of binary level compatibility is that it is easier for system development, as the code for many functions depends on the machine instructions.
- On the other hand such distributed system cannot include computers with different architectures.
- Due to this restriction binary compatibility is rarely used in large systems.

Execution Level Compatibility

Execution level compatibility is said to exist, if the same source code can be compiled and executed properly on any computer in the distributed system.

Protocol Level Compatibility

Protocol level compatibility is the least restrictive form of compatibility. It achieves interoperability by requiring all system components to support a common set of protocols.

Advantage of protocol level compatibility is that individual computers can run different operating systems. This is possible by employing common protocols for essential system services such as file access, naming and authentication.

5. **Process Synchronization**

Process synchronization is difficult in distributed systems due to the lack of shared memory and a global clock.

A distributed system has to synchronize processes running at different computers when they try to concurrently access a shared resource, such as a file directory. For correctness, it is necessary that the shared resource be accessed by a single process at a time (that is access should be atomic). The problem is known as the *mutual exclusion*.

In distributed systems, processes can request resources (local or remote) and release resources in any order. If sequence of the allocation of resources is not controlled, *deadlocks* may occur. In order to maintain system performance, deadlocks must be detected and resolved as soon as possible.

6. **Resource Management**

Resource management refers to schemes and methods devised to make local and remote resources available to users in an *effective and transparent* manner. The specific location of resources should be hidden from the users. Three general schemes exist:

- Data Migration
- Computational Migration
- Distributed Scheduling

Data Migration

In data migration, data (contents of memory or file) is brought to the location of computation that needs access to it, by the distributed operating system. If a computation updates data, the original location may have to be similarly updated. In case of physical memory access of another computer, the data request is dealt by distributed *shared memory management*, which provides a virtual address space that is shared by all the computers in a distributed system.

The major concern is the maintenance of consistency of the shared data and to minimize the access delay. In case of a file access the data request is dealt by the distributed file system, which is a component of the distributed operating system. The distributed file system provides the same functional capability to access files regardless of their location. This property of a distributed file system is known as *network transparency*.

Computational Migration

In computational migration**,** computation migrates to another location. For example,

- In distributed scheduling, one computer may require another computer status. It is more efficient and safe to find this information at the remote computer and
- Send the required information back, rather than to transfer the private data structure of the operating system at the remote computer to the requesting computer.
- The remote procedure call (RPC) mechanism is used widely for computational migration and providing communication between computers.

Distributed Scheduling

In distributed scheduling, processes are transferred from one computer to another by distributed operating system. Process relocation may be desirable if the computer where the process originated is overloaded or it does not have the necessary resources required by the process. This is done to enhance utilization of computer resources in the system.

7. **Security**

The security of the system is the responsibility of its operating system. Two issues are relevant:

- Authentication (verifying claims) and
- Authorization (deciding and authorizing the proper amount of privileges).

Authentication is the process of guaranteeing that an entity is what it claim to be, Authorization is a process of deciding what privileges an entity has and providing only these privileges.

8. **Structuring**

Structuring defines how various parts of the operating system are organized.

Unit-2

Distributed Shared Memory and Distributed File System

Basic Concepts of Distributed Shared Memory-
The two basic Paradigms for Inter process Communication-

- **Shared Memory Paradigm**
- **Message Passing Paradigm(RPC)**

- What is DSM?
  - The distributed shared memory (DSM) implements the shared memory model in distributed systems, which have no physical shared memory
  - The shared memory model provides a virtual address space shared between all nodes
  - The overcome the high cost of communication in distributed systems, DSM systems move data to the location of access
- How it works?:
  - Data moves between main memory and secondary memory (within a node) and between main memories of different nodes
  - Each data object is owned by a node
    - Initial owner is the node that created object
    - Ownership can change as object moves from node to node
  - When a process accesses data in the shared address space, the mapping manager maps shared memory address to physical memory (local or remote)

DSM Architecture:

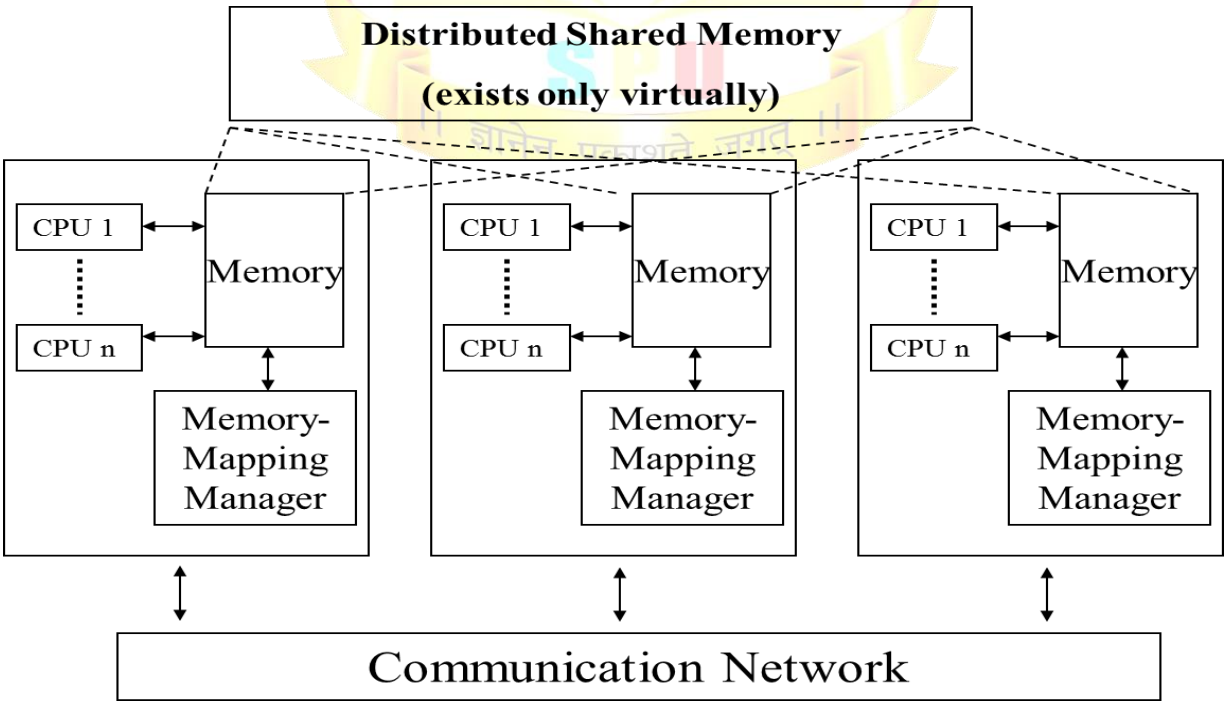


Fig 1: Distributed Shared Memory

- Distributed Shared Memory (DSM) allows programs running on separate computers to share data without the programmer having to deal with sending messages.
- Instead underlying technology will send the messages to keep the DSM consistent (or relatively consistent) between computers.
- DSM allows programs that used to operate on the same computer to be easily adapted to operate on separate computers.

Advantages of distributed shared memory (DSM)

- Data sharing is implicit, hiding data movement (as opposed to 'Send'/'Receive' in message passing model)
- Passing data structures containing pointers is easier (in message passing model data moves between different address spaces)
- Moving entire object to user takes advantage of locality difference
- Less expensive to build than tightly coupled multiprocessor system: off-the-shelf hardware, no expensive interface to shared physical memory
- Very large total physical memory for all nodes: Large programs can run more efficiently
- No serial access to common bus for shared physical memory like in multiprocessor systems
- Programs written for shared memory multiprocessors can be run on DSM systems with minimum changes

Design and Implementation Issues in DSM:

- How to keep track of the location of remote data
- How to minimize communication overhead when accessing remote data
- How to access concurrently remote data at several nodes

Some other important issues involved in the Design and Implementation of DSM-

1. Granularity: size of shared memory unit
    a. If DSM page size is a multiple of the local virtual memory (VM) management page size (supported by hardware), then DSM can be integrated with VM, i.e. use the VM page handling
    b. Advantages vs. disadvantages of using a large page size:
        i. Exploit locality of reference
        ii. Less overhead in page transport
        iii. More contention for page by many processes
    c. Advantages vs. disadvantages of using a small page size
        i. Less contention
        ii. Less false sharing (page contains two items, not shared but needed by two processes)
        iii. More page traffic
    d. Examples
        i. PLUS: page size 4 Kbytes, unit of memory access is 32-bit word
        ii. Clouds, Munin: object is unit of shared data structure

2. Structure of Shared Address Space: Structure refers to the layout of the shared data in the memory.
   The Three commonly approaches for structuring the shared memory space of a DSM system are-
   i) No Structuring

ii) Structuring by Data Type
iii) Structuring as a Data Base.

3. Memory coherence and access synchronization: The value returned by a read is the same as the value written by the most recent write
4. Data Location and Access: To share data in a DSM system, it should be possible to locate and retrieve the data accessed by a user process.
5. Replacement Strategy: If the local memory of a node is full, a cache miss at that node implies not only a fetch of the accessed data block from a remote node but also a replacement.
6. Thrashing: Thrashing occurs when network resources are exhausted, and more time is spent invalidating data and sending updates than is used doing actual work.
   Based on system specifics, one should choose write-update or write-invalidate to avoid thrashing
7. Heterogeneity: The DSM system must be designed to take care of heterogeneity so that it functions properly with machines having different architectures.

Consistency Model:

A consistency model is the definition of when modifications to data may be seen at a given processor. It defines how memory will appear to a programmer by placing restrictions on the values that can be returned by a read of a memory location. A consistency model must be well understood. It determines how a programmer reasons about the correctness of programs and determines what hardware and compiler optimizations may take place.

There are various types of Consistency Model-

- Linearizability (also known as strict or atomic consistency)
- Sequential Consistency
- Casusal Consistency
- Release Consistency
- Eventual Consistency
- Delta consistency
- PRAM consistency (also known as FIFO consistency)
- Weak Consistency

Linearizability:

In concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition — they either successfully change the state of the system, or have no visible effect.

Atomicity is commonly enforced by mutual exclusion, whether at the hardware level building on a cache coherency protocol, or the software level using semaphores or locks. Thus, an atomic operation does not actually occur instantaneously. The benefit comes from the appearance: the system behaves as if each operation occurred instantly, separated by pauses.

Because of this, implementation details may be ignored by the user, except insofar as they affect performance. If an operation is not atomic, the user will also need to understand and cope with sporadic extraneous behavior caused by interactions between concurrent operations, which by its nature is likely to be hard to reproduce and debug.

Sequential consistency

Sequential consistency is one of the consistency models used in the domain of concurrent programming (e.g. in distributed shared memory, distributed transactions, etc.). It was first defined as the property that requires that "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

The system provides sequential consistency if every node of the system sees the (write) operations on the same memory part (page, virtual object, cell, etc.) in the same order, although the order may be different from the order as defined by real time (as observed by hypothetical external observatory or global clock) of issuing the operations. The sequential consistency is weaker than strict consistency (which would demand that operations are seen in order in which they were actually issued, which is essentially impossible to secure in distributed system as deciding global time is impossible) and is the easiest consistency model to understand, since a system preserving that model is behaving in a way expected by an instantaneous system.

The simplicity is achieved at cost of efficiency: distributed systems with sequential consistency model are, without further optimization such as speculation, one magnitude slower than those providing weaker models such as causal consistency.

Causal consistency

Causal consistency is one of the consistency models used in the domain of the concurrent programming (e.g. in distributed shared memory, distributed transactions etc). A system provides causal consistency if memory operations that potentially are causally related are seen by every node of the system in the same order. Concurrent writes (i.e. ones that are not causally related) may be seen in different order by different nodes. This is weaker than sequential consistency, which requires that all nodes see all writes in the same order, but is stronger than PRAM consistency, which requires only writes done by a single node to be seen in the same order from every other node.

When a node performs a read followed later by a write, even on a different variable, the first operation is said to be causally ordered before the second, because the value stored by the write may have been dependent upon the result of the read. Similarly, a read operation is causally ordered after the earlier write on the same variable that stored the data retrieved by the read. Also, even two write operations performed by the same node are defined to be causally ordered, in the order they were performed. Intuitively, after writing value v into variable x, a node knows that a read of x would give v, so a later write could be said to be (potentially) causally related to the earlier one. Finally, we force this causal order to be transitive: that is, we say that if operation A is (causally) ordered before B, and B is ordered before C, A is ordered before C.

Release consistency

Release consistency is one of the consistency models used in the domain of the concurrent programming (e.g. in distributed shared memory, distributed transactions etc.). Systems of this kind are characterized by the existence of two special synchronization operations, release and acquire. Before issuing a write to a memory object a node must acquire the object via a special operation, and later release it. Therefore the application that runs within the operation acquire and release constitutes the critical region. The system is said to

provide release consistency, if all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter acquire it.

There are two kinds of coherence protocols that implement release consistency:

- eager, where all coherence actions are performed on release operations, and
- lazy, where all coherence actions are delayed until after a subsequent acquire

Eventual consistency

Eventual consistency is one of the consistency models used in the domain of parallel programming, for example in distributed shared memory, distributed transactions, and optimistic replication.

The term itself suggests the following definition: Given a sufficiently long period of time, over which no updates are sent, we can expect that during this period, all updates will, eventually, propagate through the system and all the replicas will be consistent. While some authors use that definition (e.g., the Vogels citation above), others prefer a stronger definition that requires good things to happen even in the presence of continuing updates, reconfigurations, or failures. In the Terry et. al. work referenced above, eventual consistency means that for a given accepted update and a given replica eventually either the update reaches the replica or the replica retires from service.

Delta consistency

Delta consistency is one of the consistency models used in the domain of parallel programming, for example in distributed shared memory, distributed transactions, and Optimistic replication

The delta consistency model states that an update will propagate through the system and all replicas will be consistent after a fixed time period δ.

PRAM consistency

PRAM consistency (pipelined random access memory) also known as FIFO consistency, or Processor consistency. All processes see memory writes from one process in the order they were issued from the process.

Writes from different processes may be seen in a different order on different processes. Only the write order needs to be consistent, thus the name pipelined.

Weak consistency

The name weak consistency may be used in two senses. In the first sense, strict and more popular, the weak consistency is one of the consistency models used in the domain of the concurrent programming (e.g. in distributed shared memory, distributed transactions etc.). The protocol is said to support weak consistency if:

1. All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
2. All other accesses may be seen in different order on different processes (or nodes, processors).

3. The set of both read and write operations in between different synchronization operations is the same in each process.

Therefore, there can be no access to synchronization variable if there are pending write operations. And there can-not be any new read/write operation started if system is performing any synchronization operation. In the second, more general, sense weak consistency may be applied to any consistency model weaker than sequential consistency.

Thrashing:

Thrashing is said to occur when the system spends a large amount of time transferring shared data blocks from one node to another, compared to the time spent doing the useful work of executing application processes.

The following methods may be used to solve the thrashing problem in DSM system-

1) Providing Application Control Lock- Locking data to prevent other nodes from accessing that data for a short period of time can reduce thrashing.
2) Nailing a block to a node for a minimum amount of time-Another method to reduce thrashing is to disallow a block to be taken away from a node until a minimum amount time t is elapse after its allocation to that node. Time t can either be fixed statically or be tuned dynamically on the basis of access pattern.
3) Tailoring the coherence algorithm to the shared data usage pattern- Thrashing can also be minimized by using different coherence protocols for shared data having different characteristics.

Distributed File System:

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.

Two main Purposes of Using a File:

1) Permanent Storage of Information
2) Sharing of Information

<u>Distributed File System Supports-</u>

- Remote Information Sharing:- A DFS allows a file to be transparently accessed by processes of any node of the system irrespective of the file's location.
- User Mobility: It implies that a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different time.
- Availability: For better fault tolerance files should be available for use even in the failure of one or more node. File system normally keeps multiple copies of files it is called replica.
- Diskless WorkStation: A distributed file system with its transparent remote file accessing capabilities allows the use of Diskless workstations in a system.

Services Provided by the Distributed File System-

1. **Storage Service:**
   - It deals with the allocation and management of space on a secondary storage device.
   - The storage service is also known as Disk service.
   - Several systems allocate disk space in units of fixed size blocks and hence the storage service is also known as block service.
2. **True File Service:**

It is concerned with the operations on individual files, such as accessing, modification, creating and deletion of files.

3. **Name Service:**

It provides a mapping between text names for files and references to files that is FILE ID. Most File System use directories to perform this mapping, so Name Service is also known as Directory Service.

Desirable Features of a Good Distributed File System

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write). The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently used data and issuing disk I/O requests.

A distributed file system is to present certain degrees of transparency to the user and the system:

Transparency:

Four types of Transparencies-

1. Structure Transparency- distributed file system normally uses multiple file server. Client should not know the number or locations of the file server and the storage devices. Ideally a distributed file system should look to its clients like a conventional file system offered by a centralized time sharing operating system.
2. Access Transparency: Both local and remote files should be accessible in the same way.
3. Naming Transparency: The name of a file should give no hint as to where the file is located.
4. Replication Transparency: If a file is replicated on multiple nodes both the existence of multiple copies and their locations should be hidden from the user.

Heterogeneity: File service should be provided across different hardware and operating system platforms.

Scalability: The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).

Replication transparency: To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.

Migration transparency: Files should be able to move around without the client's knowledge. Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them.

Tolerance for network partitioning: The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

Note:
• **Storage service**
– Disk service: giving a transparent view of distributed disks.
– Block service giving the same logical view of disk-accessing units.
• **True file service**
– File-accessing mechanism: deciding a place to manage remote files and unit to transfer data (at server or client? file, block or byte?)
– File-sharing semantics: providing similar to Unix but weaker file update semantics
– File-caching mechanism: improving performance/scalability
– File-replication mechanism: improving performance/availability
• **Name service**
– Mapping between text file names and reference to files, (i.e. file IDs)
– Directory service

File Models:

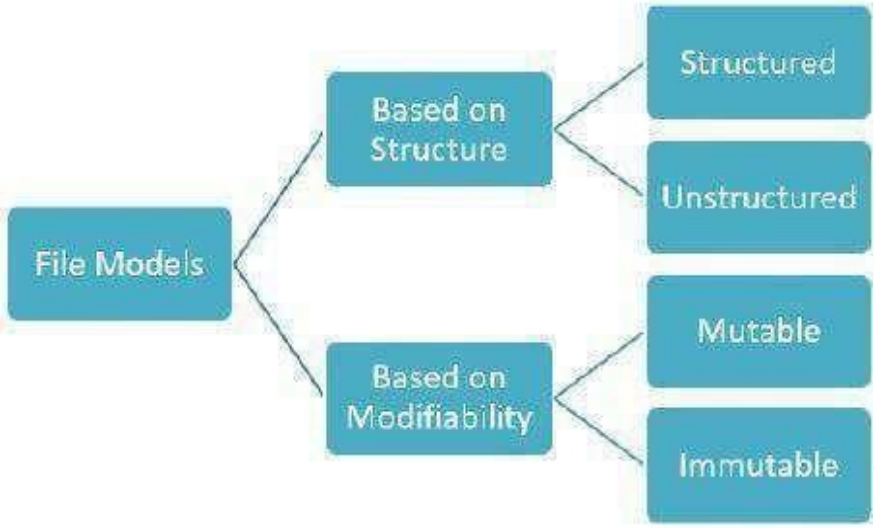


Fig 2. Types of File Model

Unstructured files

- In this model, there is no substructure known to the file server.
- Contents of each file of the file system appear to the file server as an un interpreted sequence of bytes.
- Interpretation of the meaning and structure of the data stored in the files are entirely up to the application programs.
- UNIX, MS-DOS and other modern operating systems use this file model.

This is mainly because sharing of a file by different applications is easier compared to the structured file model.

Structured files

- In structured files (rarely used now) a file appears to the file server as an ordered sequence of records.
- Also, Records of different files of the same file system can be of different sizes.

❑ **Files with a non-indexed record**

- File record accessed by specifying its position within the file.
- Also, For example, the fifth record from the beginning of the file or the second record from the end of the file.

❑ **Files with indexed records**

- Records have one or more key fields and can be addressed by specifying the values of the key fields.
- The file maintained B-tree or other suitable data structure or hash table to locate records quickly.

Mutable and Immutable files

Mutable Files

- Update performed on a file overwrites on its old contents to produce the new contents.
- Moreover, A file can modify by each update operation.
- Also, File overwrites its old contents to produce the new contents.
- Most existing operating systems use the mutable file model.(MSDOS,UNIX)

Immutable Files

- Rather than updating the same file, a new the version of the file created each time a change made to the file contents.
- A file cannot modify once it has created except to deleted.
- Moreover, File versioning approach used to implement file updates.
- It rarely used now a day.(Cedar File System)

File Accessing Models

The client's request to access a file is serviced depends on the file accessing model used by the file system.

The file accessing model of a distributed file system mainly depends on two factors:

- **The method used for accessing remote files**
- **The unit of data access**

Accessing Remote Files:- A distributed file system may use one of the following models to service client's file access request.

Remote service model

- The client's request for file access delivered to the server, the server machine performs the access request, and finally, the result is forwarded back to the client.
- The access requests from the client and the server replies for the client transferred across the network as messages.
- The file server interface and the communication protocols must design carefully to minimize the overhead of generating messages as well as the number of messages that must exchange in order to satisfy a particular request.

Data caching model

- If the data needed to satisfy the client's access request not present locally, it copied from the server's node to the client's node and cached there.
- The client's request is processed on the client's node itself by using the cached data.
- Thus repeated accesses to the same data can handle locally.
- A replacement strategy LRU used for cache management.

Unit of Data Transfer:-Unit of data refers to the fraction of file data that transferred to and from clients as a result of a single read or write operations.

File-level transfer model:-The whole File Accessing Models treated as the unit of data transfer between client and server in both the direction.

Advantages of File level Transfer Models

- Transmitting an entire File Accessing Models in response to a single request is more efficient than transmitting it page by page as the network protocol overhead required only once.
- It has better scalability because it requires fewer accesses to file servers, resulting in reduced server load and network traffic.
- Disk access routines on the servers can better optimize if it known that requests are always for entire files rather than for random disk blocks.
- Once an entire file cached at a client's site, it becomes immune to the server and network failures.

Disadvantage of File level Transfer Models

- This model requires sufficient storage space on the client's node for storing all the requires files in their entirety.

Block level transfer model

- In this model, file data transfer across the network between a client and a server take place in units of file blocks.
- Moreover, A file blocks a contiguous portion of a file and usually fixed in length.
- In page level transfer model block size is equal to virtual memory page size.

Advantages

- This model does not require client nodes to have large storage.
- Also, It eliminates the need to copy an entire file when only a small portion of the file data needed.
- It provides large virtual memory for client nodes that do not have their own secondary storage devices.

Disadvantage

- When an entire file is to accessed, multiple server requests needed in this model, results in more network traffic and more network protocol overhead

Byte-level transfer model

- In this model, file data transfers across the network between a client and a server take pace in units of bytes.

Advantages

- This model provides maximum provides maximum flexibility. Because it allows storage and retrieval of an arbitrary sequential sub range of a file, specified by an offset within a file, and a length.

Disadvantage

- Cache management is difficult due to variable length data for different requests.

Record level transfer model

- In this model, file data transfers across the network between a client and a server take place in units of records.

File sharing semantics:

- A shared file may be simultaneously accessed by multiple users.

- In such a situation, an important design issue for any File Sharing Semantics system is to clearly define when modifications of file data made by a user are observable by other users.

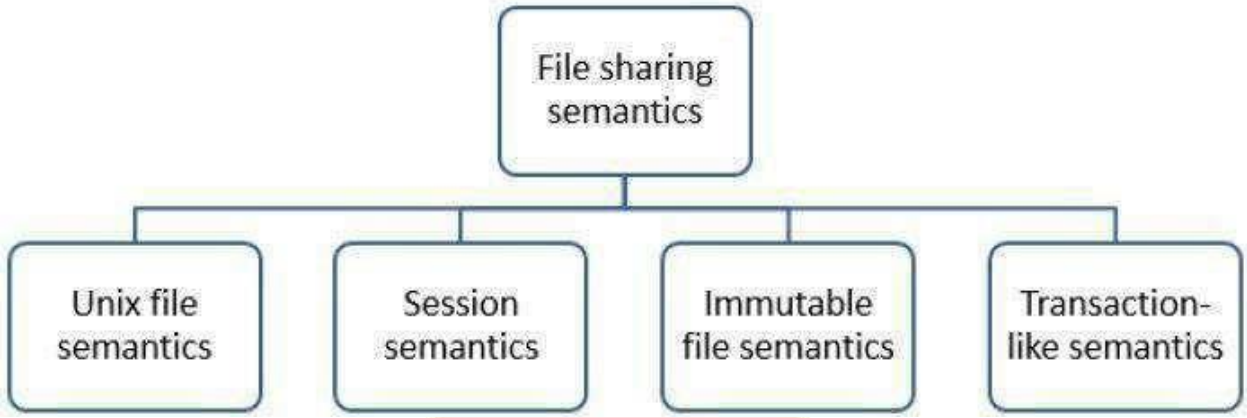


Fig. 3 File Sharing Semantics

Unix semantics

- This File Sharing Semantics enforces an absolute time ordering on all operations.
- Every read operation on a file sees the effects of all previous write operations performed on that file.
- This semantics can achieve in a distributed system by disallowing files to cached at client nodes.
- Allowing a shared file to managed by only one file server that processes all read and write requests for the file strictly in the order in which it receives them.

5. There a possibility that, due to network delays, client requests from different nodes may arrive and get processed at the server node out of order.

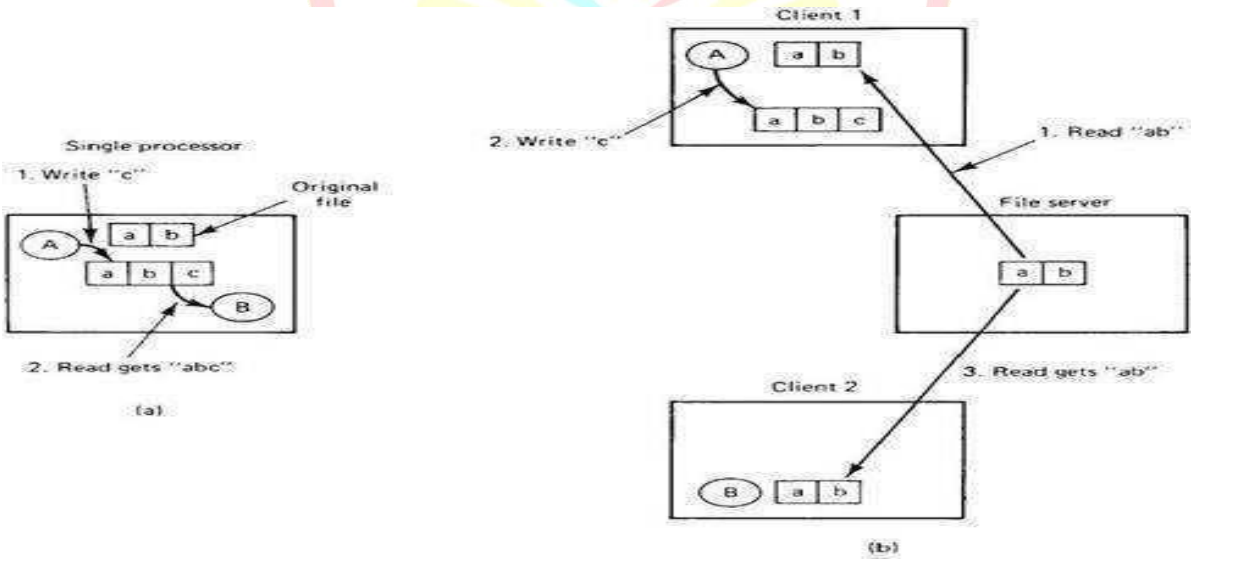


Fig 3. (a) On a single processor, when a READ follows a WRITE, the value returned by the read is the value just written. (b) In a distributed system with caching, obsolete values may be returned.

Session semantic

- A client opens a file, performs a series of reading/write operations on the file. And finally closes the file when he or she did with the file.
- A session a series of file accesses made between the open and close operations.
- In session semantics, all changes made to a file during a session initially made visible only to the client process that opened the session and is invisible to other remote processes who have the same file open simultaneously.
- Once the session closed, the changes made to the file made visible to remote processes only in later starting sessions.

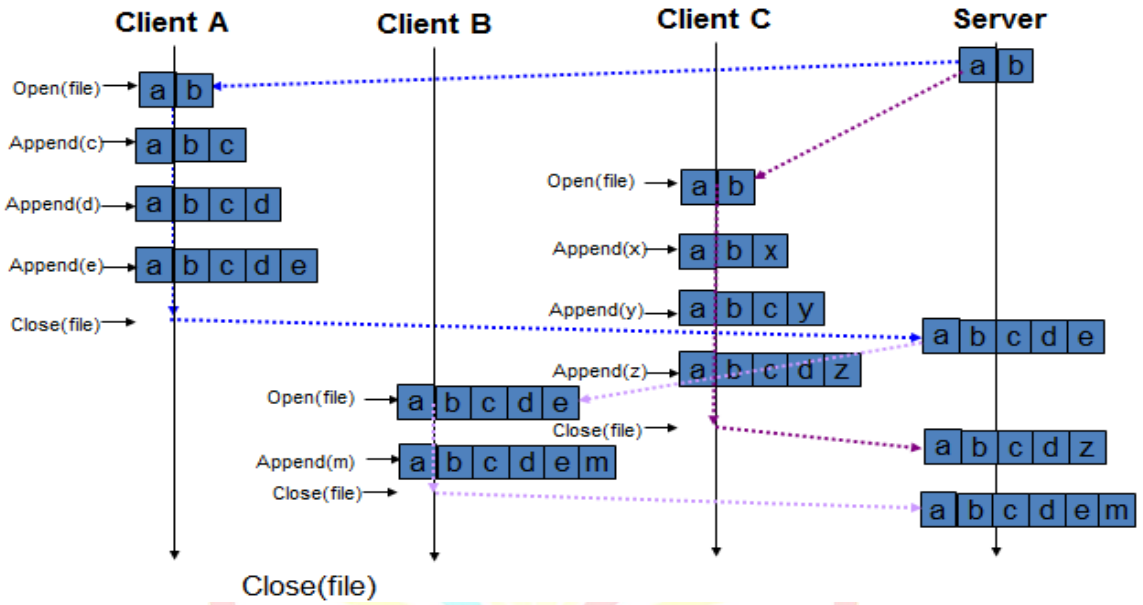


Fig 4 Session Semantics

Immutable shared file semantics

- According to this semantics, once the creator of a file declares it to Shareable. The file treated as immutable so that it cannot modify anymore.
- Change to the file handled by creating a new updated version of the file.
- Therefore, the semantics allows files to shared only in the read-only mode.

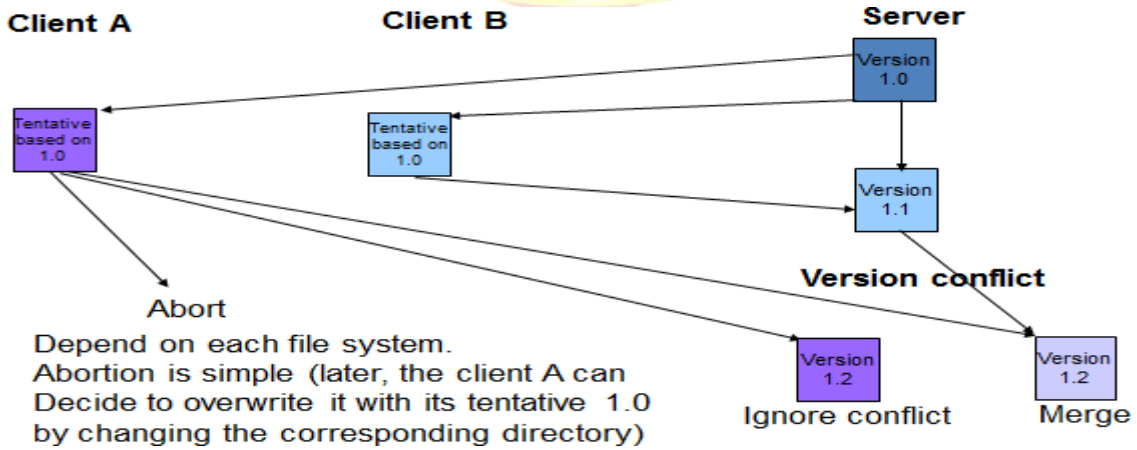


Fig 5 Immutable Semantics

Transaction like File Sharing Semantics

- This semantic based on the transaction mechanism, which is a high-level mechanism for controlling concurrent access to shared mutable data.
- A transaction a set of operations enclosed in-between a pair of begin transaction and end transaction like operations.
- The transaction mechanism ensures that the partial modifications made to the shared data by a transaction will not be visible to other concurrently executing transactions until the transaction ends.

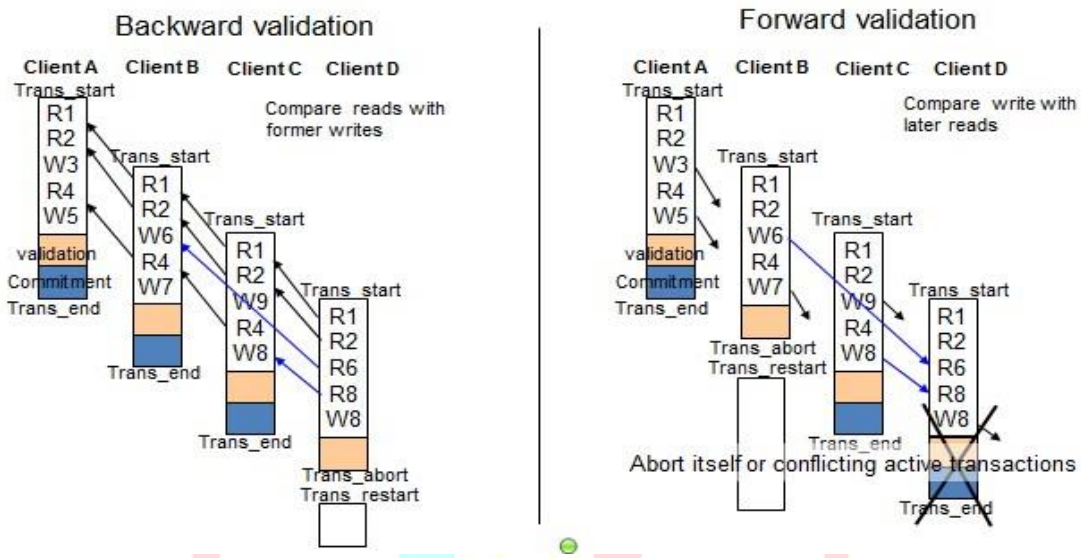
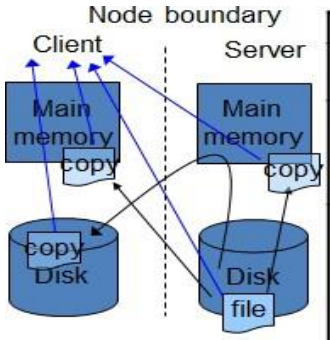


Fig 6 Transaction Like Semantics

File Caching Scheme-

- Retaining most recently accessed disk blocks.
- Repeated accesses to a block in cache can be handled without involving the disk.
- Advantages
  - Reduce delays
  - Reduce contention for disk arm



Location	Merits	Demerits
No caching	No modifications	Frequent disk access, Busy network traffic
In server's main memory	One-time disk access, Easy implementation, Unix-like file-sharing semantics	Busy network traffic
In client's disk	One-time network access, No size restriction	Cache consistency problem, File access semantics, Frequent disk access, No Diskless workstation
In client's main memory	Maximum performance, Diskless workstation, Scalability	Size restriction, Cache consistency problem, File access semantics

Fig 7 File Caching Scheme

When design a DFS, we need to consider these stuff to make decision. We will use some practical systems to show how they solve these. We will see, make different decision based on different assumptions and goals.

- Cache location (disk vs. memory)
- Cache Placement (client vs. server)
- Cache structure (block vs. file)
- Stateful vs. Stateless server
- Cache update policies
- Consistency
- Client-driven vs. Server-driven protocols

There are Three Design Issues:

- Cache Location
- Modification Propagation
- Cache Validation

Cache Location Disk vs. Main Memory

Advantages of disk caches

- More Reliable
- Cached data are still there during recovery and don't need to be fetched again

Advantages of main-memory caches:

- Permit workstations to be diskless
- More quick access
- Server caches(used to speed up disk I/O) are always in main memory; using main-memory caches on the clients permits a single caching mechanism for servers and users

Modification Propagation

- Write-through
  Write-through – all writes be propagated to stable storage immediately
  - Reliable, but poor performance
- Delayed-write
  Delayed-write – modification written to cache and then written through to server later
- Write-on-close (variation of delayed-write)
  Write-on-close – modification written back to server when file close
  - Reduces intermediate read and write traffic while file is open

Cache Validation

- Pros for delayed-write/write-on-close
  - Lots of files have lifetimes of less than 30s
  - Redundant writes are absorbed
  - Lots of small writes can be batched into larger writes
- Disadvantage:
  - Poor reliability; unwritten data may be lost when client crash

File-Caching Schemes (Modification Propagation)-

- Write-through scheme
  - Pros: Unix-like semantics and high reliability
  - Cons: Poor write performance

- Delayed-write scheme
  - Write on cache displacement
  - Periodic write
  - Write on close
  - Pros:
    - Write accesses complete quickly
    - Some writes may be omitted by the following writes.
    - Gathering all writes mitigates network overhead.
  - Cons:
    - Delaying of write propagation results in fuzzier file-sharing semantics.

File-Caching Schemes (Cache Validation Schemes – Client-Initiated Approach)

- Checking before every access (Unix-like semantics but too slow)
- Checking periodically (better performance but fuzzy file-sharing semantics)
- Checking on file open (simple, suitable for session-semantics)
- Problem: High network traffic

File-Caching Schemes (Cache Validation Schemes – Server-Initiated Approach)

- Keeping track of clients having a copy
- Denying a new request, queuing it, and disabling caching
- Notifying all clients of any update on the original file
- Problem:
  - violating client-server model
  - Stateful servers
  - Check-on-open still needed for the 2nd file opening.

Fault Tolerance:-

- System fails when it does not match its promises. An error in a system can lead to a failure. The cause of an error is called a fault.
- Faults are generally transient, intermittent or permanent. Transient occurs once and disappears, intermittent fault keep reoccurring and permanent faults continue till system is repaired.
- Being fault tolerant is related to dependable systems. The dependability includes availability, reliability, safety and maintainability.
- The ability of a system to continue functioning in the event of partial failure is known as fault tolerance.
- It is an important goal in distributed system design to construct a system that can automatically recover from partial failures without affecting overall performance.

Naming:-

A good naming system for a distributed system should have the features –

Location Transparency

- This feature implies that the name of the object must not indicate the physical location of the object, directly or indirectly.
- The name should not depend on physical connectivity, a topology of the system, or even the current location of the object.

Location Independence

- Any distributed system allows object migration, movement, and relocation of objects dynamically among the distributed nodes.
- Location independence means that the name of the object need not be changed when the object's location is changed.
- The user must be able to access the object irrespective of the node from where it is being accessed.

Scalability

- The naming system should be capable of adapting to the dynamically-changing scale, leading to dynamic changes in a namespace.
- Any changes in system scale should not require any change in naming or location mechanisms.

Uniform Naming Convention

- Most of the distributed systems use some standard naming conventions for naming different types of objects.

File names named differently from usernames and process names.

- Ideally, a good Naming Distributed system must use the same naming mechanism for all types of objects, so as to reduce the complexity of designing.

Meaningful names

- Meaningful names indicate what the object actually contains.
- A good naming system must support at least two levels of object identifiers; one which is convenient for users and the other which is relevant for machines.

Allow multiple user-defined names for the same object

- Users must be allowed to use their own names for accessing the object.
- A naming system must have the flexibility of assigning multiple user-defined names to the same object.
- It should be possible for the user to either change or delete the user-defined name for the object without affecting the names given by other users to the same object.

Group Naming

- The broadcast facility used to communicate with groups of nodes, which may not comprise the entire distributed system.
- A good naming system must allow many different objects to identified by the same name.

Performance

- One important aspect of performance of a naming system the time required to map an object's name to its attributes.
- Moreover, This depends on the number of messages transmitted to and fro during the mapping operation.
- The Naming Distributed system should use a small number of message exchanges.

Fault tolerance

- The naming system must be capable of tolerating faults such as the failure of a node or network congestion.
- The naming system should continue functioning, maybe poorly, in case of such failures, but should still be operational.

Replication transparency

- Replicas generally created in a distributed system to improve system performance and reliability.
- Also, A good naming system should support multiple copies of the same object in a user transparent manner.

Locating the nearest replica

- When a naming system supports replicas, the object location mechanism should locate the nearest replica of the requested object.
- Locating all replicas
- From a reliability and consistency point of view, the object-locating mechanism must be able to locate all replicas.

Basic Concepts of Naming:-

Name: –String of bits or characters
–Refers to an entity
Entity: –Resource, process, user, etc.
–Operations performed on entities
–Operations performed at access points
–Access point named by an address
–Entity address = address of entity's access point
–Entity can have multiple access points
–Entity's access points may change

System Names Versus Human Names -

Related to the purity of names is the distinction between system-oriented and human-oriented names. Human-oriented names are usually chosen for their mnemonic value, whereas system oriented names are a means for efficient access to, and identification of, objects. Taking into account the desire for transparency human-oriented names would ideally be pure. In contrast, system-oriented names are often non pure which speeds up access to repeatedly used object attributes. We can characterize these two kinds of names as follows:

• **System-oriented names** are usually fixed size numerals (or a collection thereof); thus they are easy to store, compare, and manipulate, but difficult for the user to remember.
• **Human-oriented names** are usually variable-length strings, often with structure; thus they are easy for humans to remember, but expensive to process by machines.

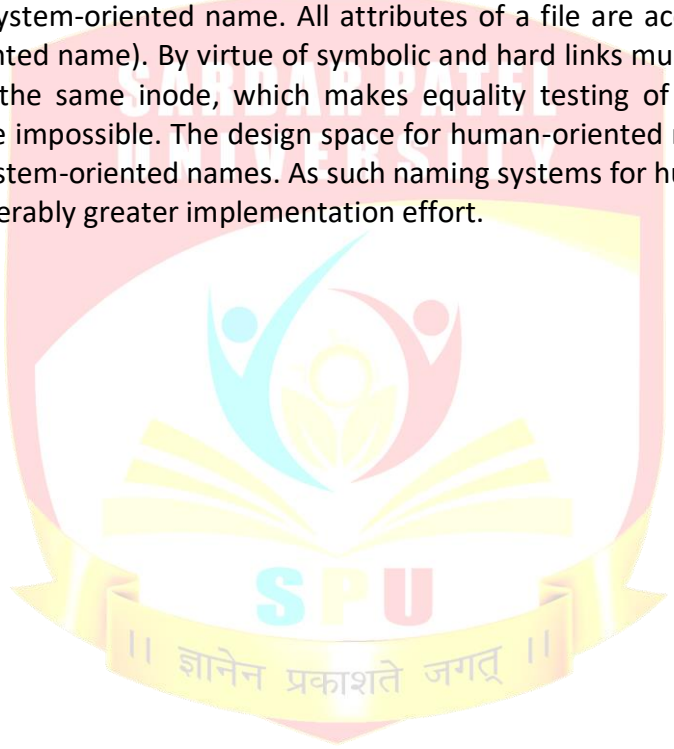
System-Oriented Names-

As mentioned, system-oriented names are usually implemented as one or more fixed-sized numerals to facilitate efficient handling. Moreover, they typically need to be unique identifiers and may be sparse to convey access rights (e.g., capabilities). Depending on whether they are globally or locally unique, we also call them structured or unstructured.
These are two examples of how structured and unstructured names may be implemented:
Globally unique integer unstructured
Node identifier + local unique identifier structured
The structuring may be over multiple levels. Note that a structured name is not pure.
Global uniqueness without further mechanism requires a centralized generator with the usual drawbacks regarding scalability and reliability. In contrast, distributed generation without excessive communication usually leads to structured names.

For example, a globally unique structured name can be constructed by combining the local time with a locally unique identifier. Both values can be generated locally and do not require any communication.

Human-Oriented Names-

In many systems, the most important attribute bound to a human-oriented name is the system oriented name of the object. All further information about the entity is obtained via the system oriented name. This enables the system to perform the usually costly resolution of the human oriented name just once and implement all further operations on the basis of the system-oriented name (which is more efficient to handle). Often a whole set of human-oriented names is mapped to a single system-oriented name (symbolic links, relative addressing, and so on). As an example of all this, consider the naming of files in Unix. A pathname is a human-oriented name that, by means of the directory structure of the file system, can be resolved to an inode number, which is a system-oriented name. All attributes of a file are accessible via the inode (i.e., the system- oriented name). By virtue of symbolic and hard links multiple human-oriented names may refer to the same inode, which makes equality testing of files merely by their human-oriented name impossible. The design space for human-oriented names is considerably wider than that for system-oriented names. As such naming systems for human-oriented names usually require considerably greater implementation effort.



Unit-3

Inter Process Communication and Synchronization

The application program interface to UDP provides a message passing abstraction the simplest form of inter process communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called datagrams. In the Java and UNIX APIs, the sender specifies the destination using a socket an indirect reference to a particular port used by the destination process at a destination computer. The application program interface to TCP provides the abstraction of a two-way stream between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Streams provide a building block for producer-consumer communication. A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them. The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

API for Internet Protocol-

- The java API for inter process communication in the internet provides both datagram and stream communication.
- The two communication patterns that are most commonly used in distributed programs:
  - **Client-Server communication:-**The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).
- **Group communication:-**The same message is sent to several processes.

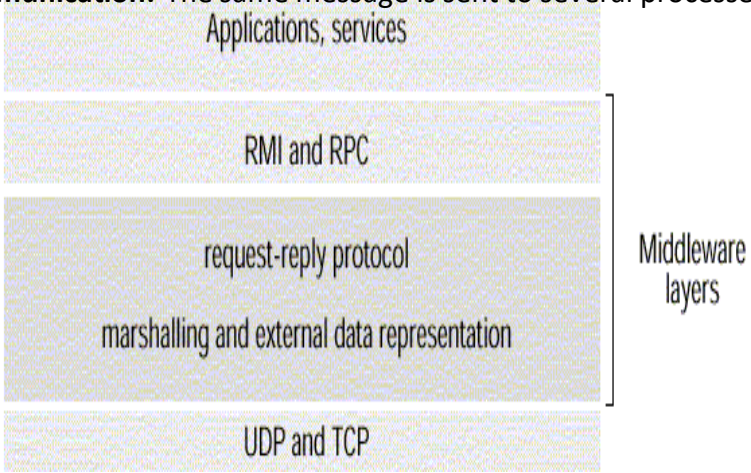


Fig.1 Middleware

- **Remote Method Invocation (RMI)**
  - It allows an object to invoke a method in an object in a remote process.
    - E.g. CORBA and Java RMI
- **Remote Procedure Call (RPC)**
  - It allows a client to call a procedure in a remote server.
- The application program interface (API) to UDP provides a message passing abstraction.
  - Message passing is the simplest form of inter process communication.

- API enables a sending process to transmit a single message to a receiving process.
- The independent packets containing the messages are called datagrams.
- In the Java and UNIX APIs, the sender specifies the destination using a socket.
- Socket is an indirect reference to a particular port used by the destination process at a destination computer.

- The application program interface (API) to TCP provides the abstraction of a two-way stream between pairs of processes.
- The information communicated consists of a stream of data items with no message boundaries.
- Request-reply protocols are designed to support client-server communication in the form of either RMI or RPC.
- Group multicast protocols are designed to support group communication.
- Group multicast is a form of inter process communication in which one process in a group of processes transmits the same message to all members of the group.

The API for the Internet Protocols

- **The CHARACTERISTICS of INTERPROCESS COMMUNICATION**

Synchronous and asynchronous communication

- In the synchronous form, both send and receive are blocking operations.
- In the asynchronous form, the use of the send operation is non-blocking and the
- receive operation can have blocking and non-blocking variants.

Message destinations

- A local port is a message destination within a computer, specified as an integer.
- A port has an exactly one receiver but can have many

senders. Reliability

- A reliable communication is defined in terms of validity and integrity.
- A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

Ordering

- Some applications require that messages be delivered in sender order.

- **SOCKET**
  - Internet IPC mechanism of Unix and other operating systems (BSD Unix, Solaris, Linux, Windows NT, Macintosh OS)
  - Processes in the above OS can send and receive messages via a socket.
  - Sockets need to be bound to a port number and an internet address in order to send and receive messages.
  - Each socket has a transport protocol (TCP or UDP).
  - Messages sent to some internet address and port number can only be received by a process using a socket that is bound to this address and port number.
  - Processes cannot share ports (exception: TCP multicast).

- Both forms of communication, UDP and TCP, use the socket abstraction, which provides and endpoint for communication between processes.

- Inter process communication consists of transmitting a message between a socket in one process and a socket in another process.

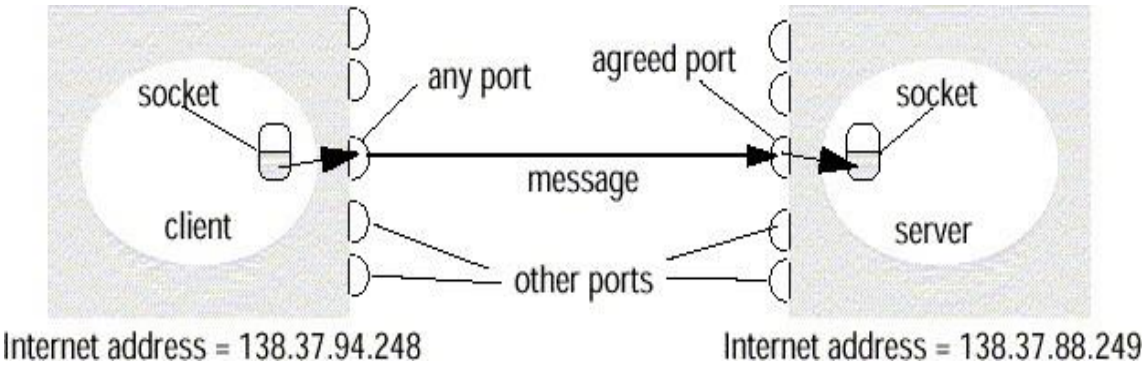


Fig 2: **Sockets and ports**

- **UDP DATAGRAM COMMUNICATION**

UDP datagram properties

- No guarantee of order preservation
- Message loss and duplications are

possible Necessary steps

- Creating a socket
- Binding a socket to a port and local Internet address
  - A client binds to any free local port
  - A server binds to a server port

Receive method

- It returns Internet address and port of sender, plus message.

Issues related to datagram communications are: Message size

- IP allows for messages of up to 216 bytes.
- Most implementations restrict this to around 8 kbytes.
- Any application requiring messages larger than the maximum must fragment.
- If arriving message is too big for array allocated to receive message content, truncation occurs.

Blocking

- Send: non-blocking

  upon arrival, message is placed in a queue for the socket that is bound to the destination port.
- Receive: blocking

  Pre-emption by timeout possible

  If process wishes to continue while waiting for packet, use separate thread

The Java API provides datagram communication by two classes:

- Datagram Packet
  - It provides a constructor to make an array of bytes comprising:
    - Message content
    - Length of message
    - Internet address
    - Local port number

- ❖ It provides another similar constructor for receiving a message.

➢ Datagram Socket
- ❖ This class supports sockets for sending and receiving UDP datagram.
- ❖ It provides a constructor with port number as argument.
- ❖ No-argument constructor is used to choose a free local port.
- ❖ Datagram Socket methods are:
  - send and receive
  - set So Timeout
  - connect

- **TCP STREAM COMMUNICATION**
  The API to the TCP protocol provides the abstraction of a stream of bytes to be written to or read from.
  - Characteristics of the stream abstraction:
    - Message sizes
    - Lost messages
    - Flow control
    - Message destinations

Use of TCP
- Many services that run over TCP connections, with reserved port number are:
  - HTTP (Hypertext Transfer Protocol)
  - FTP (File Transfer Protocol)
  - Telnet
  - SMTP (Simple Mail Transfer Protocol)

Java API for TCP streams

The Java interface to TCP streams is provided in the classes:
- **Server Socket**
  - It is used by a server to create a socket at server port to listen for connect requests from clients.
- **Socket**
  - It is used by a pair of processes with a connection.
  - The client uses a constructor to create a socket and connect it to the remote host and port of a server.
  - It provides methods for accessing input and output streams associated with a socket.

External Data Representation
- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.
- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.

Marshalling
- Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

Unmarshalling
- Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.

Note:

- **Marshalling and unmarshalling activities is usually performed automatically by middleware layer.**
- **Marshalling is likely error-prone if carried out by hand.**

Two approaches to external data representation and marshalling are:

- **CORBA Common Data Representation (CDR)**
  - CORBA CDR is the external data representation defined with CORBA 2.0.
  - It consists 15 primitive types:
    - Short (16 bit)
    - Long (32 bit)
    - Unsigned short
    - Unsigned long
    - Float(32 bit)
    - Double(64 bit)
    - Char
    - Boolean(TRUE,FALSE)
    - Octet(8 bit)
    - Any(can represent any basic or constructed type)

Composite types are (CORBA CDR for constructed types)-

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Message in CORBA CDR that contains the three fields of a struct whose respective types are string, string, and unsigned long.

For Example: struct with value {„Smith", „London", 1934}

index in sequence of bytes	←4 bytes→	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1934	*unsigned long*

- **Java's object serialization**
  - In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.
  - An object is an instance of a Java class.

Group Communication

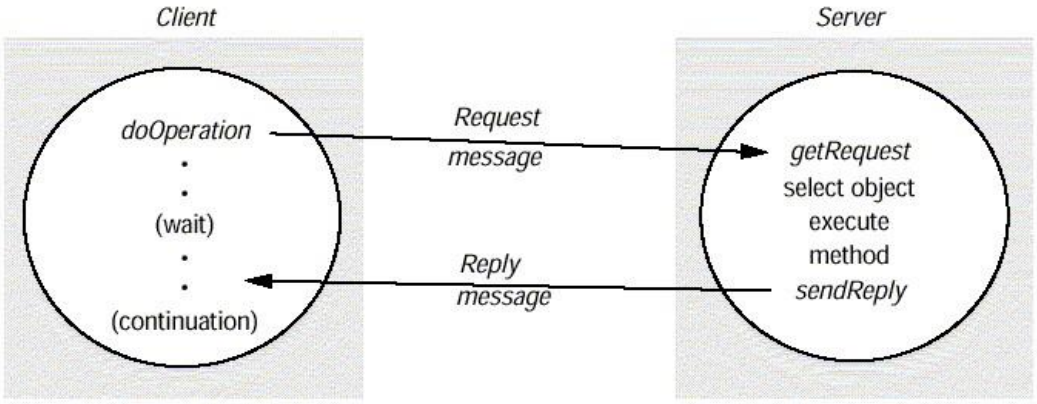
- The pairwise exchange of messages is not the best model for communication from one process to a group of other processes.
- A multicast operation is more appropriate.
- Multicast operation is an operation that sends a single message from one process to each of the members of a group of processes.
- The simplest way of multicasting, provides no guarantees about message delivery or ordering.
- Multicasting has the following characteristics:
  - Fault tolerance based on replicated services
    - A replicated service consists of a group of servers.
    - Client requests are multicast to all the members of the group, each of which performs an identical operation.
  - Finding the discovery servers in spontaneous networking
    - Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
  - Better performance through replicated data
    - Data are replicated to increase the performance of a service.
  - Propagation of event notifications
    - Multicast to a group may be used to notify processes when something happens.

Client-Server Communication

- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.

- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.
- Often built over UDP datagrams
- Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
- Avoidance of connection establishment overhead
- No need for flow control due to small amounts of data are transferred

The request-reply protocol



- The designed request-reply protocol matches requests to replies.
- If UDP datagrams are used, the delivery guarantees must be provided by the request- reply protocol, which may use the server reply message as an acknowledgement of the client request message.

RPC-

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

- It further aims at hiding most of the intricacies of message passing and is idle for client-server application.
- RPC allows programs to call procedures located on other machines. But the procedures „send" and „receive" do not conceal the communication which leads to achieving access transparence in distributed systems.
- Example: when process A calls a procedure on B, the calling process on A is suspended and the execution of the called procedure takes place. (PS: function, method, procedure difference, stub, 5 state process model definition)
- Information can be transported in the form of parameters and can come back in procedure result. No message passing is visible to the programmer. As calling and called procedures exist on different machines, they execute in different address

spaces, the parameters and result should be identical and if machines crash during communication, it causes problems.

Implementing RPC Mechanism-

To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of stubs, which provide a perfectly normal (local) procedure call abstraction by concealing from programs the interface to the underlying RPC system. We saw that an RPC involves a client process and a server process. Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Moreover, to hide the existence and functional details of the underlying network, an RPC communication package (known as RPC Runtime) is used on both the client and server sides. Thus, implementation of an RPC mechanism usually involves the following five elements of program

1. The client
2. The client stub
3. The RPC Runtime
4. The server stub
5. The server

The interaction between them is shown in Figure 3. The client, the client stub, and one instance of RPC Runtime execute on the client machine, while the server, the server stub, and another instance of RPC Runtime execute on the server machine. The job of each of these elements is described below.

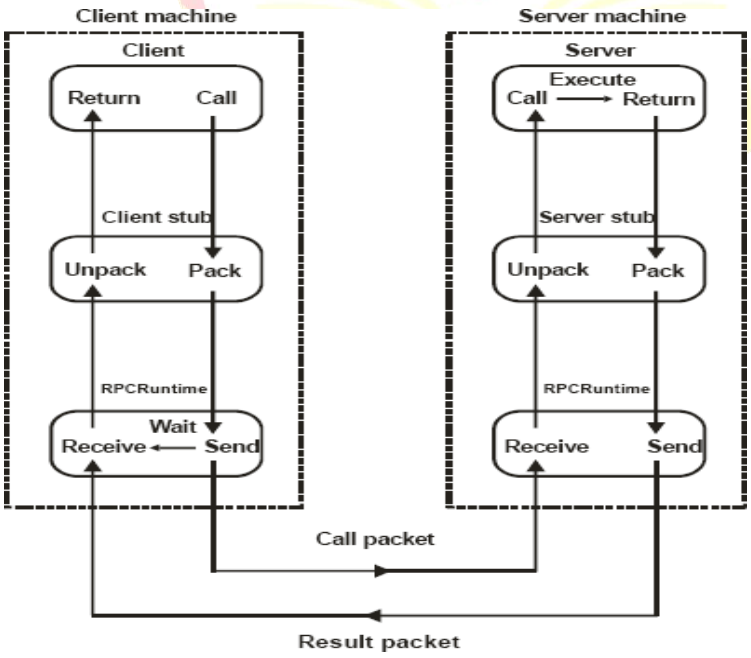


Fig 3. Implementing RPC Mechanism

Stub Generation:-

Stubs can be generated in one of the following two ways:

1. Manually: In this method, the RPC implementer provides a set of translation functions from which a user can construct his or her own stubs. This method is simple to implement and can handle very complex parameter types.

2. Automatically: This is the more commonly used method for stub generation. It uses Interface Definition Language (IDL) that is used to define the interface between a client and a server. An interface definition is mainly a list of procedure names supported by the interface, together with the types of their arguments and results. This is sufficient information for the client and server to independently perform compile-time type checking and to generate appropriate calling sequences. However, an interface definition also contains other information that helps RPC reduce data storage and the amount of data Transferred over the network. For example, an interface definition has information to indicate whether each argument is input, output, or both – only input arguments need be copied from client to server and only output arguments need be copied from server to client. Similarly, an interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls making it unnecessary for both the client and the server to store this information separately. A server program that implements procedures in an interface is said to export the interface and a client program that calls procedures from an interface is said to import the

3.

4. interface. When writing a distributed application, a programmer first writes an interface definition using the IDL. He or she can then write the client program that imports the interface and the server program that exports the interface. The interface definition is processed using an IDL computer to generate components that can be combined with client and server programs, without making any changes to the existing compliers. In particular, from an interface definition, an IDL complier generate a client stub procedure and a server such procedure for each procedure is the interface, the appropriate marshaling and un-marshaling operations in each stub procedure, and a header file that supports the data types in the interface definition. The header file is included in the source files of the client and server programs, the client stub procedures are compiled and linked with the client program, and the server stub procedures are compiled and linked with the server program. An IDL compiler is designed to process interface definitions for use with different languages, enabling clients and servers written in different languages, to communicate by using remote procedure calls.

RPC Messages:-

Any remote procedure call involves a client process and a server process that are possibly located on different computers. The mode of interaction between the client and server is that the client asks the server to execute a remote procedure and the server returns the result of execution of the concerned procedure to the client. Based on this mode of interaction, the two types of messages involved in the implementation of an RPC system are as follows :

1. Call messages that are sent by the client to the server for requesting execution of a particular remote procedure.
2. Reply messages that are sent by the server to the client for returning the result of remote procedure execution.

Call message is used to request execution of a particular remote procedure the two basic components necessary in a call message are as follows :
1. The identification information of the remote procedure to be executed.
2. The arguments necessary for the execution of the procedure.

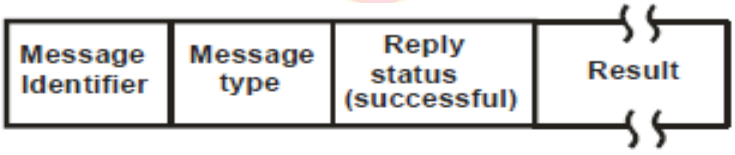
In addition to these two fields, a call message normally has the following fields.

3. A message identification field that consists of a sequence number. This field is useful of two ways – for identifying lost messages and duplicate messages in case of system failures and for properly matching reply messages to outstanding call messages, especially in those cases when the replies of several outstanding call messages arrive out of order.
4. A message type field that is used to distinguish call messages from reply messages. For example, in an RPC system, this field may be set to 0 for all call messages and set to 1 for all reply messages.
5. A client identification field that may be used for two purposes – to allow the server of the RPC to identify the client to whom the reply message has to be returned and to allow the server to check the authentication of the client process for executing the concerned procedure.

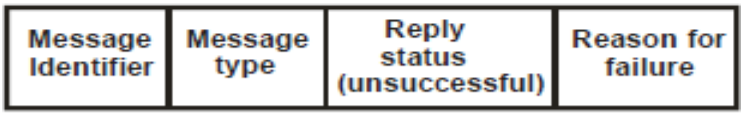
Message Identifier	Message Type	Client Identifier	Remote Procedure Identifier			Arguments
			Program Number	Version Number	Procedure Number	

Reply Messages

When the server of an RPC receives a call message from a client, it could be faced with one of the following conditions. In the list below, it is assumed for a particular condition that no problem was detected by the server for any of the previously listed conditions:



(a)



(b)

Fig 4: (a) a successful reply message format; (b) an unsuccessful reply message format

Synchronization:-

Clock synchronization deals with understanding the temporal ordering of events produced by concurrent processes. It is useful for synchronizing senders and receivers of messages, controlling joint activity, and the serializing concurrent access to shared objects. The goal is that multiple unrelated processes running on different machines should be in agreement with and be able to make consistent decisions about the ordering of events in a system. For these kinds of events, we introduce the concept of a *logical clock*, one where the clock need not have any bearing on the time of day but rather be able to create event sequence numbers that can be used for comparing sets of events, such as a messages, within a distributed system. Another aspect of clock synchronization deals with synchronizing time-of-day clocks among groups of machines. In this case, we want to ensure that all machines can report the same time, regardless of how imprecise their clocks may be or what the network latencies are between the machines.

A computer clock usually consists of three components – a quartz crystal that oscillates at a well – defined frequency, a counter register, and a holding register. The holding register is used to store a constant value that is decided based on the frequency of oscillation of the quartz crystal. That is, the value in the counter register is decremented by 1 for each oscillation of the quartz crystal. When the value of the counter register becomes zero, an interrupt is generated and its value is reinitialized to the value in the holding register. Each interrupt is called clock tick. The value in the holding register is chosen 60 so that on 60 clock ticks occur in a second.

- The time difference between two computers is known as **drift**. Clock drift over time is known as **skew**. Computer clock manufacturers specify a maximum skew rate in their products.
- Computer clocks are among the least accurate modern timepieces.
    - Inside every computer is a chip surrounding a quartz crystal oscillator to record time. These crystals cost 25 seconds to produce.
    - Average loss of accuracy: 0.86 seconds per day
- This skew is unacceptable for distributed systems. Several methods are now in use to attempt the synchronization of physical clocks in distributed systems:

Physical Clocks –Christian's Algorithm

Assuming there is one time server with UTC:

- Each node in the distributed system periodically polls the time server.
- Time(T1) is estimated as Stime + (T1 –T0)/2
- This process is repeated several times and an average is provided.
- Machine T1 then attempts to adjust its

time. Disadvantages:

- Must attempt to take delay between server T1 and time server into account
- Single point of failure if time server fails

Physical Clocks –Berkeley Algorithm

One daemon without UTC:

- Periodically, the daemon polls all machines on the distributed system for their times.
- The machines answer.
- The daemon computes an average time and broadcasts it to the machines so they can adjust.

Physical Clocks –Decentralized Averaging Algorithm

- Each machine on the distributed system has a daemon without UTC.
- Periodically, at an agreed-upon fixed time, each machine broadcasts its local time.
- Each machine calculates the correct time by averaging all results.

Mutual Exclusion:-

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such as tape drives or printers must be restricted to a single process at a time. Therefore, exclusive access to such a shared resource by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes. The sections of a program that need exclusive access to shared resources are referred to as critical sections. For mutual exclusion, means are introduced to prevent processes from executing concurrently within their associated critical sections.

Election Algorithms:-

Several distributed algorithms require that there be a coordinator process in the entire system that performs some type of coordination activity needed for the smooth running of other processes in the system. Two examples of such coordinator processes encountered in this chapter are the coordinator in the centralized algorithm for mutual exclusion and the central coordinator in the centralized deadlock detection algorithm.

- Since all other processes in the system have to interact with the coordinator, they all must unanimously agree on who the coordinator is. Furthermore, if the coordinator process fails due to the failure of the site on which it is located, a new coordinator process must be elected to take up the of the failed coordinator.
- Election algorithms are meant for electing to take 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.

Election algorithm are based on the following assumptions :

1. Each process in the system has a unique priority number.
2. Whenever an election is held, the process having the highest priority number among the currently active processes is elected as the coordinator.

3. On recovery, a failed process can take appropriate actions to rejoin the set of active processes. 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. Different election algorithms differ in the way they do this.

Two such election algorithms are described below-

When any process notices that the coordinator is no longer responding to the requests, it asks for the election. Example: A process P holds an election as follows

1) P sends an ELECTION message to all the processes with higher numbers.

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

3) If one higher process answers; it takes over the job and P‟s job is done.

At any moment an “election” message can arrive to process from one of its lowered numbered colleague. The receiving process replies with an OK to say that it is alive and can take over as a coordinator. Now this receiver holds an election and in the end all the processes give ok except one and that one is the new coordinator. The new coordinator announces its new post by sending all the processes a message that it is starting immediately and is the new coordinator of the system. If the old coordinator was down and if it gets up again; it holds for an election which works in the above mentioned fashion. The biggest numbered process always wins and hence the name “bully” is used for this algorithm.

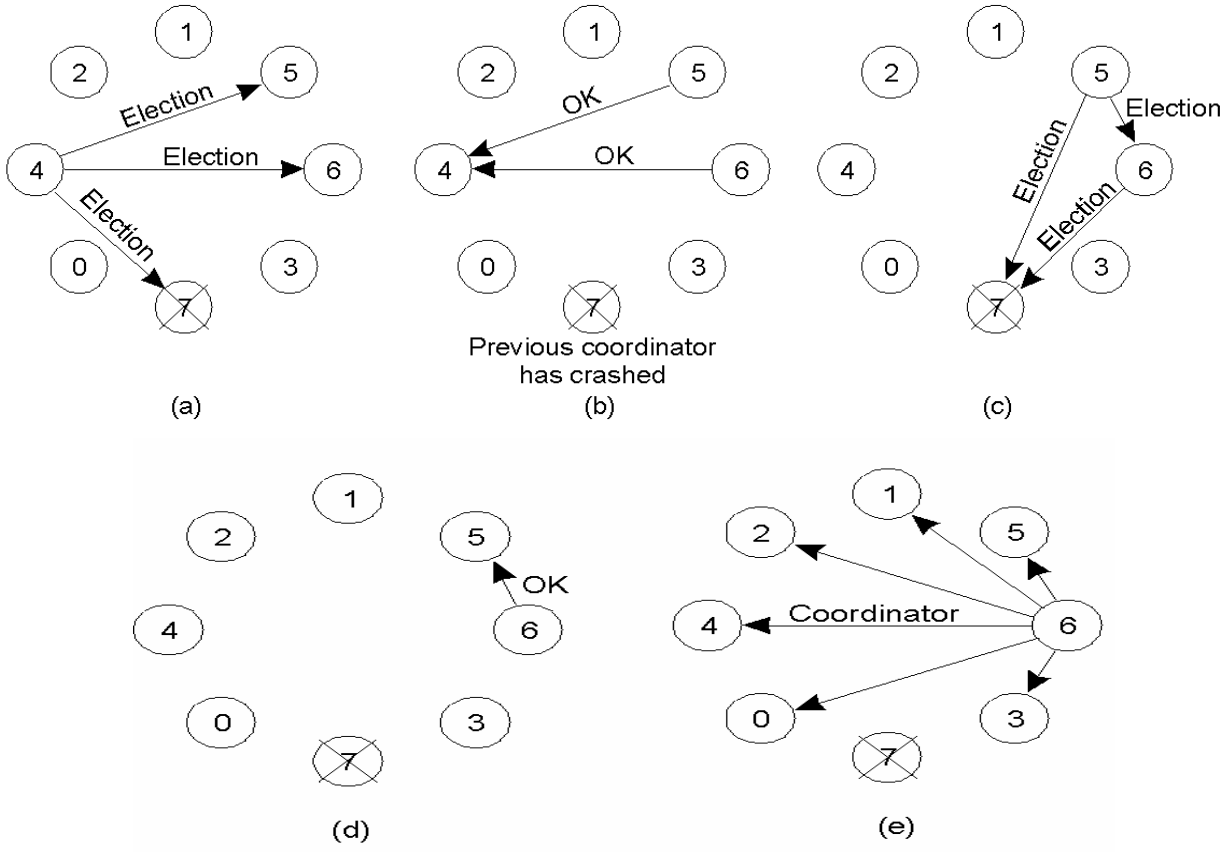


Fig 5: Bully Election Algorithm

Ring Algorithm:

It is based on the use of a ring as the name suggests. But this does not use a toke. Processes are physically ordered in such a way that every process knows its successor.

When any process notices that the coordinator is no longer functioning, it builds up an ELECTION message containing its own number and passes it along the to its successor. If the successor is down, then sender skips that member along the ring to the next working process.

At each step, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as the coordinator. At the end, the message gets back to the process that started it.

That process identifies this event when it receives an incoming message containing its own process number. Then the same message is changed as coordinator and is circulated once again. Example: two process, Number 2 and Number 5 discover together that the previous coordinator; Number 7 has crashed. Number 2 and Number 5 will each build an election message and start circulating it along the ring. Both the messages in the end will go to Number 2 and Number 5 and they will convert the message into the coordinator with exactly the same number of members and in the same order. When both such messages have gone around the ring, they both will be discarded and the process of election will re-start.

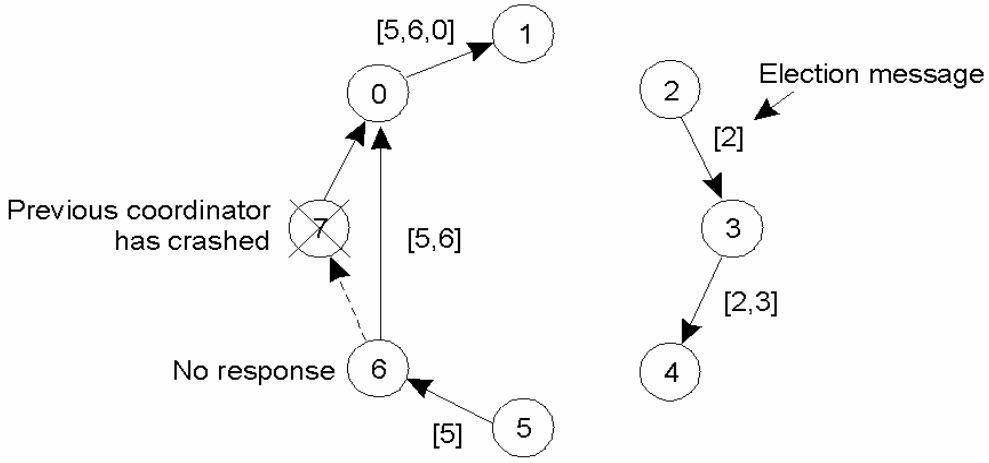


Fig. 6 Ring Algorithm

Unit IV

Distributed Scheduling-Issues in Load Distributing

Distributed scheduling is concerned with distributing the load of a system among the available resources in a manner that improves overall system performance and maximizes resource utilization.
The basic idea is to transfer tasks from heavily loaded machines to idle or lightly loaded ones.

Load distributing algorithms can be classified as static, dynamic or adaptive.
Static means decisions on assignment of processes to processors are hardwired into the algorithm, using a priori knowledge, perhaps gained from analysis of a graph model of the application.

Dynamic algorithms gather system state information to make scheduling decisions and so can exploit under-utilized resources of the system at run time but at the expense of having to gather system information. An adaptive algorithm changes the parameters of its algorithm to adapt to system loading conditions. It may reduce the amount of information required for scheduling decisions when system load or communication is high.

Most models assume that the system is fully connected and that every processor can communicate, perhaps in a number of hops, with every other processor although generally, because of communication latency, load scheduling is more practical on tightly coupled networks of homogeneous processors if the workload involves communicating tasks. In this way also, those tasks might take advantage of multicast or broadcast features of the communication mechanism.

Processor allocation strategies can be non migratory (non preemptive) or migratory (preemptive). Process migration involves the transfer of a running process from one host to another. This is an expensive and potentially difficult operation to implement. It involves packaging the tasks virtual memory, threads and control block, I/O buffers, messages, signals and other OS resources into messages which are sent to the remote host, which uses them to reinitiate the process. Non migratory algorithms involve the transfer of tasks which have not yet begun, and so do not have this state information, which reduces the complexities of maintaining transparency.

Resource queue lengths and particularly CPU queue length are good indicators of load as they correlate well with the task response time. It is also very easy to measure queue length. There is a danger though in making scheduling decisions too simplistic. For example, a number of remote hosts could observe simultaneously that a particular site had a small CPU queue length and could initiate a number of process transfers. This may result in that site becoming flooded with processes and its first reaction might be to try to move them elsewhere. As migration is an expensive operation, we do not want to waste resources (CPU time and communication bandwidth) by making poor choices which result in increased migration activity.

Components for Load Distributing Algorithms

A load distributing algorithm has, typically, four components: - transfer, selection, location and information policies.

Transfer Policy

When a process is an about to be created, it could run on the local machine or be started elsewhere. Bearing in mind that migration is expensive; a good initial choice of location for a process could eliminate the need for future system activity. Many policies operate by using a threshold. If the machine's load is below the threshold

then it acts as a potential receiver for remote tasks. If the load is above the threshold, then it acts as a sender for new tasks. Local algorithms using thresholds are simple but may be far from optimal.

Process Selection Policy

A selection policy chooses a task for transfer. This decision will be based on the requirement that the overhead involved in the transfer will be compensated by an improvement in response time for the task and/or the system. Some means of knowing that the task is long-lived will be necessary to avoid needless migration. This could be based perhaps on past history.

A number of other factors could influence the decision. The size of the task's memory space is the main cost of migration. Small tasks are more suited. Also, for efficiency purposes, the number of location dependent calls made by the chosen task should be minimal because these must be mapped home transparently. The resources such as a window or mouse may only be available at the task's originating site.

Site Location Policy

Once the transfer policy has decided to send a particular task, the location policy must decide where the task is to be sent. This will be based on information gathered by the information policy.

Polling is a widely used sender-initiated technique. A site polls other sites serially or in parallel to determine if they are suitable sites for a transfer and/or if they are willing to accept a transfer. Nodes could be selected at random for polling, or chosen more selectively based on information gathered during previous polls. The number of sites polled may vary.

A receiver-initiated scheme depends on idle machines to announce their availability for work. The goal of the idle site is to find some work to do. An interesting idea is for it to offer to do work at a price, leaving the sender to make a cost/performance decision in relation to the task to be migrated.

Information Policy

The information policy decides what information about the states of other nodes should be collected and where it should be collected from. There are a number of approaches:

Demand Driven

A node collects the state of other nodes only when it wishes to become involved in either sending or receiving tasks, using sender initiated or receiver initiated polling schemes. Demand driven policies are inherently adaptive and dynamic as their actions depend on the system state.

Periodic

Nodes exchange information at fixed intervals. These policies do not adapt their activity to system state, but each site will have a substantial history over time of global resource usage to guide location algorithms. Note that the benefits of load distribution are minimal at high system loads and the periodic exchange of information therefore may be an unnecessary overhead.

State-Change Driven

Nodes disseminate state information whenever their state changes by a certain amount. This state information could be sent to a centralized load scheduling point or to peers.

Different Types of Load Distributing Algorithms

Classification According to Approach

Load distribution algorithms can be classified as static, dynamic or adaptive. Static schemes are those when the algorithm uses some priori information of the system based on which the load is distributed from one

server to another. The disadvantage of this approach is that it cannot exploit the short term fluctuations in the system state to improve performance. This is because static algorithms do not collect the state information of the system. These algorithms are essentially graph theory driven or based on some mathematical programming aimed to find a optimal schedule, which has a minimum cost function. Dynamic scheduling collect system state information and make scheduling decisions on these state information. An extra overhead of collecting and storing system state information is needed but they yield better performance than static ones.

Adaptive load balancing algorithms are a special class of dynamic load distribution algorithms, in that they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state.

Pre-emptive and Non pre-emptive Type

A pre-emptive transfer involves transfer of task which is partially executed. These transfers are expensive because the state of the tasks also needs to be transferred to the new location.

Non pre-emptive transfers involve transfer of tasks which has not been started. For a system that experiences wide fluctuations in load and has a high cost for the migration of partly executed tasks, non pre-emptive transfers are appropriate.

Load Sharing and Load Balancing

Although both type of algorithms strive to reduce the likelihood of unshared state i.e. wait and idle state, load balancing goes a step further by attempting to equalize loads at all computers. Because a load balancing algorithm involves more task transfers than load sharing algorithms, the higher overhead incurred by load balancing types may outweigh its potential improvement.

Initiation Based

In general the algorithms are also categorized on which node initiates the load distribution activity. The variations are sender initiated, receiver initiated or symmetrically initiated (by both sender and receiver).

Task Migration and its issues

Although it is difficult to implement, a mechanism for process migration broadens the scope and flexibility of load sharing algorithms. Decisions on process to processor assignment can be dynamically reconsidered, in response to changing system conditions or user or process preferences. Tasks can be preempted and moved elsewhere rather than being statically assigned to a host until completion. Some of the motivating situations where such a mechanism could be useful are given next.

Load Balancing

Empirical evidence from some studies has shown that often, a small subset of processes running on a multiprocessor system often account for much of the load and a small amount of effort spent off-loading such processes may yield a big gain in performance.

Load sharing algorithms avoid idle time on individual machines when others have non-trivial work queues. **Load balancing** algorithms attempt to keep the work queues similar in length and reduce average response time, but are clearly more complex as they have more global and deterministic requirements. Commonly, load level is defined as CPU utilisation of hosts in a catchment area. A more elaborate index could be developed based on the contention for a number of resources and which employed a smoothing or dampening technique. An important metric of the load balancing policy is the anticipated residency period required by the process, to avoid useless and costly migrations. A competitive algorithm allows a process to migrate only after it executes for a period of time that equals or exceeds its predicted migration time. This

idea is based on the probabilistic assumption that a process will exhibit a 'past-repeat' execution pattern. This approach has been shown to be simple to implement, provides an adequate method of imposing the residency requirement and adapts well to the current workload state of the participating processors.

Communication Performance

Network saturation can be caused by heavy communication traffic induced by data transfers from one task to another, residing on separate hosts. The inter processor traffic is always the most costly and the least reliable factor in a loosely coupled distributed system. As a result of the saturation effect, the incremental increase in computing resources could actually degrade system throughput.

This situation can occur when processes perform data reduction on a volume of data larger than the process's size, e.g. a large database. In these circumstances, it may be advantageous to move the process to the data, rather than using network bandwidth, especially between geographically distant parties. The principle of spatial locality indicates that strongly interacting entities should be mapped in close proximity. In other words, the system must endeavour to match the logical locality with the physical locality.

It has been observed that for many parallel and distributed computations that it may be possible to determine communication loads at compile time and use this information to guide migration at runtime by including additional information in the process descriptor. To reduce IPC costs, the compiler may analyze the structure and phases of communication and suggest which processes should reside on the same machines. One technique is to associate a migration identifier with each process and to keep processes with equivalent migration identifiers on the one machine. A second technique (capitalizing on compiler intelligence) known as eager migration can be used when it is known that a process A will soon migrate to B's location. Portions of process A may be piggybacked, in advance, on other messages bound for the same destination, to reduce migration time.

Fault Tolerance

Long running processes may be considered as valuable elements in any system because of the resource expenditure that has been outlaid already. Failure of mature processes becomes important if time and resources are precious and cannot be reassigned. Long running processes are particularly susceptible to transient hardware faults and may be worth rescuing by rapid evacuation. That is, a policy could be adopted which prioritized migrations from a machine about to shut down, based on the order of their computational value to the system. Such policies would require a high degree of concurrency within the migration mechanism to abandon a site quickly.

Application Concurrency

The divide and conquer, or crowd approach to problem solving decomposes a problem into a set of smaller problems, similar in nature, and solves them separately. The partial solutions are combined to form a final solution to the original problem. The smaller problems may be solved independently and concurrently in many cases using a copy or subset of initial data, exemplified by a tree or graph structure. Evaluating a game position, multiplying matrices or finding shortest paths can be performed in this way. Applications exhibiting high concurrency with little inter-task communication can benefit from these tasks being distributed to execute with true concurrency, on separate hosts. Processor pool architectures are particularly suited to these applications where migration may occur before processes establish locally.

Deadlock-Issues in deadlock detection & resolution

Deadlock can occur whenever two or more processes are competing for limited resources and the processes are allowed to acquire and hold a resource (obtain a lock) thus preventing others from using the resource while the process waits for other resources. Two common places where deadlocks may occur are with processes in an operating system (distributed or centralized) and with transactions in a database.

- Locking protocols such as the popular Two Phase give rise to deadlock as follows: process A gets a lock on data item X while process B gets a lock on data item Y. Process A then tries to get a lock on Y. As Y is already locked, process A enters a blocked state. Process B now decides to get a lock on X, but is blocked. Both processes are now blocked, and, by the rules of Two Phase Locking, neither will relinquish their locks.
- No progress will take place without outside intervention.
- Several processes can be involved in a deadlock when there exists a cycle of processes waiting for each other.
    - Process A waits for B which waits for C which waits for A.

Locks

Two major types of locks are utilized:

- Write-lock (exclusive lock) is associated with a database object by a transaction (the transaction locks it; acquires lock for it) before writing (inserting/modifying/deleting) this object.
- Read-lock (shared lock) is associated with a database object by a transaction before reading (retrieving the state of) this object.
- The common interactions between these lock types are defined by blocking behavior as follows:
    - An existing write-lock on a database object blocks an intended write upon the same object (already requested/issued) by another transaction by blocking a respective write-lock from being acquired by the other transaction. The second write-lock will be acquired and the requested write of the object will take place (materialize) after the existing write-lock is released.
    - Write-lock blocks an intended (already requested/issued) read by another transaction by blocking the respective read lock.
    - A read-lock blocks an intended write by another transaction by blocking the respective write-lock.
    - A read-lock does not block an intended read by another transaction. The respective read-lock for the intended read is acquired (shared with the previous read) immediately after the intended read is requested, and then the intended read itself takes place.
    - A read-lock blocks a write-lock.
    - A write-lock blocks a read-lock and a write-lock.

Distributed deadlock

Occurs when processes spread over different computers in a network wait for events that will not occur.

Three types of distributed deadlock:

- Resource deadlock
- Communication deadlock
    - Circular waiting for communication signals
- Phantom deadlock
    - Due to the communications delay associated with distributed computing, it is possible that a deadlock detection algorithm might detect a deadlock (called phantom deadlock, a perceived deadlock) that does not exist.
    - Although this form of deadlock cannot immediately cause the system to fail, it is a source of inefficiency

Four Required Conditions for Deadlock

- Exclusive use – when a process accesses a resource, it is granted exclusive use of that resource.
- Hold and wait – a process is allowed to hold onto some resources while it is waiting for other resources.
- No preemption – a process cannot preempt or take away the resources held by another process.

- Cyclical wait – there is a circular chain of waiting processes, each waiting for a resource held by the next process in the chain.

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-for graph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

- Choose the youngest transaction.
- Choose the transaction with fewest data items.
- Choose the transaction that has performed least number of updates.
- Choose the transaction having least restart overhead.
- Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

In the fig 2 we can see that how deadlock occur in the centralized system and Distributed system (at two different sites forming the deadlock condition.)

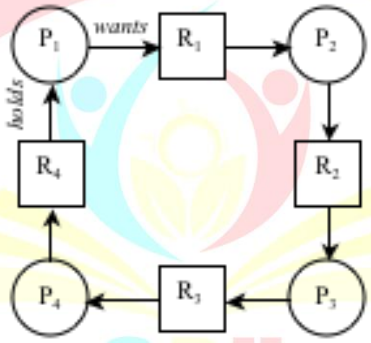
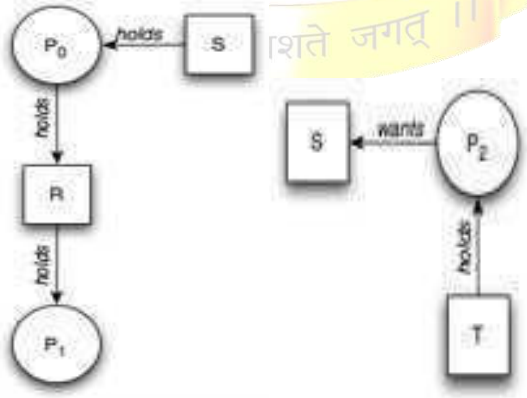


Fig 1: Deadlock (Centralized System)



(a) (b)

Fig 2: Dead Lock in Distributed System (a) At site A (b) At site B

Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are transaction location and transaction control. Once these

concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

- One server may be selected as the center of control.
- The center of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

- **Distributed Wound-Die**

- If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
- If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.

- **Distributed Wait-Wait**
  - If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
  - If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

- **Centralized Deadlock Detector** − One site is designated as the central deadlock detector.
- **Hierarchical Deadlock Detector** − A number of deadlock detectors are arranged in hierarchy.
- **Distributed Deadlock Detector** − All the sites participate in detecting deadlocks and removing them.

Distributed Deadlock-Detection Algorithms

A Path-Pushing Algorithm

- The site waits for deadlock-related information from other sites
- The site combines the received information with its local TWF graph to build an updated TWF graph
- For all cycles 'EX -> T1 -> T2 -> Ex' which contains the node 'Ex', the site transmits them in string form 'Ex, T1, T2, Ex' to all other sites where a sub-transaction of T2 is waiting to receive a message from the sub-transaction of T2 at that site.

Edge-Chasing Algorithm

Chandy-Misra-Haas's Algorithm:

- A probe(i, j, k) is used by a deadlock detection process Pi. This probe is sent by the home site of Pj to Pk.
- This probe message is circulated via the edges of the graph. Probe returning to Pi implies deadlock detection.
- Terms used:
  - Pj is dependent on Pk, if a sequence of Pj, Pi1,.., Pim, Pk exists.
  - Pj is locally dependent on Pk, if above condition + Pj,Pk on same site.
  - Each process maintains an array dependenti: dependenti(j) is true if Pi knows that Pj is dependent on it. (initially set to false for all i & j).

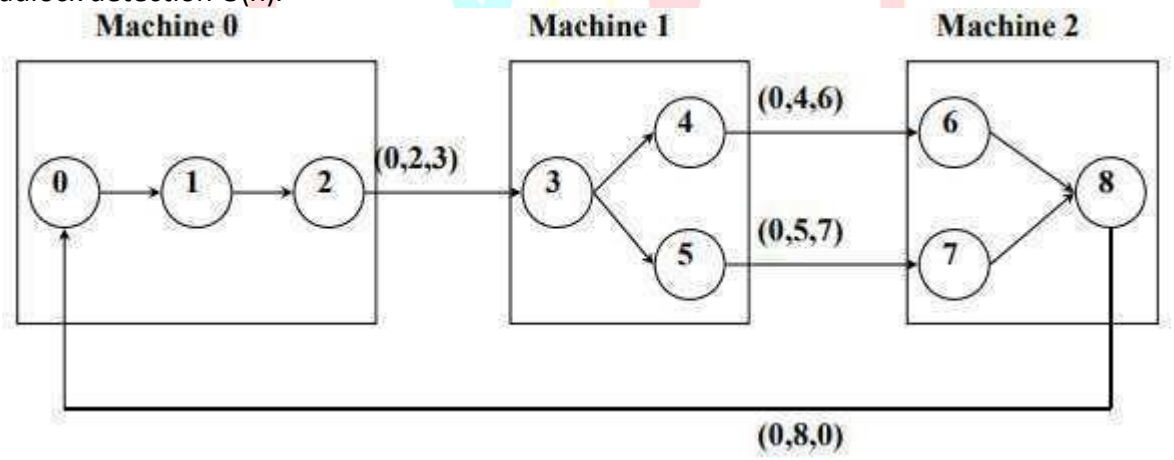
Sending the probe:

if Pi is locally dependent on itself then deadlock.
else for all Pj and Pk such that
(a) Pi is locally dependent upon Pj, and
(b) Pj is waiting on Pk, and
(c) Pj and Pk are on different sites, send probe(i,j,k) to the home site of Pk.

Receiving the probe:
if (d) Pk is blocked, and
(e) dependentk(i) is false, and
(f) Pk has not replied to all requests of Pj,
then begin
dependentk(i) := true;
if k = i then Pi is deadlocked
else ...

Receiving the probe:
……. else for all Pm and Pn such that
(a') Pk is locally dependent upon Pm, and
(b') Pm is waiting on Pn, and
(c') Pm and Pn are on different sites, send probe(i,m,n) to the home site of Pn. end.

Performance:
For a deadlock that spans m processes over n sites, m(n-1)/2 messages are needed. Size of the message 3 words.
Delay in deadlock detection O(n).



There are several ways to break the deadlock:
– The process that initiates commit suicide -- this is overkilling because several processes might initiate a probe and they will all commit suicide in fact only one of them is needed to be killed.
– Each process append its id onto the probe, when the probe come back, the originator can kill the process which has the highest number by sending him a message. (Even for several probes, they will all choose the same guy)

Unit-5

Distributed Multimedia & Database system

Distributed Data Base Management System (DDBMS)-

A distributed database system allows applications to access data from local and remote databases. In a homogenous distributed database system, each database is an Oracle Database. In a heterogeneous distributed database system, at least one of the databases is not an Oracle Database. Distributed databases use client/server architecture to process information requests.

- A distributed database system consists of loosely coupled sites that share no physical component.
- Database systems that run on each site are independent of each other.
- Transactions may access data at one or more

sites. It contains the following database systems:

Homogenous Distributed Database Systems-

In a homogeneous distributed database

- All sites have identical software
- Are aware of each other and agree to cooperate in processing user requests.
- Each site surrenders part of its autonomy in terms of right to change schemas or software
- Appears to user as a single system

A homogenous distributed database system is a network of two or more Oracle Databases that reside on one or more machines.

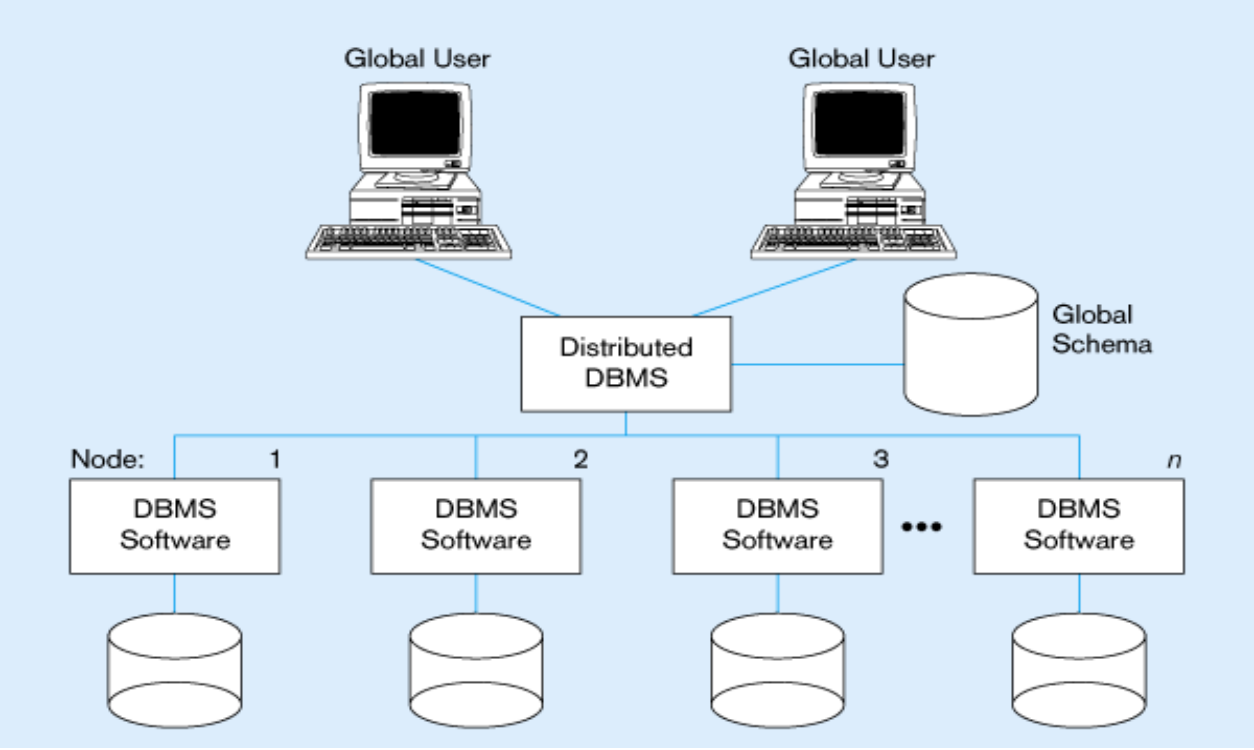


Fig 1: Homogenous Distributed Data Base System

Heterogeneous Distributed Data Base-

In a heterogeneous distributed database

- Different sites may use different schemas and software
  - Difference in schema is a major problem for query processing
  - Difference in software is a major problem for transaction processing
- Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

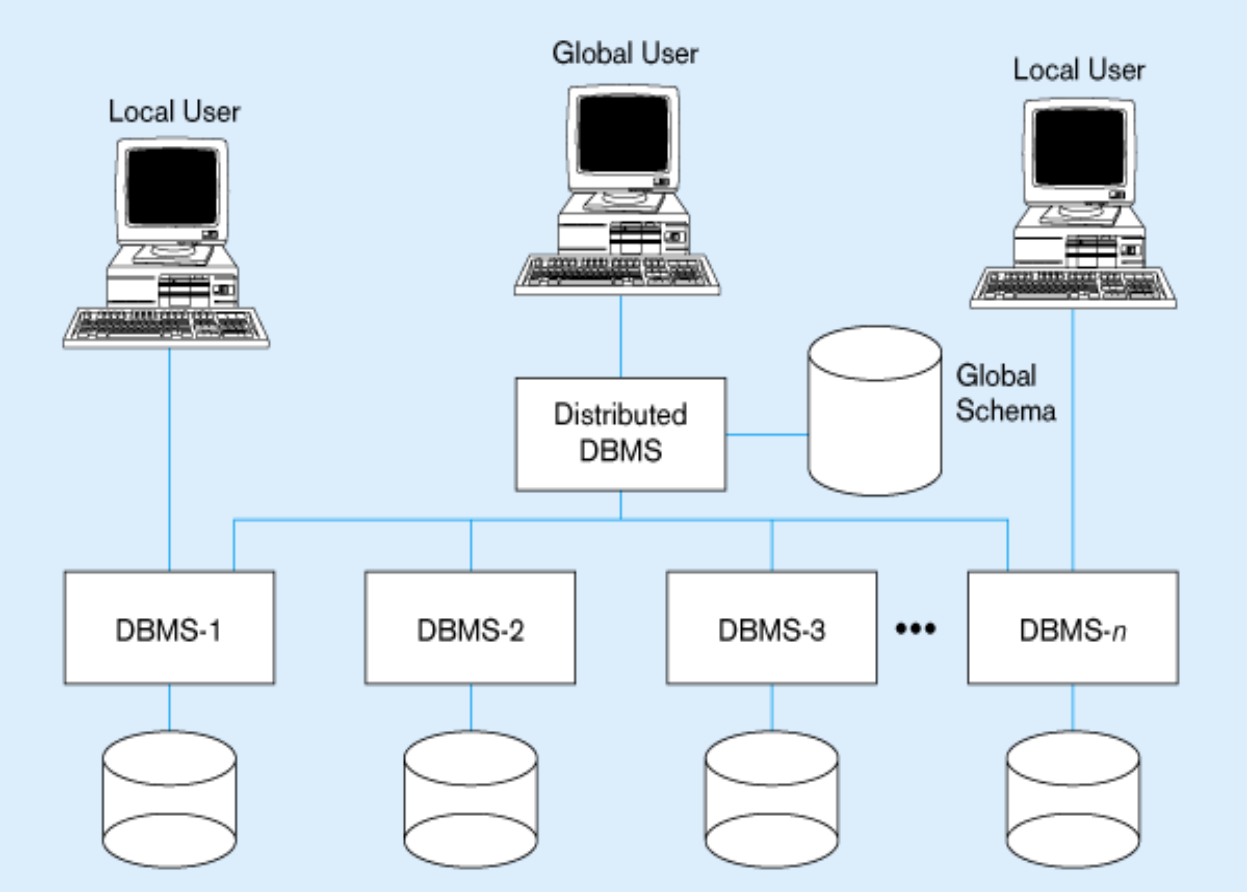


Fig. 2: Heterogeneous Distributed Database System

Advantages of DDBMS:

- Data are located near "greatest demand" site
- Faster data access
- Faster data processing
- Growth facilitation
- Improved communications
- Reduced operating costs
- User-friendly interface
- Less danger of a single-point failure
- Processor independence

Disadvantages of DDBMS: -

- Complexity of management and control
- Security
- Lack of standards
- Increased storage requirements
- Greater difficulty in managing the data environment
- Increased training cost

Distributed Processing Vs Distributed Data base: -

- Distributed processing – a database's logical processing is shared among two or more physically independent sites that are connected through a network-
  - One computer performs I/O, data selection and validation while second computer creates reports.
  - Uses a single-site database but the processing chores are shared among several sites.
- Distributed database – stores a logically related database over two or more physically independent sites. The sites are connected via a network -
  - Database is composed of database fragments which are located at different sites and may also be replicated among various sites.

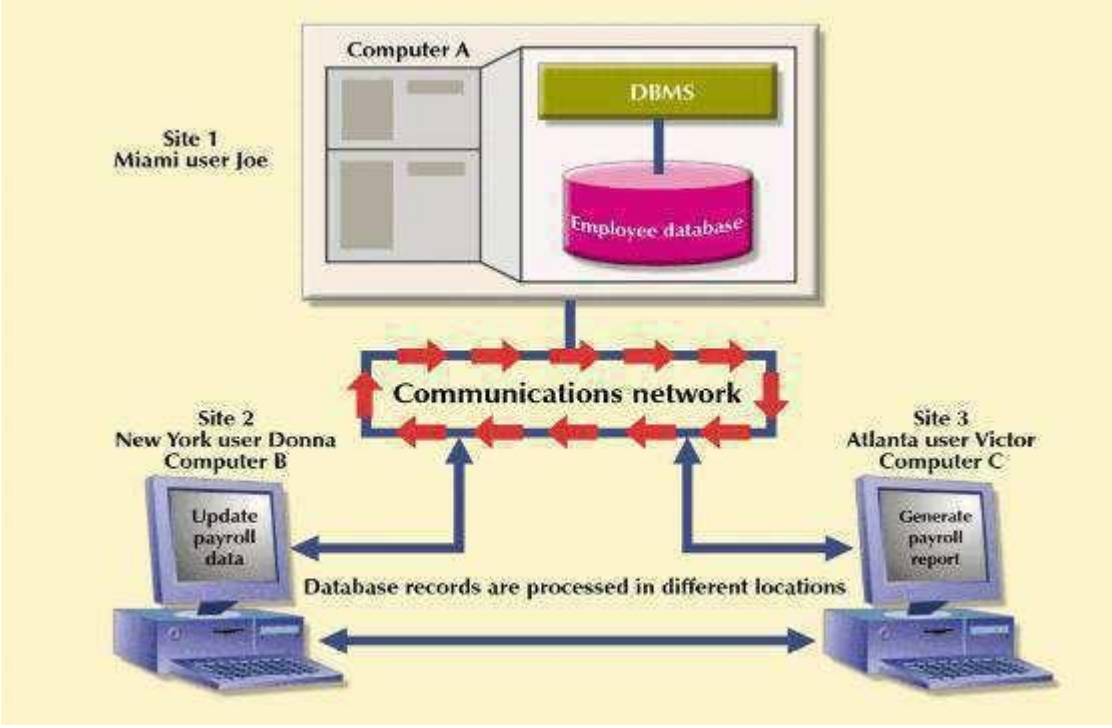


Fig 3: Distributed Processing Environment

Distributed Multimedia: -

Modern computers can handle streams of continuous, time-based data such as digital audio and video. This capability has led to the development of distributed multimedia applications such as networked video libraries, Internet telephony and video conferencing. Such applications are

viable with current general-purpose networks and systems, although the quality of the resulting audio and video is often less than satisfactory. More demanding applications such as large-scale video conferencing, digital TV production, interactive TV and video surveillance systems are beyond the capabilities of current networking and distributed system technologies. Multimedia applications demand the timely delivery of streams of multimedia data to end users. Audio and video streams are generated and consumed in real time, and the timely delivery of the individual elements (audio samples, video frames) is essential to the integrity of the application. In short, multimedia systems are real-time systems: they must perform tasks and deliver results according to a schedule that is externally determined. The degree to which this is achieved by the underlying system is known as the *quality of service* (QoS) enjoyed by an application.

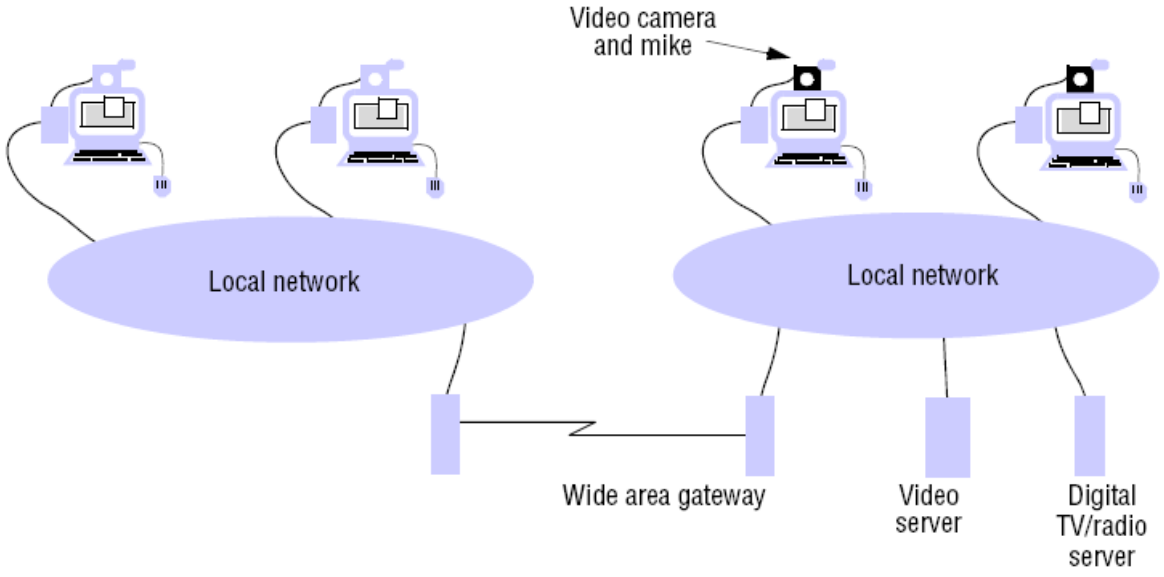


Fig 4: Distributed Multimedia

Characteristics of Multimedia: -

- Referring to video and audio data as continuous and time based.
- Continuous refers to the user's view of data.
- Internally, continuous media are represented as sequences of discrete values that replace each other over time.

  For Example:

  - The value of an image array is replaced 25 times per second to give the impression of a TV-quality view of moving scene.
  - A Sound amplitude value is replaced 8000 times per second to convey telephone- quality speech.
- Multimedia streams are said to be time-based because timed data elements in audio and video streams define the content of the stream.
- The systems that support multimedia applications need to preserve the timing when they handle continuous data

- Multimedia streams are often bulky. Hence systems that support multimedia applications need to move data with greater throughput than conventional systems.
  Example:
- A standard video stream requires more than 120Mbps, which exceeds the capacity of a 100Mbps Ethernet network.
- The use of compressed representations is therefore essential.

Quality of Service Managements: -

When multimedia applications run in networks of personal computers, they compete for resources at the workstations running the applications (processor cycles, bus cycles, buffer capacity) and in the networks (physical transmission links, switches, gateways). Workstations and networks may have to support several multimedia and conventional applications. There is competition between the multimedia and conventional applications, between different multimedia applications and even between the media streams within individual applications.

.

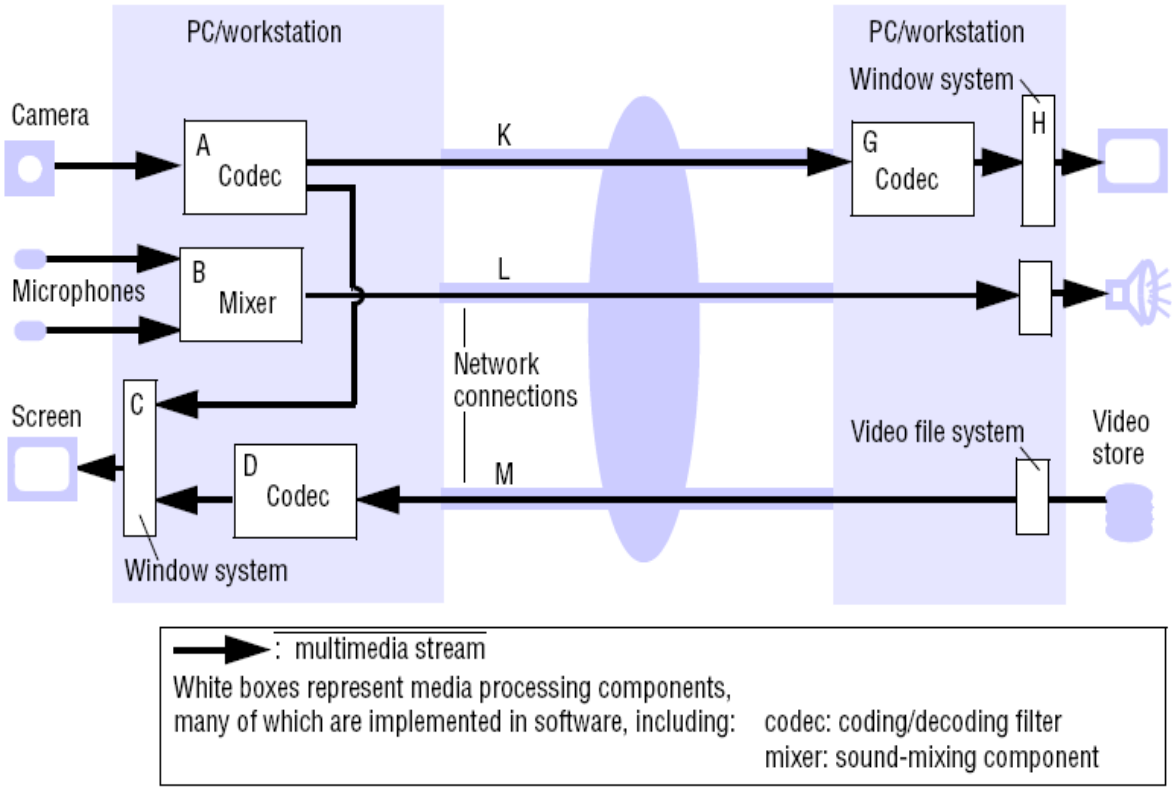


Fig 5: Typical infrastructure components for multimedia applications

The planned allocation is referred to as quality of service management.

QoS manager's responsibilities are:

- Quality of Service negotiation
- Admission Control

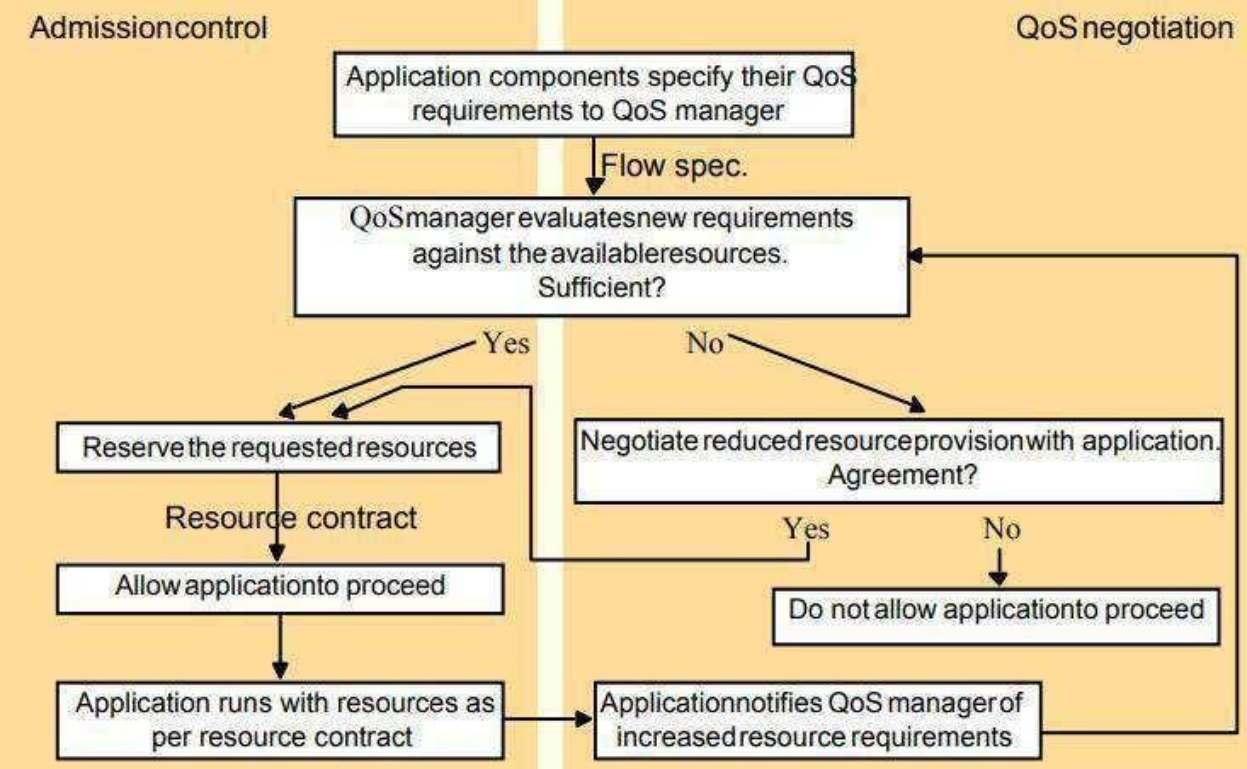


Fig 6: QoS Managers Task

Quality of service negotiation

- The application indicates its resource requirements to the QoS manager.
- The QoS manager evaluates the feasibility of meeting the requirements against a database of the available resources and current resource commitments and gives a positive or negative response.

Three parameters are of primary interest when it comes to processing and transporting multimedia streams:

- Bandwidth
- Latency
- Loss rate

– bandwidth

• The rate at which a multimedia stream flow. – Latency

• The time required for an individual data element to the time required for an individual data element to move through a stream from the source to the destination

• jitter

– Loss rate

• Data loss due to unmet resource requirements

Traffic Shaping: -

- Traffic Shaping is the term used to describe the use of output buffering to smooth the flow of data elements.

- The closer the actual traffic pattern matches the description, the better a system will be able to handle the traffic, when it uses scheduling methods.

Case Study of Distributed System: -

Amoeba is a complete and novel distributed operating system constructed as a collection of user level servers supported by the microkernel. Mach and Chorus are primarily microkernel designs geared towards the emulation of existing operating systems, notably UNIX, in a distributed system. Mach, Chorus and Amoeba have many common general goals, including the support of network transparency, encapsulated resource management and user-level servers. In Mach and Chorus, some objects are managed by the kernel and others by user-level servers, whereas all Amoeba objects are managed outside the kernel. Chorus allows user-level servers to be loaded dynamically in the kernel address space. Mach, Chorus and Amoeba all provide separate abstractions of processes and threads. Mach and Chorus can take advantage of a multiprocessor. The Amoeba kernel interface is very simple, because of its simple communication model and lack of support for virtual memory. The Mach and Chorus kernels offer many more calls due to their more complex communication models and the desire to emulate UNIX.

Naming and protecting resources - Resources are named and protected by capabilities in Amoeba and Chorus and by ports in Mach. Resources identified by capabilities in Amoeba and in Chorus are accessed by sending a message to the appropriate server port and the server accesses the particular resource identified in the capability. In Mach, servers generally manage many ports, one for every resource. Resources are accessed by sending messages to the corresponding ports. Capabilities alone are not suitable for implementing the sort of identity-based access control required in UNIX file systems. The Chorus kernel provides protection identifiers to enable user level services to authenticate the actor that sent a message and the port used by that actor. Mach's port rights are capabilities that confer send or receive rights on the process that possesses them. However, unlike Amoeba's capabilities and Chorus's port identifiers, which can be freely constructed and manipulated at user-level, Mach's port rights are stored inside the kernel and protected by it, allowing efficient representations and rapid access. But the Mach kernel has the additional expense of processing port rights in messages. For local communication, the Mach approach eliminates the need for random number generators and one- way functions, which are associated with Amoeba capabilities. However, in a secure environment, the Mach network servers must encrypt port rights transmitted in messages. Amoeba capabilities and Chorus capabilities can persist beyond the execution of any process that uses them. An Amoeba capability for a persistent resource such as a file can be stored in a directory. Each component of the capability, in particular the port identifier, remains valid as long as the file exists and the check field has not been changed at the server. By contrast, Mach capabilities for send rights are volatile. To access a resource, a Mach client requires a higher- level, persistent identifier to obtain the current identifier of the appropriate send rights; this higher-level identifier must be resolved by the service concerned before access can be obtained. There does not seem to be a clear winner between Mach's scheme of kernel-managed port capability transfers, and the Chorus and Amoeba scheme of user-controlled

capability transfers. In Amoeba, processes are obliged to generate port identifiers themselves, and then test for their

uniqueness. The Chorus port naming scheme improves upon this since the kernel generates UIs for port identifiers, thus avoiding clashes. Both Amoeba and Chorus allow for groups of servers to manage resources. In Amoeba the processes form groups, but in Chorus processes effectively become members of groups by making their ports join port groups.

Inter process communication - The Mach, Chorus and Amoeba kernels all provide a synchronous request-reply protocol. Chorus and Mach also provide for asynchronous message passing. Mach packages all forms of message passing in a single system call, whereas Chorus provides alternative calls. Amoeba and Chorus provide for group communication. In the case of Amoeba this is a reliable, totally ordered multicast to be used by the members of a process group. Chorus provides an unreliable multicast to all the members of a group of ports or to selected members. Chorus services can be reconfigured by servers adding or removing their ports from a group.

Messages - Amoeba messages consist of a fixed-size header and an optional out-of-line block of data of variable size. But Mach is more flexible in that it allows multiple out-of-line blocks in a single message. Mach and Chorus employ their virtual memory management techniques to the passing of large messages between processes in the same computer. The contents of Mach messages are typed – enabling port rights to be transmitted. Messages in Chorus and Amoeba are just contiguous sequences of bytes.

Network communication - Communication between processes in different computers is performed in Mach and Chorus by user-level network servers running at every computer. The network servers are also responsible for locating ports. The Amoeba kernel supports network communication directly and is highly tuned to achieve rapid request-reply interactions over a LAN. The Amoeba designers considered that the extra context switching costs that would be incurred through use of a separate, user-level network manager process would be prohibitive, Mach, Chorus and Amoeba use similar schemes for locating ports. Location hints are used first but if those fail they resort to broadcasting. The use of user-level network servers should allow for a variety of protocols. However, Mach's network servers primarily use TCP/IP as the transport protocol. Chorus also sticks to international standards, providing Internet and OSI protocols. However, this is not necessarily suitable on LANs when request-reply interactions predominate.

Memory management - Amoeba has very simple memory management scheme without virtual memory. In contrast Mach and Chorus provide similar very powerful and flexible virtual memory management schemes allowing a variety of different ways of sharing between processes, including copy-on-write. Mach and Chorus both make use of external pagers and local caches, which allow virtual memory to be shared between processes, even when they run in different computers. UNIX emulation ◊ Amoeba does not provide binary compatibility with UNIX. Chorus and Mach both provide emulation of UNIX as subsystems.