UNIT-I

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1) Define Distributed System & discuss its characteristics. Give examples for Distributed Systems.

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

Characteristics of Distributed System:

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

Examples of distributed systems / applications of distributed computing :

Intranets, Internet, WWW, email.

Telecommunication networks: Telephone networks and Cellular networks.

Network of branch office computers -Information system to handle automatic processing of orders,

Real-time process control: Aircraft control systems,

Electronic banking,

Airline reservation systems,

Sensor networks,

Mobile and Pervasive Computing systems.

2) List the challenges in distributed systems. Explain in detail any two of them.

Challenges and Failures of a Distributed System are:

- Heterogeneity
- Scalability
- Openness
- Transparency
- Concurrency
- Security
- Failure Handling

Heterogeneity

Heterogeneity refers to the differences that arise in networks, programming languages, hardware, operating systems and differences in software implementation. For example, there are different hardware devices, tablets, mobile phones, computers, and etc.

Some challenges may present themselves due to heterogeneity. When programs are written in different languages or developers utilize different implementations (data structures, etc.) problems will arise when the computers try to communicate with each other. Thus it is important to have common standards agreed upon and adopted to streamline the process. Additionally, when we consider mobile code - code that can be transferred from one computer to the next - we may encounter some problems if the executables are not specified to accomodate both computers' instructions and specifications.

Scalability

A program is scalable if a program does not need to be redesigned to ensure stability and consistent performance as its workload increases. As such, a program (distributed system in our case) should not have a change in performance regardless of whether it has 10 nodes or 100 nodes.

As a distributed system is scaled, several factors need to be taken into account: size, geography, and administration. The associated problem with size is overloading. Overloading refers to the degradation of the system that refers as the system's workload increases (increase in number of users, resourced consumed, etc.). Secondly, with geography, as the distance that our distributed system encompasses increases, the reliability of our communication may break down. Additionally, as a distributed system is scaled, we may have to implement controls in the system; however, this may devolve into what we can effectively call an administrative mess.

Openness

The openness of distributed systems refers to the system's extensability and ability to be reimplemented. More specifically, the openness of a distributed system can be measured by three characteristics: interoperability, portability, and extensability as we previously mentioned. Interoperability refers to the system's ability to effectively

3>Synchronous and Asynchoronous in DC

Two variants of the interaction model ◊ In a distributed system it is hard to set time limits on the time taken for process execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models: the first has a strong assumption of time and the second makes no assumptions about time.

Synchronous distributed systems: Hadzilacos and Toueg [1994] define a synchronous distributed system to be one in which the following bounds are defined:

- the time to execute each step of a process has known lower and upper bounds;
- each message transmitted over a channel is received within a known bounded time;

- each process has a local clock whose drift rate from real time has a known bound.

It is possible to suggest likely upper and lower bounds for process execution time, message delay and clock drift rates in a distributed system. But it is difficult to arrive at realistic values and to provide guarantees of the chosen values. Unless the values of the bounds can be guaranteed, any design based on the chosen values will not be reliable. However, modelling an algorithm as a synchronous system may be useful for giving some idea of how it will behave in a real distributed system. In a synchronous system, it is possible to use timeouts, for example to detect the failure of a process, as shown in the section on the failure model.

Synchronous distributed systems can be built. What is required is for the processes to perform tasks with known resource requirements for which they can be guaranteed sufficient processor cycles and network capacity; and for processes to be supplied with clocks with bounded drift rates.

Asynchronous distributed systems: Many distributed systems, for example the Internet, are very useful without being able to qualify as synchronous systems. Therefore we need an alternative model: an asynchronous distributed system is one in which there are no bounds on:

- process execution speeds – for example, one process step may take only a picosecond and another a century; all that can be said is that each step may take an arbitrarily long time;
- message transmission delays – for example, one message from process A to process B may be delivered in zero time and another may take several years. In other words, a message may be received after an arbitrarily long time;
- clock drift rates - again, the drift rate of a clock is arbitrary.

The asynchronous model allows no assumptions about the time intervals involved in any execution. This exactly models the Internet, in which there is no intrinsic bound on server or network load and therefore on how long it takes, for example, to transfer a file using ftp. Sometimes an email message can take days to arrive. The box on the next page illustrates the difficulty of reaching an agreement in an asynchronous distributed system.

But some design problems can be solved even with these assumptions. For example, although the Web cannot always provide a particular response within a reasonable time limit, browsers have been designed to allow users to do other things while they are waiting. Any solution that is valid for an asynchronous distributed system is also valid for a synchronous one.

SYSTEM MODELS

1) Define Architecture Model. Mention its goal & explain the following with an example.

i) Mobile Code ii) Mobile Agent iii)Proxy Server & Cache

An architectural model of a distributed system is concerned with the placement of its parts and the relationships between them. Examples include the client-server model and the peer process model. The client-server model can be modified by:

- the partition of data or replication at cooperating servers;
- the caching of data by proxy servers and clients;
- the use of mobile code and mobile agents;
- the requirement to add and remove mobile devices in a convenient manner.

The architecture of a system is its structure in terms of separately specified components. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. The architectural design of a building has similar aspects – it determines not only its appearance but also its general structure and architectural style (gothic, neo-classical, modern) provides a consistent frame of reference for the design.

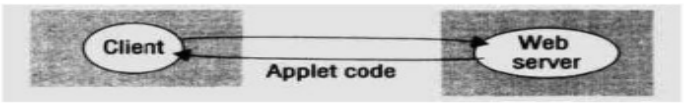
Proxy servers and caches ◊ A *cache* is a store of recently used data objects that is closer than the objects themselves. When a new object is received at a computer it is added to the cache store, replacing some existing objects if necessary. When an object is needed by a client process the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-date copy is fetched. Caches may be collocated with each client or they may be located in a proxy server that can be shared by several clients.

Caches are used extensively in practice. Web browsers maintain a cache of recently visited web pages and other web resources in the client's local file system, using a special HTTP request to check with the original server that cached pages are up to date before displaying them. Web proxy servers (Figure 2.4) provide a shared cache of web resources for the client machines at a site or across several sites. The purpose of proxy servers is to increase availability and performance of the service by reducing the load on the wide-area network and web servers. Proxy servers can take on other roles; for example, they may be used to access remote web servers through a firewall.

Mobile code ◊ Chapter 1 introduced mobile code. Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there, as shown in Figure 2.6. An advantage of running the downloaded code locally is

Web applets

a) client request results in the downloading of applet code



b) client interacts with the applet

that it can give good interactive response since it does not suffer from the delays or variability of bandwidth associated with network communication.

Accessing services means running code that can invoke their operations. Some services are likely to be so standardized that we can access them with an existing and well-known application – the Web is the most common example of this, but even there, some web sites use functionality not found in standard browsers and require the downloading of additional code. The additional code may for example communicate with the server. Consider an application that requires that users should be kept up to date with changes as they occur at an information source in the server. This cannot be achieved by normal interactions with the web server, which are always initiated by the client. The solution is to use additional software that operates in a manner often referred to as a *push* model – one in which the server instead of the client initiates interactions.

Mobile agents ◊ A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, eventually returning with the results. A mobile agent may make many invocations to local resources at each site it visits – for example accessing individual database entries. If we compare this architecture with a static client making remote invocations to some resources, possibly transferring large amounts of data, there is a reduction in communication cost and time through the replacement of remote invocations with local ones.

2. Summarize the following design requirements for Distributed Architectures ;

i) Performance Issues ii) Quality of Service

Performance issues ◊ The challenges arising from the distribution of resources extend well beyond the need for the management of concurrent updates. Performance issues arising from the limited processing and communication capacities of computers and networks are considered under the following subheadings:

Responsiveness: Users of interactive applications require a fast and consistent response to interaction; but client programs often need to access shared resources. When a remote service is involved, the speed at which the response is generated is determined not just by the load and performance of the server and the network but also by delays in all the software components involved – the client and server operating systems' communication and middleware services (remote invocation support, for example) as well as the code of the process that implements the service.

Quality of service ◊ Once users are provided with the functionality that they require of a service such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main non-functional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*. *Adaptability* to meet changing system configurations and resource availability has recently been recognized as a further important aspect of service quality. Birman has long argued for the importance of these quality aspects and his book [Birman 1996] provides some interesting perspectives of their impact on system design.

3. Explain the Failure Model. With the help of a tabular column describe the various classes of Arbitrary, Omission & Timing failures

UNIT-II

INTER PROCESS COMMUNICAITON

1. Explain the characteristics of IPC & With a neat diagram explain sockets
2. Compare & Contrast between Synchronous & Asynchronous communication in the context of IPC.

Synchronous and asynchronous communication ◊ A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving process may be either synchronous or asynchronous. In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations. Whenever a *send* is issued the sending process is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued the process blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *send* operation is *non-blocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has few disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, current systems do not generally provide the non-blocking form of *receive*.

3. Explain Java API for the following.
    - UDP datagrams
    - TCP streams
4. Discuss issues relating to datagram communication.
5. Discuss the Characteristics and issues related to stream communication.
6. Define marshalling and unmarshalling.
7. Explain CORBA CDR with an example

CORBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 [OMG 1998a]. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.

Primitive types: CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit *short* occupies two bytes in the message and for big-endian ordering, the most significant bits occupy the first byte and the least significant bits occupy the second byte. Each primitive value is placed at an index in the sequence of bytes according to its size. Suppose that the sequence of bytes is indexed from zero upwards. Then a primitive value of size n bytes (where $n = 1, 2, 4$ or 8) is appended to the sequence at an index that is a multiple of n in the stream of bytes. Floating-point values follow the IEEE standard – in which the sign, exponent and fractional part are in bytes 0–n for big-endian ordering and the other way round for little-endian. Characters are represented by a code set agreed between client and server.

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 4.7.

Example:

CORBA CDR message

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*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

8. Explain Java object serialization with an example.
9. Define Marshalling. Construct a marshalled form that represents a Organization with instance variable values :{ 'KLSGIT','BELGAUM', 1979, 590008} by using CORBA-CDR & Java Serialization.
10. Analyze the failure model of Request/Reply protocol in client-server Communication using UDP

Failure model of the request-reply protocol ◊ If the three primitives *doOperation*, *getRequest* and *sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures. That is:

- they suffer from omission failures;
- messages are not guaranteed to be delivered in sender order.

In addition, the protocol can suffer from the failure of processes (see Section 2.3.2). We assume that processes have crash failures. That is, when they halt, they remain halted – they do not produce byzantine behaviour.

To allow for occasions when a server has failed or a request or reply message is dropped, *doOperation* uses a timeout when it is waiting to get the server's reply message. The action taken when a timeout occurs depends upon the delivery guarantees to be offered.

Timeouts: There are various options as to what *doOperation* can do after a timeout. The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed. This is not the usual approach – the timeout may have been due to the request or reply message getting lost – and in the latter case, the operation will have been performed. To compensate for the possibility of lost messages, *doOperation* sends the request message repeatedly until either it gets a reply or else it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages. Eventually, when *doOperation* returns it will indicate to the client by an exception that no result was received.

Discarding duplicate request messages: In cases when the request message is retransmitted, the server may receive it more than once. For example, the server may receive the first request message but take longer than the client's timeout to execute the command and return the reply. This can lead to the server executing an operation more than once for the same request. To avoid this, the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates. If the server has not yet sent the reply, it need take no special action – it will transmit the reply when it has finished executing the operation.

Lost reply messages: If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution. Some servers can execute their operations more than once and obtain the same results each time. An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once. For example, an operation to add an element to a set is an idempotent operation because it will always have the same effect on the set each time it is performed, whereas an operation to append an item to a sequence is not an idempotent operation, because it extends the sequence each time it is performed. A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

History: For servers that require retransmission of replies without re-execution of operations, a history may be used. The term 'history' is used to refer to a structure that contains a record of (reply) messages that have been transmitted. An entry in a history contains a request identifier, a message and an identifier of the client to which it was sent. Its purpose is to allow the server to retransmit reply messages when client processes request them. A problem associated with the use of a history is its memory cost. A history will become very large unless the server can tell when the messages will no longer be needed for retransmission.

As clients can make only one request at a time, the server can interpret each request as an acknowledgement of its previous reply. Therefore the history need contain

11. Discuss the drawbacks of UDP over TCP stream to implement the request-reply protocol

12. Explain request-reply communication with the neat diagram and specify the operations of the same.

13. List and explain RPC exchange protocols.

14. Explain HTTP request and reply message format.