**UNIT – I**

**LESSON 1: DISTRIBUTED SYSTEMS**

**CONTENTS**

1.0 Aim and Objectives

1. Introduction
2. Organization
3. Goals and Advantages
4. Disadvantages
5. Architecture
6. Concurrency
7. Languages

**1.0. AIM AND OBJECTIVES**

At the end of this Lesson you will be able to

understand the concept of Distributed Computing, organization of Distributed Computing,

advantages and limitations of Distributed Computing

1. **INTRODUCTION**

**Distributed computing** is a method of computer processing in which differentparts of a program are run simultaneously on two or more computers that are communicating with each other over a network. Distributed computing is a type of **segmented** or parallel computing, but the latter term is most commonly used to refer toprocessing in which different parts of a program run simultaneously on two or more processors that are part of the same computer. While both types of processing require that a program be segmented—divided into sections that can run simultaneously, distributed computing also requires that the division of the program take into account the different environments on which the different sections of the program will be running. For example, two computers are likely to have different file systems and different hardware components.

An example of distributed computing is BOINC, a framework in which large problems can be divided into many small problems which are distributed to many computers. Later, the small results are reassembled into a larger solution.

1

Distributed computing is a natural result of using networks to enable computers to communicate efficiently. But distributed computing is distinct from computer networking or **fragmented** computing. The latter refers to two or more computers interacting with each other, but not, typically, sharing the processing of a single program. The World Wide Web is an example of a network, but not an example of distributed computing.

There are numerous technologies and standards used to construct distributed computations, including some which are specially designed and optimized for that purpose, such as Remote Procedure Calls (RPC) or Remote Method Invocation (RMI) or

.NET Remoting.

**1.2. ORGANIZATION**

Organizing the interaction between each computer is of prime importance. In order to be able to use the widest possible range and types of computers, the protocol or communication channel should not contain or use any information that may not be understood by certain machines. Special care must also be taken that messages are indeed delivered correctly and that invalid messages are rejected which would otherwise bring down the system and perhaps the rest of the network.

Another important factor is the ability to send software to another computer in a portable way so that it may execute and interact with the existing network. This may not always be possible or practical when using differing hardware and resources, in which case other methods must be used such as cross-compiling or manually porting this software.

**1.3. GOALS AND ADVANTAGES**

There are many different types of distributed computing systems and many challenges to overcome in successfully designing one. The main goal of a distributed computing system is to connect users and resources in a transparent, open, and scalable way. Ideally this arrangement is drastically more fault tolerant and more powerful than many combinations of stand-alone computer systems.

**Openness**

Openness is the property of distributed systems such that each subsystem is continually open to interaction with other systems (see references). Web Services protocols are standards which enable distributed systems to be extended and scaled. In general, an open system that scales has an advantage over a perfectly closed and self-contained system.

Consequently, open distributed systems are required to meet the following challenges:

**Monotonicity**

Once something is published in an open system, it cannot be taken back.

2

**Pluralism**

Different subsystems of an open distributed system include heterogeneous, overlapping and possibly conflicting information. There is no central arbiter of truth in open distributed systems.

**Unbounded nondeterminism**

Asynchronously, different subsystems can come up and go down and communication links can come in and go out between subsystems of an open distributed system. Therefore the time that it will take to complete an operation cannot be bounded in advance.

**1.4. DISADVANTAGES**

**Technical issues**

If not planned properly, a distributed system can decrease the overall reliability of computations if the unavailability of a node can cause disruption of the other nodes. Leslie Lamport famously quipped that: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

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.

Many types of computation are not well suited for distributed environments, typically owing to the amount of network communication or synchronization that would be required between nodes. If bandwidth, latency, or communication requirements are too significant, then the benefits of distributed computing may be negated and the performance may be worse than a non-distributed environment.

**Project-related problems**

Distributed computing projects may generate data that is proprietary to private industry, even though the process of generating that data involves the resources of volunteers. This may result in controversy as private industry profits from the data which is generated with the aid of volunteers. In addition, some distributed computing projects, such as biology projects that aim to develop thousands or millions of "candidate molecules" for solving various medical problems, may create vast amounts of raw data. This raw data may be useless by itself without refinement of the raw data or testing of candidate results in real-world experiments. Such refinement and experimentation may be so expensive and time-consuming that it may literally take decades to sift through the data. Until the data is refined, no benefits can be acquired from the computing work.

Other projects suffer from lack of planning on behalf of their well-meaning originators. These poorly planned projects may not generate results that are palpable, or may not generate data that ultimately result in finished, innovative scientific papers. Sensing that a project may not be generating useful data, the project managers may

3

decide to abruptly terminate the project without definitive results, resulting in wastage of the electricity and computing resources used in the project. Volunteers may feel disappointed and abused by such outcomes. There is an obvious opportunity cost of devoting time and energy to a project that ultimately is useless, when that computing power could have been devoted to a better planned distributed computing project generating useful, concrete results.

Another problem with distributed computing projects is that they may devote resources to problems that may not ultimately be soluble, or to problems that are best pursued later in the future, when desktop computing power becomes fast enough to make pursuit of such solutions practical. Some distributed computing projects may also attempt to use computers to find solutions by number-crunching mathematical or physical models. With such projects there is the risk that the model may not be designed well enough to efficiently generate concrete solutions. The effectiveness of a distributed computing project is therefore determined largely by the sophistication of the project creators.-

**1.5. ARCHITECTURE**

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 set of highly integrated machines that run the same process in parallel, subdividing the task 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.

4

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.

**1.6. CONCURRENCY**

Distributed computing implements a kind of concurrency. It interrelates tightly with concurrent programming so much that they are sometimes not taught as distinct subjects.

**Multiprocessor systems**

A multiprocessor system is simply a computer that has >1 & not <=1 CPU on its motherboard. If the operating system is built to take advantage of this, it can run different processes (or different threads belonging to the same process) on different CPUs.

**Multicore systems**

Intel CPUs from the late Pentium 4 era (Northwood and Prescott cores) employed a technology called Hyperthreading that allowed more than one thread (usually two) to run on the same CPU. The more recent Sun UltraSPARC T1, AMD Athlon 64 X2, AMD Athlon FX, AMD Opteron, Intel Pentium D, Intel Core, Intel Core 2 and Intel Xeon processors feature multiple processor cores to also increase the number of concurrent threads they can run.

**Multicomputer systems**

A multicomputer may be considered to be either a loosely coupled NUMA computer or a tightly coupled cluster. Multicomputers are commonly used when strong compute power is required in an environment with restricted physical space or electrical power.

Common suppliers include Mercury Computer Systems, CSPI, and SKY Computers.

Common uses include 3D medical imaging devices and mobile radar.

5

**Computing taxonomies**

The types of distributed systems are based on Flynn's taxonomy of systems; single instruction, single data (SISD), single instruction, multiple data (SIMD), multiple instruction, single data (MISD), and multiple instruction, multiple data (MIMD). Other taxonomies and architectures available at Computer architecture and in Category:Computer architecture.

**Computer clusters**

A cluster consists of multiple stand-alone machines acting in parallel across a local high speed network. Distributed computing differs from cluster computing in that computers in a distributed computing environment are typically not exclusively running "group" tasks, whereas clustered computers are usually much more tightly coupled. Distributed computing also often consists of machines which are widely separated geographically.

**Grid computing**

A grid uses the resources of many separate computers, loosely connected by a network (usually the Internet), to solve large-scale computation problems. Public grids may use idle time on many thousands of computers throughout the world. Such arrangements permit handling of data that would otherwise require the power of expensive supercomputers or would have been impossible to analyze.

**1.7. LANGUAGES**

Nearly any programming language that has access to the full hardware of the system could handle distributed programming given enough time and code. Remote procedure calls distribute operating system commands over a network connection. Systems like CORBA, Microsoft DCOM, Java RMI and others, try to map object oriented design to the network. Loosely coupled systems communicate through intermediate documents that are typically human readable (e.g. XML, HTML, SGML, X.500, and EDI).

Languages specifically tailored for distributed programming are:

Ada programming language Alef programming language E programming language

Erlang programming language Limbo programming language Oz programming language

ZPL (programming language) Orca programming language

**LESSON 2: DESIGNING OF DISTRIBUTED SYSTEMS**

**CONTENTS**

2.0 Aim and Objectives

1. Introduction
2. Implementation Process
   1. Distributed Mutual Exclusion (DME)
   2. Centralized Approach
   3. Fully Distributed Approach
   4. Behavior of Fully Distributed Approach
3. Designing a Distributed Processing System

**2.0. AIM AND OBJECTIVES**

At the end of this Lesson, you will be able to understand

Distributed Mutual Exclusion (DME), Centralized Approach,

Fully Distributed Approach,

Behavior of Fully Distributed Approach, and

Understand the Designing Process of a Distributed Systems

1. **INTRODUCTION**

The term distributed system is used to describe a system with the following characteristics: it consists of several computers that do not share a memory or a clock; the computers communicate with each other by exchanging messages over a communication network; and each computer has its own memory and runs its own operating system. The resources owned and controlled by a computer are said to be local to it, while the resources owned and controlled by other computers and those that can only be accessed through the network are said to be remote. Typically, accessing remote resources is more expensive e than accessing local resources because of the communication delays that occur in the network and the CPU overhead incurred to process communication protocols. Based on the context, the terms computer, note, host, site, machine, processor, and workstation are used interchangeably to denote a computer throughout this lesson.

**2.2. IMPLEMENTATION PROCESS**

Associate a timestamp with each system event

Require that for every pair of events A and B, if A B, then the timestamp of A is less than the timestamp of B

Within each process Pi a **logical clock**, LCi is associated

The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process

Logical clock is **monotonically increasing**

A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock

If the timestamps of two events A and B are the same, then the events are concurrent We may use the process identity numbers to break ties and to create a total ordering

**2.2.1. Distributed Mutual Exclusion (DME)**

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 Requirement

If *Pi* is executing in its critical section, then no other process *Pj* is executing in its critical section

We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections

**2.2.2. DME: Centralized Approach**

One of the processes in the system is chosen to coordinate the entry to the critical section

A process that wants to enter its critical section sends a request message to the coordinator

The coordinator decides which process can enter the critical section next, and its sends that process a reply message

When the process receives a reply message from the coordinator, it enters its critical section

After exiting its critical section, the process sends a release message to the coordinator and proceeds with its execution

This scheme requires three messages per critical-section entry: request

reply release

9

**2.2.3. DME: Fully Distributed Approach**

When process *Pi* wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request* (*Pi, TS*) to all other processes in the system

When process *Pj* receives a *request* message, it may reply immediately or it may defer sending a reply back

When process *Pi* receives a *reply* message from all other processes in the system, it can enter its critical section

After exiting its critical section, the process sends *reply* messages to all its deferred requests

The decision whether process *Pj* replies immediately to a *request*(*Pi, TS*) message or defers its reply is based on three factors:

If *Pj* is in its critical section, then it defers its reply to *Pi*

If *Pj* does *not* want to enter its critical section, then it sends a *reply* immediately to *Pi*

If *Pj* wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp *TS*

If its own request timestamp is greater than *TS*, then it sends a *reply* immediately to *Pi* (*Pi* asked first)

Otherwise, the reply is deferred

**2.2.4. Behavior of Fully Distributed Approach**

n Freedom from Deadlock is ensured

1. Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering
2. The timestamp ordering ensures that processes are served in a first-come, first served order

n The number of messages per critical-section entry is

2 x (*n* – 1)

This is the minimum number of required messages per critical-section entry when processes act independently and concurrently

**2.3. DESIGNING A DISTRIBUTED PROCESSING SYSTEM**

In general, designing a distributed operating system is more difficult than designing a centralized operating system for several reasons.

**Transparency**

We saw that one of the main goals of a distributed operating system is to make the existence of multiple computers invisible (transparent) and provide a single system image to its users. That is, a distributed operating system must be designed in such a way that a collection of distinct machines connected by a communication subsystem appears to its

10

users as a virtual uniprocessor. Achieving complete transparency is a difficult task and requires that several different aspects of transparency be supported by the distributed operating system. The eight forms of transparency identified by the International Standards Organization's Reference Model for Open Distributed Processing [ISO 1992] are access transparency, location transparency, replication transparency, failure transparency, migration transparency, concurrency transparency, performance transparency, and scaling transparency. These transparency aspects are described below

**Access Transparency**

Access transparency means that users should not need or be able to recognize whether a resource (hardware or software) is remote or local. This implies that the distributed operating system should allow users to access remote resources in the same way as local resources.

**Location Transparency**

The two main aspects of location transparency are as follows:

1 Name transparency. This refers to the fact that the name of a resource (hardware or software) should not reveal any hint as to the physical location of the resource.

2. User mobility. This refers to the fact that no matter which machine a user is logged onto, he or she should be able to access a resource with the same name.

**Replication Transparency**

For better performance and reliability, almost all distributed operating systems have the provision to create replicas (additional copies) of files and other resources on different nodes of the distributed system. In these systems, both the existence of multiple copies of a replicated resource and the replication activity should be transparent to the users.

**Failure Transparency**

Failure transparency deals with masking from the users' partial failures in the system, such as a communication link failure, a machine failure, or a storage device crash. A distributed operating system having failure transparency property will' continue to function, perhaps in a degraded form, in the face of partial failures.

**Migration Transparency**

For better performance, reliability, and security reasons, an object that is capable of being moved (such as a process or a file) is often migrated from one node to another in a distributed system.

**Concurrency Transparency**

Concurrency transparency means that each user has a feeling that he or she is the sole user of the system and other users do not exist in the system.

11

**Performance Transparency**

The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system.

**Scaling Transparency**

The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users.

**Reliability**

In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources. However, the existence of multiple instances of the resources alone cannot increase the system's reliability. Rather, the distributed operating system, which manages these resources, must be designed properly to increase the system's reliability by taking full advantage of this characteristic feature of a distributed system.

A fault is a mechanical or algorithmic defect that may generate an error. A fault in a system causes system failure. Depending on the manner in which a failed system behaves, system failures are of two types-fail-stop.

For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover from faults. Commonly used methods for dealing with these issues are fault avoidance and fault tolerance.

**Flexibility**

Another important issue in the design of distributed operating systems is flexibility. Flexibility is the most important feature for open distributed systems. The design of a distributed operating system should be flexible due to the following reasons:

1. Ease of modification.
2. Ease of enhancement

**Performance**

If a distributed system is to be used, its performance must be at least as good as a centralized system. That is, when a particular application is run on a distributed system, its overall performance should be better than or at least equal to that of running the same application on a single-processor system. However, to achieve this goal, it is important that the various components of the operating system of a distributed system be designed properly; otherwise, the overall performance of the distributed system may turn out to be worse than a centralized system. Some design principles considered useful for better performance are as follows:

12

1. Batch if possible. 2.Cache whenever possible. 3.Minimize copying of data.

1. Minimize network traffic.
2. Take advantage of fine-grain parallelism for multiprocessing.

**Scalability**

Scalability refers to the capability of a system to adapt to increased service load. It is inevitable that a distributed system will grow with time since it is very common to add new, machines or an entire sub network to the system to take care of increased workload or organizational changes in a company. Therefore, a distributed operating system should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users. Some guiding principles for designing scalable distributed systems are as follows:

1. Avoid centralized entities.
2. Avoid centralized algorithms.
3. Perform most operations on client workstations.

**Heterogeneity**

A heterogeneous distributed system consists of interconnected sets of dissimilar hardware or software systems. Because of the diversity, designing heterogeneous distributed systems is far more difficult than designing homogeneous distributed systems in which each system is based on the same, or closely related, hardware and software. However, as a consequence of large scale, heterogeneity is often inevitable in distributed systems. Furthermore, often heterogeneity is preferred by many users because heterogeneous distributed systems provide the flexibility to their users of different computer platforms for different applications.

**Security**

In order that the users can trust the system and rely on it, the various resources of a computer system must be protected against destruction and unauthorized access. Enforcing security in a distributed system is more difficult than in a centralized system because of the lack of a single point of control and the use of insecure networks for data communication. Therefore, as compared to a centralized system, enforcement of security in a distributed system has the following additional requirements:

1. It should be possible for the sender of a message to know that the message was received by the intended receiver.
2. It should be possible for the receiver of a message to know that the message was sent by the genuine sender.
3. It should be possible for both the sender and receiver of a message to be guaranteed that the contents of the message were not changed while it was in transfer.

13

Cryptography is the only known practical method for dealing with these security aspects of a distributed system.

**Emulation Of Existing Operating System**

For commercial success, it is important that a newly designed distributed operating system be able to emulate existing popular operating systems such as UNIX. With this property, new software can be written using the system call interface of the new operating system to take full advantage of its special features of distribution, but a vast amount of already existing old software can also be run on the same system without the need to rewrite them. Therefore, moving to the new distributed operating system will allow both types of software to be run side by side