TULSIRAMJI GAIKWAD-PATIL College of Engineering & Technology, Nagpur

Department of Information Technology

Unit wise notes

Academic Session: 2018-2019

Subject: Distributed System **Semester: VIII**

UNIT 1

Introduction to distributed systems

Distributed System: Definition

A distributed system is a piece of software that ensures that: *A collection of independent*

Computers that appears to its users as a single coherent system

Two aspects:

(1) Independent computers and

(2) Single system _ **middleware**.

Distributed computing is a field of computer science that studies distributed systems. A **distributed system** consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a **distributed program**, and **distributed programming** is the process of writing such programs.

Introduction:

The word *distributed* in terms such as "distributed system", "distributed programming", and "distributed algorithm" originally referred to computer networks where individual computers were physically distributed within some geographical area. The terms are nowadays used in a much wider sense, even when referring to autonomous processes that run on the same physical computer and interact with each other by message passing. While there is no single definition of a distributed system, the following defining properties are commonly used:

There are several autonomous computational entities, each of which has its own local memory.

The entities communicate with each other by message passing. In this article, the computational entities are called *computers* or *nodes*. A distributed system may have a common goal, such as solving a large computational problem. Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users. Other typical properties of distributed systems include the following:

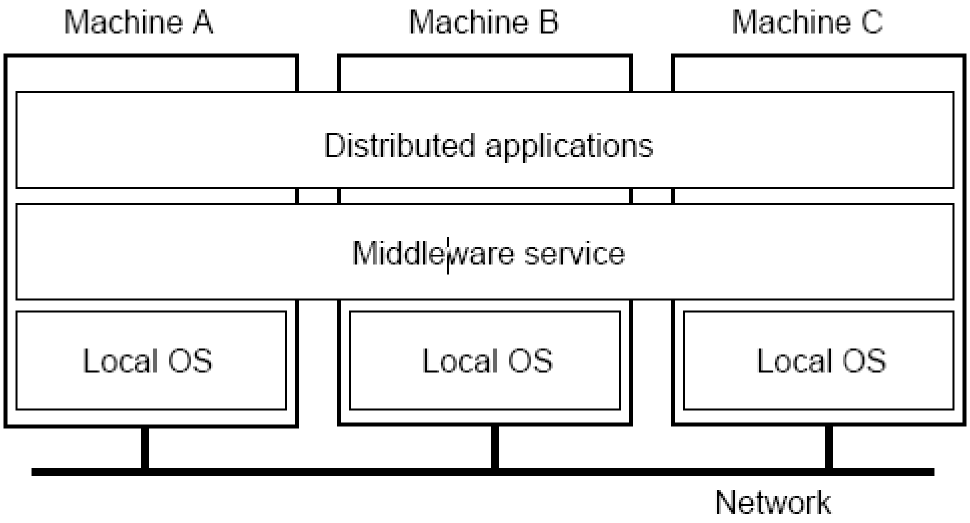
The system has to tolerate failures in individual computers.

The structure of the system (network topology, network latency, number of computers) is not known in advance, the system may consist of different kinds of computers and network links, and the system may change during the execution of a distributed program.

Each computer has only a limited, incomplete view of the system. Each computer

may know only one part of the input.

Architecture for Distributed System:



Various hardware and software architectures are used for distributed computing. At a lower level, it is necessary to interconnect multiple CPUs with some sort of network, regardless of whether that network is printed onto a circuit board or made up of loosely coupled devices and cables. At a higher level, it is necessary to interconnect processes running on those CPUs with some sort of communication system. Distributed programming typically falls into one of several basic architectures or categories: client–server, 3-tier architecture, *n*-tier architecture, distributed objects, loose coupling, or tight coupling.

Client–server: Smart client code contacts the server for data then formats and displays it to the user. Input at the client is committed back to the server when it represents a permanent change.

3-tier architecture: Three tier systems move the client intelligence to a middle tier so that stateless clients can be used. This simplifies application deployment. Most web applications are 3-Tier.

n-tier architecture: *n*-tier refers typically to web applications which further forward their requests to other enterprise services. This type of application is the one most responsible for the success of application servers.

Tightly coupled (clustered): refers typically to a cluster of machines that closely work together, running a shared process in parallel. The task is subdivided in parts that are made individually by each one and then put back together to make the final result.

Peer-to-peer: an architecture where there is no special machine or machines that provide a service or manage the network resources. Instead all responsibilities are uniformly divided among all machines, known as peers. Peers can serve both as clients and servers.

Space based: refers to an infrastructure that creates the illusion (virtualization) of one single address-space. Data are transparently replicated according to application needs. Decoupling in time, space and reference is achieved. Another basic aspect of distributed computing architecture is the method of communicating and coordinating work among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master/slave relationship. Alternatively, a "database-centric" architecture can enable distributed computing to be done without any form of direct inter process communication, by utilizing a shared database.

Goals of Distributed system

- Connecting resources and users
- Distribution transparency
- Openness
- Scalability

Hardware and Software concepts

Hardware Concepts

1. Multiprocessors
2. Multi computers
3. Networks of Computers

DistinguishingFeatures:

- Private versus shared memory
- Bus versus switched interconnection

Networks of Computers

High degree of node heterogeneity:

- High-performance parallel systems (multiprocessors as well as multicomputers)
- High-end PCs and workstations (servers)
- Simple network computers (offer users only network Access)
- Mobile computers (palmtops, laptops)
- Multimedia workstations

High degree of network heterogeneity:

- Local-area gigabit networks
- Wireless connections
- Long-haul, high-latency connections
- Wide-area switched megabit connections

Observation: Ideally, a distributed system hides these Differences

Software Concepts

- Distributed operating system
- Network operating system
- Middleware

Distributed Computing Model

Many tasks that we would like to automate by using a computer are of question–answer type: we would like to ask a question and the computer should produce an answer. In theoretical computer science, such tasks are called computational problems. Formally, a computational problem consists of *instances* together with a *solution* for each instance. Instances are questions that we can ask, and solutions are desired answers to these questions. Theoretical computer science seeks to understand which computational problems can be solved by using a computer (computability theory) and how efficiently (computational complexity theory). Traditionally, it is said that a problem can be solved by using a computer if we can design an algorithm that produces a correct solution for any given instance. Such an algorithm can be implemented as a computer program that runs on a general-purpose computer: the program reads a problem instance from input, performs some computation, and produces the solution as output. Formalisms such as random access machines or universal Turing machines can be used as abstract models of a

Sequential general-purpose computer executing such an algorithm. The field of concurrent and distributed computing studies similar questions in the case of either multiple computers, or a computer that executes a network of interacting processes: which computational problems can be solved in such a network and how efficiently? However, it is not at all obvious what is meant by "solving a problem" in the case of a concurrent or distributed system: for example, what is the task of the algorithm designer, and what is the concurrent and/or distributed equivalent of a sequential general-purpose computer?

The discussion below focuses on the case of multiple computers, although many of the issues are the same for concurrent processes running on a single computer. Three viewpoints are commonly used:

Parallel algorithms in shared-memory model

□□ All computers have access to a shared memory. The algorithm designer chooses the program executed by each computer.

□□ One theoretical model is the parallel random access machines (PRAM) are used. However, the classical PRAM model assumes synchronous access to the shared memory.

□□ A model that is closer to the behavior of real-world multiprocessor machines and takes into account the use of machine instructions such as Compare-and-swap (CAS) is that of *asynchronous shared memory*. There is a wide body of work on this model, a summary of which can be found in the literature.

Parallel algorithms in message-passing model

□□ The algorithm designer chooses the structure of the network, as well as the program executed by each computer.

□□ Models such as Boolean circuits and sorting networks are used. A Boolean circuit can be seen as a computer network: each gate is a computer that runs an extremely simple computer program. Similarly, a sorting network can be seen as a computer network: each comparator is a computer.

Distributed algorithms in message-passing model

□□ The algorithm designer only chooses the computer program. All computers run the same program. The system must work correctly regardless of the structure of the network.

□□ A commonly used model is a graph with one finite-state machine per node. In the case of distributed algorithms, computational problems are typically related to graphs. Often the graph that describes the structure of the computer network *is* the problem instance. This is illustrated in the following example.

An example

Consider the computational problem of finding a coloring of a given graph *G*. Different fields might take the following approaches:

Centralized algorithms

The graph *G* is encoded as a string, and the string is given as input to a computer. The computer program finds a coloring of the graph, encodes the coloring as a string, and outputs the result.

Parallel algorithms

□□ Again, the graph *G* is encoded as a string. However, multiple computers can access the same string in parallel. Each computer might focus on one part of the graph and produce a coloring for that part.

□□ The main focus is on high-performance computation that exploits the processing power of multiple computers in parallel.

Distributed algorithms

□□ The graph *G* is the structure of the computer network. There is one computer for each node of *G* and one communication link for each edge of *G*. Initially, each computer only knows about its immediate neighbors in the graph *G*; the computers must exchange messages with each other to discover more about the structure of *G*. Each computer must produce its own colour as output.

□□ The main focus is on coordinating the operation of an arbitrary distributed system. While the field of parallel algorithms has a different focus than the field of distributed algorithms, there is a lot of interaction between the two fields. For example, the Cole– Vishkin algorithm for graph colouring was originally presented as a parallel algorithm, but the same technique can also be used directly as a distributed algorithm. Moreover, a parallel algorithm can be implemented either in a parallel system (using shared memory) or in a distributed system (using message passing). The traditional boundary between parallel and distributed algorithms (choose a suitable network vs. run in any given network) does not lie in the same place as the boundary between parallel and distributed systems (shared memory vs. message passing).

Advantages & Disadvantage distributed system

1: Incremental growth: Computing power can be added in small increments

2: Reliability: If one machine crashes, the system as a whole can still survive

3: Speed: A distributed system may have more total computing power than a mainframe

4: Open system: This is the most important point and the most characteristic point of a distributed system. Since it is an open system it is always ready to communicate with other systems. An open system that scales has an advantage over a perfectly closed and self-contained system.

5. Economic: Microprocessors offer a better price/performance than mainframes

Disadvantages of Distributed Systems over Centralized ones

1:As it is previously told you distributed systems will have an inherent security issue.

2:Networking: If the network gets saturated then problems with transmission will surface.

3:Software:There is currently very little less software support for Distributed system.

4:Troubleshooting: Troubleshooting and diagnosing problems in a distributed system can also become more difficult, because the analysis may require connecting to remote nodes or inspecting communication between nodes.

Issues in designing Distributed System

 Secure communication over public networks ACI: who sent it, did anyone see it, did anyone change it

 Fault-tolerance : Building reliable systems from unreliable components nodes fail independently; a distributed system can "partly fail" [Lamport]: "A distributed system is one in which the failure of a machine I've never heard of can prevent me from doing my work." Replication, caching, naming Placing data and computation for effective resource sharing, hiding latency, and finding it again once you put it somewhere. Coordination and shared state What should the system components do and when should they do it? Once they've all done it, can they all agree on what they did and when?

Synchronization

Clock synchronization is a problem from computer science and engineering which deals with the idea that internal clocks of several computers may differ. Even when initially set accurately, real clocks will differ after some amount of time due to clock drift, caused by clocks counting time at slightly different rates. There are several problems that occur as a repercussion of rate differences and several solutions, some being more appropriate than others in certain contexts. In serial communication, some people use the term "clock synchronization" merely to discuss getting one metronome-like clock signal to pulse at the same frequency as another one frequency synchronization and phase synchronization. Such "clock synchronization" is used in synchronization in telecommunications and automatic baud rate detection.

Process scheduling

Preemptive scheduling is widespread in operating systems and in parallel processing on symmetric multiprocessors. However, in distributed systems it is practically unheard of. Scheduling in distributed systems is an important issue, and has performance impact on parallel processing, load balancing and meta computing. Non-preemptive scheduling can perform well if the task lengths and processor speeds are known in advance and hence job placement is done intelligently

Deadlock handling

Deadlocks in Distributed Systems: Deadlocks in Distributed Systems Deadlocks in distributed systems are similar to deadlocks in single processor systems, only worse. They are harder to avoid, prevent or even detect. They are hard to cure when tracked down because all relevant information is scattered over many machines. People sometimes might classify deadlock into the following types: Communication deadlocks -- competing with buffers for send/receive Resources deadlocks -- exclusive access on I/O devices, files, locks, and other resources. We treat everything as resources; there we only have resources deadlocks. Four best-known strategies to handle deadlocks: The ostrich algorithm (ignore the problem) Detection (let deadlocks occur, detect them, and try to recover) Prevention (statically make deadlocks structurally impossible) Avoidance (avoid

deadlocks by allocating resources carefully)

The FOUR Strategies for handling deadlocks : The FOUR Strategies for handling deadlocks The ostrich algorithm No dealing with the problem at all is as good and as popular in distributed systems as it is in single-processor systems. In distributed systems used for programming, office automation, process control, no system-wide deadlock mechanism is present -- distributed databases will implement their own if they need one. Deadlock detection and recovery is popular because prevention and avoidance are so difficult to implement. Deadlock prevention is possible because of the presence of atomic transactions. We will have two algorithms for this. Deadlock avoidance is never used in distributed system, in fact, it is not even used in single processor systems. The problem is that the banker's algorithm need to know (in advance) how much of each resource every process will eventually need. This information is rarely, if ever, available. Hence, we will just talk about deadlock detection and deadlock prevention.

Load Balancing

Resource Scheduling (RS)

RS continuously monitors utilization across resource pools and intelligently aligns resources with business needs, enabling you to:

□□ Dynamically allocate IT resources to the highest priority applications. Create rules and policies to prioritize how resources are allocated to virtual machines. □□Give IT autonomy to business organizations. Provide dedicated IT infrastructure to business units while still achieving higher hardware utilization through resource pooling.

□□ Empower business units to build and manage virtual machines within their resource pool while giving central IT control over hardware resources.

File Sharing

File sharing is the practice of distributing or providing access to digitally stored information, such as computer programs, multi-media (audio, video), documents, or electronic books. It may be implemented through a variety of storage, transmission, and distribution models and common methods of file sharing incorporate manual sharing using removable media, centralized computer file server installations on computer networks, World Wide Web-based hyper linked documents, and the use of distributed peer-to-peer (P2P) networking. The Distributed File System is used to build a hierarchical view of multiple file servers and shares on the network. Instead of having to think of a specific machine name for each set of files, the user will only have to remember one name; which will be the 'key' to a list of shares found on multiple servers on the network. Think of it as the home of all file shares with links that point to one or more servers that actually host those shares. DFS has the capability of routing a client to the closest available file server by using Active

Directory site metrics. It can also be installed on a cluster for even better performance and reliability. Medium to large sized organizations are most likely to benefit from the use of DFS - for smaller companies it is simply not worth setting up since an ordinary file server would be just fine.

Concurrency Control

In computer science, especially in the fields of computer programming (see also concurrent programming, parallel programming), operating systems (see also parallel computing), multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible. **Distributed concurrency control** is the concurrency control of a system distributed over a computer network.

Failure handling

In a distributed system, **failure transparency** refers to the extent to which errors and subsequent recoveries of hosts and services within the system are invisible to users and applications. For example, if a server fails, but users are automatically redirected to another server and never notice the failure, the system is said to exhibit *high failure transparency*.

Failure transparency is one of the most difficult types of transparency to achieve since it

is often difficult to determine whether a server has actually failed, or whether it is simply responding very slowly. Additionally, it is generally impossible to achieve full failure transparency in a distributed system since networks are unreliable.

Configuration

Dynamic system configuration is the ability to modify and extend a system while it is running. The facility is a requirement in large distributed systems where it may not be possible or economic to stop the entire system to allow modification to part of its hardware or software. It is also useful during production of the system to aid incremental integration of component parts, and during operation to aid system evolution.

UNIT III

In computer networking, an **Internet socket** or **network socket** is an endpoint of a bidirectional inter-process communication flow across an Internet Protocol based computer network, such as the Internet. The term *Internet sockets* is also used as a name for an application programming interface (API) for the TCP/IP protocol stack, usually provided by the operating system. Internet sockets constitute a mechanism for delivering incoming data packets to the appropriate application process or thread, based on a combination of local and remote IP addresses and port numbers. Each socket is mapped by the operating system to a communicating application process or thread.

A **socket address** is the combination of an IP address (the location of the computer) and a port (which is mapped to the application program process) into a single identity, much like one end of a telephone connection is the combination of a phone number and a particular extension. An Internet socket is characterized by a unique combination of the following:

□□ **Protocol:** A transport protocol (e.g., TCP, UDP), raw IP, or others. TCP port 53 and UDP port 53 are different, distinct sockets.

□□ **Local socket address:** Local IP address and port number

□□ **Remote socket address:** Only for established TCP sockets. As discussed in the Client-Server section below, this is necessary since a TCP server may serve several clients concurrently. The server creates one socket for each client, and these sockets share the same local socket address.

Sockets are usually implemented by an API library such as Berkeley sockets, first introduced in 1983. Most implementations are based on Berkeley sockets, for example Winsock introduced in 1991. Other socket API implementations exist, such as the STREAMS-based Transport Layer Interface (TLI). Development of application programs that utilize this API is called socket programming or network programming. These are examples of functions or methods typically provided by the API library:

 socket() creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.

 bind() is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.

 listen() is used on the server side, and causes a bound TCP socket to enter listening state.

 connect() is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

 accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.

 send() and recv(), or write() and read(), or recvfrom() and sendto(), are used for sending and receiving data to/from a remote socket.

 close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

 gethostbyname() and gethostbyaddr() are used to resolve host names and addresses.

 select() is used to prune a provided list of sockets for those that are ready to read, ready to write or have errors

 poll() is used to check on the state of a socket. The socket can be tested to see if it can be written to, read from or has errors.

Data Representation & Marshaling

In computer science, **marshalling** (similar to serialization) is the process of transforming the memory representation of an object to a data format suitable for storage or transmission. It is typically used when data must be moved between different parts of a computer program or from one program to another. The opposite, or reverse, of marshalling is called *unmarshalling* (or *demarshalling*, similar to *deserialization*).

Group Communication

Computer systems consisting of multiple processors are becoming commonplace. Many companies and institutions, for example, own a collection of workstations connected by a local area network (LAN). Although the hardware for distributed computer systems is advanced, the software has many problems. We believe that one of the main problems is the communication paradigms that are used. This thesis is concerned with software for distributed computer systems. In it, we will study an abstraction, called *group communication* that simplifies building reliable efficient distributed systems. We will discuss a design for group communication, show that it can be implemented efficiently, and describe the design and implementation of applications based on group communication. Finally, we will give extensive performance measurements. Our goal is to demonstrate that group communication is a suitable abstraction for distributed systems.

Client Server Communication

Client–server model of computing is a distributed application structure that partitions tasks or workloads between service providers, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share its resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await (*listen* for) incoming requests.

RPC- Implementing RPC Mechanism

In computer science, a **remote procedure call** (**RPC**) is an Inter-process communication that allows a computer program to cause a subroutine or 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. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object oriented principles, RPC is called **remote invocation** or **remote method invocation**.

The steps in making a RPC

1. The client calling the Client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.

2. The client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called marshaling.

3. The kernel sending the message from the client machine to the server machine.

4. The kernel passing the incoming packets to the server stub.

5. Finally, the server stub calling the server procedure. The reply traces the same in other direction

Stub Generation, RPC Messages.

Following figure illustrates the basic operation of RPC. A client application issues a normal procedure call to a *client stub*. The client stub receives arguments from the calling procedure and returns arguments to the calling procedure. An argument may instantiate an input parameter, an output parameter, or an input/output parameter. In the discussion of this Section, the term *input argument* refers to a parameter which may be either an input parameter or an input/output parameter, and the term *output argument* refers to either an output parameter or an input/output parameter. The client stub converts the input arguments from the local data representation to a

common data representation, creates a message containing the input arguments in their common data representation, and calls the client runtime, usually an object library of routines that supports the functioning of the client stub. The client runtime transmits the message with the input arguments to the server runtime which is usually an object library that supports the functioning of the server stub. The server runtime issues a call to the *server stub* which takes the input arguments from the message, converts them from the common data representation to the local data representation of the server, and calls the server application which does the processing. When the server application has completed, it returns to the server stub the results of the processing in the output arguments. The server stub converts the output arguments from the data representation of the server to the common data representation for transmission on the network and encapsulates the output arguments into a message which is passed to the server runtime. The server runtime transmits the message to the client runtime which passes the message to the client stub. Finally, the client stub extracts the arguments from the message and returns them to the calling procedure in the required local data representation.

Synchronization: - Clock Synchronization

Clock synchronization is a problem from computer science and engineering which deals with the idea that internal clocks of several computers may differ. Even when initially set accurately, real clocks will differ after some amount of time due to clock drift, caused by clocks counting time at slightly different rates. There are several problems that occur as a repercussion of rate differences and several solutions, some being more appropriate than others in certain contexts. In a centralized system the solution is trivial; the centralized server will dictate the system time. Cristian's algorithm and the Berkeley Algorithm are some solutions to the clock synchronization problem in a centralized server environment. In a distributed system the problem takes on more complexity because a global time is not easily known. The most used clock synchronization solution on the Internet is the Network Time Protocol (NTP) which is a layered client-server architecture based on UDP message passing. Lamport timestamps and Vector clocks are concepts of the logical clocks in distributed systems.

Cristian's algorithm

Cristian's algorithm relies on the existence of a time server. The time server maintains its clock by using a radio clock or other accurate time source, then all other computers in the system stay synchronized with it. A time client will maintain its clock by making a procedure call to the time server. Variations of this algorithm make more precise time calculations by factoring in network propagation time.

Berkeley algorithm

This algorithm is more suitable for systems where a radio clock is not present, this system has no way of making sure of the actual time other than by maintaining a global average time as the global time. A time server will periodically fetch the time from all the time clients, average the results, and then report back to the clients the adjustment that needs be made to their local clocks to achieve the average. This algorithm highlights the fact that internal clocks may vary not only in the time they contain but also in the clock rate. Often, any client whose clock differs by a value outside of a given tolerance is disregarded when averaging the results. This prevents the overall system time from being drastically skewed due to one erroneous clock.

Network Time Protocol

This algorithm is a class of mutual network synchronization algorithm in which no master or reference clocks are needed. All clocks equally participate in the synchronization of the network by exchanging their timestamps using regular beacon packets. CS-MNS is suitable for distributed and mobile applications. It has been shown to be scalable, accurate in the order of few microseconds, and compatible to IEEE 802.11 and similar standards.

Reference broadcast synchronization

This algorithm is often used in wireless networks and sensor networks. In this scheme, an initiator broadcasts a reference message to urge the receivers to adjust their clocks.

Mutual Exclusion,

Assumptions

The system consists of *n* processes; each process *Pi* resides at a different processor Each process has a critical section that requires mutual exclusion Basic Requirement If *Pi* is executing in its critical section, then no other process *Pj* is executing in its critical section The presented algorithms ensure the mutual exclusion execution of processes in their critical sections. Mutual exclusion must be enforced: only one process at a time is allowed in its critical section A process that hales in its non critical section must do so without interfering with other processes. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: No deadlock or starvation When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay No assumptions are made about relative process speeds or number of processors A process remains inside its critical section for a finite time Only

Election Algorithms:- Bully & Ring Algorithms

□□ There are at least two basic strategies by which a distributed system can adapt to failures.

□□ Operate continuously as failures occur and are repaired

□□ The second alternative is to temporarily halt normal operation and to take some time out to reorganize the system.

□□ The reorganization of the system is managed by a single node called the coordinator.

□□So as a first step in any reorganization, the operating or active nodes must elect a coordinator.

□□ Similar

□□ Like Synchronization, all processors must come to an agreement about who enters the critical region (i.e. who is the leader)

□□ Different

□□ The election protocol must properly deal with the case of a coordinator failing. On the other hand, mutual exclusion algorithms assume that the process in the critical region (i.e., the coordinator) will not fail.

□□ A new coordinator must inform all active nodes that it is the coordinator. In a mutual exclusion algorithm, the nodes not in the critical region have no need to know what node is in the region.

 The two classical election algorithms by Garcia-Molina

 Bully Algorithm

 Invitation Algorithm

 Ring Algorithm

Election algorithms

We often need one process to act as a coordinator. It may not matter which process does this, but there should be group agreement on only one. An assumption in election algorithms is that all processes are exactly the same with no distinguishing characteristics. Each process can obtain a unique identifier (for example, a machine address and process ID) and each process knows of every other process but does not know which is up and which is down.

Bully algorithm

The bully algorithm selects the process with the largest identifier as the coordinator. It works as follows:

1. When a process *p* detects that the coordinator is not responding to requests, it initiates an election:

a. *p* sends an *election* message to all processes with higher numbers.

b. If nobody responds, then *p* wins and takes over.

c. If one of the processes answers, then *p*'s job is done.

2. If a process receives an *election* message from a lower-numbered process at any time, it:

a. sends an OK message back.

b. holds an election (unless its already holding one).

3. A process announces its victory by sending all processes a message telling them that it is the new coordinator.

4. If a process that has been down recovers, it holds an election.

Ring algorithm

The ring algorithm uses the same ring arrangement as in the token ring mutual exclusion algorithm, but does not employ a token. Processes are physically or logically ordered so that each knows its successor.

 If any process detects failure, it constructs an *election* message with its process I.D. (e.g. network address and local process I.D.) and sends it to its successor.

 If the successor is down, it skips over it and sends the message to the next party. This process is repeated until a running process is located.

 At each step, the process adds its own process I.D. to the list in the message. Eventually, the message comes back to the process that started it:

1. The process sees its ID in the list.

2. It changes the message type to *coordinator*.

3. The list is circulated again, with each process selecting the highest numbered ID in the list to act as coordinator.

4. When the *coordinator* message has circulated fully, it is deleted. Multiple messages may circulate if multiple processes detected failure. This creates a bit of overhead but produces the same results.

Unit V

Distributed Shared Memory (DSM), also known as a **distributed global address space** (**DGAS**), is a term in computer science that refers to a wide class of software and hardware implementations, in which each node of a cluster has access to shared memory in addition to each node's non-shared private memory.

Software DSM systems can be implemented in an operating system, or as a programming library. Software DSM systems implemented in the operating system can be thought of as extensions of the underlying virtual memory architecture. Such systems are transparent to the developer; which means that the underlying distributed memory is completely hidden from the users. In contrast, Software DSM systems implemented at the library or language level are not transparent and developers usually have to program differently. However, these systems offer a more portable approach to DSM system implementation. Software DSM systems also have the flexibility to organize the shared memory region in different ways. The page based approach organizes shared memory into pages of fixed size. In contrast, the object based approach organizes the shared memory region as an abstract space for storing shareable objects of variable sizes. Other commonly seen implementation uses tuple space, in which unit of sharing is a tuple.

Shared memory architecture may involve separating memory into shared parts distributed amongst nodes and main memory; or distributing all memory between nodes. A coherence protocol, chosen in accordance with a consistency model, maintains memory coherence.

Examples of such systems include:

 Delphi DSM

 JIAJIA

 Kerrighed

 NanosDSM

 OpenSSI

 MOSIX

 Strings

 Terracotta

 TreadMarks

 DIPC

 Intel Cluster OpenMP is internally a Software DSM.

 ScaleMP ?

 RNA networks

DSM Architecture & its Types,

 DSM Subsystem

 DSM Server

 KEY Server

DSM Subsystem

 Routines to handle page faults relating to virtual addresses corresponding to a DSM region.

 Code to service system calls which allow a user process to get, attach and detach a DSM region.

 Code to handle system calls from the DSM server.

DSM Server

 In-server :Receives messages from remote DSM servers and takes appropriate action. (E.g. Invalidate its copy of a page)

 Out-server :Receives requests from the DSM subsystem and communicates with its peer DSM servers at remote nodes. Note that the DSM subsystem itself does

not directly communicate over the network with other hosts.

 Communication with key Server.

Key Sever

 Each region must be uniquely identifiable across the entire LAN. When a process executes system call with a key and is the first process at that host to do so, the key server is consulted.

 Key server‘s internal table is looked-up for the key, if not found then it stores the specified key in the table as a new entry.

Design & Implementations issues In DSM System

There are various factors that have to be kept in mind while designing and implementing the DSM systems. They are as follows:

1.Block Size:

As we know, transfer of the memory blocks is the major operation in the DSM systems. Therefore block size matters a lot here. Block size is often referred to as the **Granularity**. Block size is the unit of sharing or unit of data transfer in the event of network block fault. Block size can be few words, pages or few pages. Size of the block depends on various factors like, paging overhead, thrashing, false sharing, and directory size.

2. Structure of Shared Memory Space:

How the shared memory space is organized with data determines the structure of the shared memory space. It refers to the layout of shared data. It depends upon the type of application the DSM is going to handle.

3. Replacement Strategy:

It may happen that one node might be accessing for a memory block from DSM when its own local memory is completely full. In such a case, when the memory block migrating from remote node reaches, it finds no space to get placed. Thus a replacement strategy of major concern in the design and implementation of the DSM systems. Certain block must be removed so as to place the new blocks in such a situation. Several techniques are used for the replacement of old blocks such as removal of **Least Recently Used** memory blocks.

4. Thrashing:

Sometimes two or more processes might access the same memory block. Suppose two processes need to perform write operation on the same memory block. Since, to accomplish this, the block has to migrate in both directions at a very small interval of time, so it will be transferred back and forth at such a high rate that none of the two processes will be able to perform the operation accurately and completely. As such no real work is done. This condition is called thrashing. A technique should be incorporated, while designing the

DSM systems to avoid thrashing.

5. Heterogeneity:

DSM systems should be able to function on computers with different architectures. Issues Involved in

DSM Issues

- Network Communication
- Consistency
- Data Granularity
- Coherence

Consistency Model

- Strict consistency in shared memory systems.
- Sequential consistency in shared memory systems –Our focus.
- Other consistency protocols
- Casual consistency protocol.
- Weak and release consistency protocol

Unit VI

Desirable features of good Distributed File System

In computing, a **distributed file system** or **network file system** is any file system that allows access to files from multiple hosts sharing via a computer network. This makes it possible for multiple users on multiple machines to share files and storage resources. The client nodes do not have direct access to the underlying block storage but interact over the network using a protocol. This makes it possible to restrict access to the file system depending on access lists or capabilities on both the servers and the clients, depending on how the protocol is designed. In contrast, in a shared disk file system all nodes have equal access to the block storage where the file system is located. On these systems the access control must reside on the client. Distributed file systems may include facilities for transparent replication and fault tolerance. That is, when a limited number of nodes in a file system go offline, the system continues to work without any data loss. The difference between a distributed file system and a distributed data store can be

vague, but DFSes are generally geared towards use on local area networks.

Features

DFS offers many features that make managing multiple file servers much simpler and effective.

 Unified Namespace

DFS links multiple shared folders on multiple servers into a folder hierarchy. This hierarchy is same as a physical directory structure on a single hard disk. However, in this case, the individual branch of the hierarchy can be on any of the participating servers.

 Location Transparency

Even if the files are scattered across multiple servers, users need to go to only one network location. This is a very powerful feature. Users do not need to know if the actual file location has changed. There is no need to inform everyone about using new paths or server names! Imagine how much time and energy this can save. It reduces downtime required during server renames, planned or unplanned shutdowns and so on.

 Continuous Availability

As mentioned, during planned shutdowns, the file resources can be temporarily made available from another standby server, without users requiring to be notified about it. This way downtime related to maintenance or disaster recovery tasks is completely eliminated. This is very useful especially in Web servers. The Web server file locations can be configured in such a way that even when the physical location of the files changes to another server, the HTML links continues to work without breaking.

 Replication

It is possible to replicate data to one or more servers within the DFS structure. This way, if one server is down, files will be automatically served from other replicated locations. What's more, users will not even know the difference.

 Load Balancing

This is a conceptual extension of replication feature. Now that you can put copies of the same file across multiple locations. If the file is requested by more than one user at the same time, DFS will serve it from different locations. This way, the load on one server is balanced across multiple servers, which increases performance. At a user level, they do not even come to know that the file came from a particular replica on DFS.

 Security

DFS utilises the same NTFS security and file sharing permissions. Therefore, no special configuration is required to integrate base security with DFS.

 Ongoing hard disk space management

What happens when your hard disk space is exhausted? You typically add another hard disk. Now, this hard disk will have another name. What if this disk is on another server? Things would get worse. With DFS, you can keep adding new directories to the namespace on completely separate servers. Users never have to bother about the physical server name. This way, you can grow your storage in steps without having to worry about destabilizing file access by users.

 Unifying heterogeneous platforms

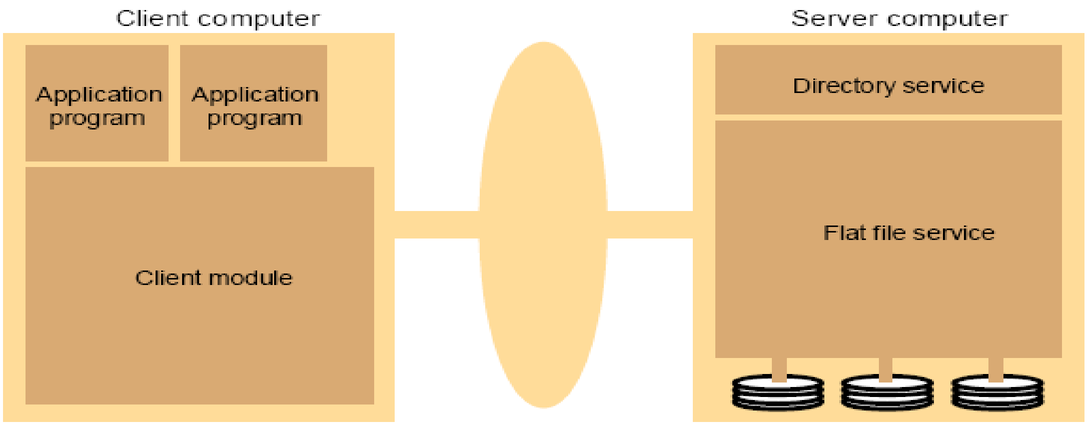
DFS also supports NetWare. This way, administrators can unify data access by combining servers running heterogeneous operating systems from a file access perspective.

 Fault tolerance or higher availability of data

DFS works with clustering services. This combination offers higher availability than just using clustering.

File Model

The Distributed File System is used to build a hierarchical view of multiple file servers and shares on the network. Instead of having to think of a specific machine name for each set of files, the user will only have to remember one name; which will be the 'key' to a list of shares found on multiple servers on the network. Think of it as the home of all file shares with links that point to one or more servers that actually host those shares. DFS has the capability of routing a client to the closest available file server by using Active Directory site metrics. It can also be installed on a cluster for even better performance and reliability. Medium to large sized organizations are most likely to benefit from the use of DFS - for smaller companies it is simply not worth setting up since an ordinary file server would be just fine.



File Service Architecture : Following figure shows the architecture of file service. Client computer can communicate with server using client module to flat file service.

File Sharing Semantics

Sequential semantics: Read returns result of last write

Easily achieved if

– Only one server

– Clients do not cache data

BUT

– Performance problems if no cache

• Obsolete data

– We can **write-through**

• Must notify clients holding copies

• Requires extra state, generates extra traffic

Session semantics

Relax the rules

• Changes to an open file are initially visible only to the process (or machine) that modified it.

• Last process to modify the file wins.

File Catching Scheme

File caching has implemented in several file system for centralized time-sharing systems to improve file I/O performance. The idea in file caching in there systems is to retain recently accessed file data in main memory, so that repeated accesses to the same information can be handled without additional disk transfers. Because of locality in file access patterns, file caching reduces disk transfers substantially, resulting in better overall performance of the file system. The property of locality in file access patterns can as well be exploited in distributed systems by designing a suitable file-caching scheme. In addition to better performance, a file-caching scheme for a distributed file system may also contribute to its scalability and reliability because it is possible to cache remotely located data on a client node. Therefore, every distributed file system in serious use today uses some form of file caching.

File Application & Fault tolerance

In engineering, **fault-tolerant design**, also known as **fail-safe design**, is a design that enables a system to continue operation, possibly at a reduced level (also known as graceful degradation), rather than failing completely, when some part of the system fails. The term is most commonly used to describe computer-based systems designed to continue more or less fully operational with, perhaps, a reduction in throughput or an increase in response time in the event of some partial failure. That is, the system as a whole is not stopped due to problems either in the hardware or the software. An example in another field is a motor vehicle designed so it will continue to be drivable

if one of the tires is punctured. Distributed fault-tolerant replication of data between nodes (between servers or servers/clients) for high availability and offline (disconnected) operation. Distributed file systems, which also are parallel and fault tolerant, stripe and replicate data over multiple servers for high performance and to maintain data integrity. Even if a server fails no data is lost. The file systems are used in both high-performance computing (HPC) and high-availability clusters.

Naming: - Features, System Oriented Names

Naming in distributed systems is modeled as a string translation problem. Viewing names as strings and name resolution mechanisms as syntax directed translators provides a formal handle on the loosely understood concepts associated with naming: we give precise definitions for such informal terminology as name spaces, addresses, routes, source-routing, and implicit-routing; we identify the properties of naming systems, including under what conditions they support unique names, relative names, absolute names, and synonyms; and we discuss how the basic elements of the model can be implemented by name servers. The resources in a distributed system are spread across different computers and a naming scheme has to be devised so that users can discover and refer to the resources that they need.

An identifier that:

• Identifies a resource

– Uniquely

– Describes the resource

• Enables us to locate that resource

– Directly

– With help

• Is it really an identifier

– Bijective, persistent

An example of such a naming scheme is the URL (Uniform Resource Locator) that is used to identify WWW pages. If a meaningful and universally understood identification scheme is not used then many of these resources will be inaccessible to system users.

□□ **Objects:**

• processes, files, memory, devices, processors, and networks.

□□ **Object access:**

• Each object associate with a defined access operation.

• Accesses via object servers

□□ **Identification of a server by**:

• Name

• Physical or Logical address

• Service that the servers provide.

□□ **Identification Issue:**

• Multiple server addresses may exist requiring a server to move requiring the name to be changed.

Object Locating Mechanism

Distributed systems based on objects are one of the newest and most popular approaches to the design and construction of distributed systems. CORBA platform is built from several standards published by the organization OMG (Object Management Group), whose objective is to provide a common system for the construction of distributed systems in a heterogeneous environment. The role of the ORB is to deliver the tasks the system the following services: network communication, locating objects, sending notifications too objects, the results to clients. The basic features of CORBA are: independence from the programming language by using language IDL and the independence of the system, hardware, communication (IIOP).

Java RMI (Remote Method Invocation) is a second example of a distributed system platform based on objects. RMI is a structure built based on Java. The model presupposes the existence of the facility, located in the address space of the server and client, which causes the object operations. Remote state of the object is located on a single machine, and the local interface of an object is released.

Human Oriented Name

Names allow **us** to identify objects. to talk about them and to access them. Naming is therefore an important issue for large scale distributed systems. It becomes a critical issue when those systems are intended to support collaboration between humans. **A** large volume of research has already been published on the subject of naming, particularly within the context of name servers and directories. However, it **can** be argued that the hierarchical nature ***of*** many of the naming mechanisms so far proposed is too constraining to fully support the great flexibility of human naming practice, particularly where group work is concerned.