PSNA COLLEGE OF ENGINEERING AND TECHNOLOGY, DINDIGUL

DEPARTMENT OF COMPUTER APPLICATIONS (MCA)

MC5207 – CLOUD COMPUTING TECHNOLOGIES

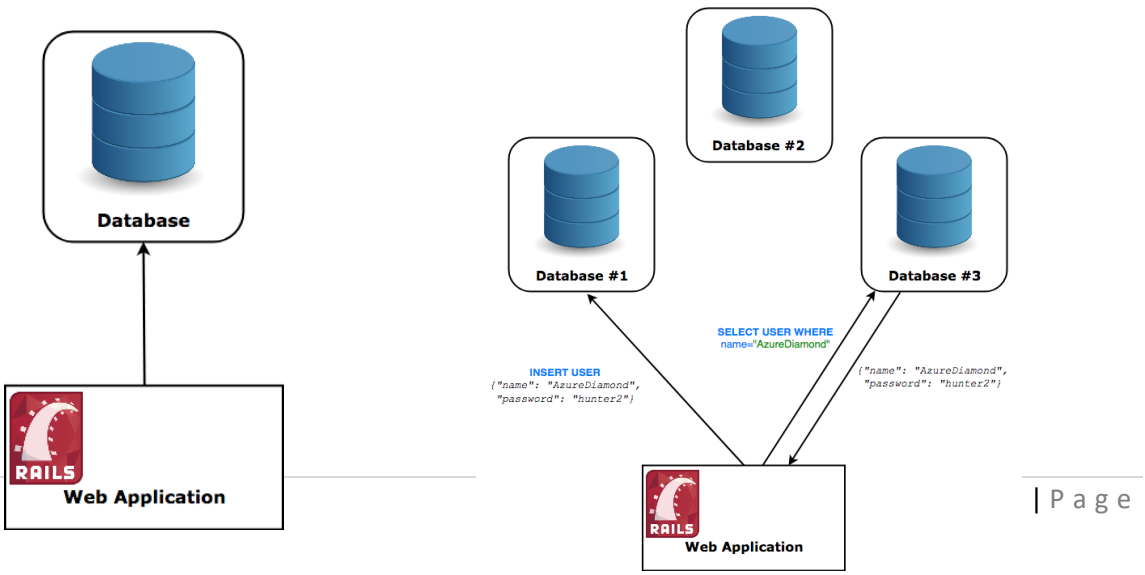
UNIT - 1 DISTRIBUTED SYSTEMS

Introduction to Distributed Systems - Characterization of Distributed Systems - Distributed Architectural Models - Remote Invocation - Request-Reply Protocols - Remote Procedure Call - Remote Method Invocation - Group Communication - Coordination in Group Communication - Ordered Multicast - Time Ordering - Physical Clock Synchronization - Logical Time and Logical Clocks

INTRODUCTION TO DISTRIBUTED SYSTEMS

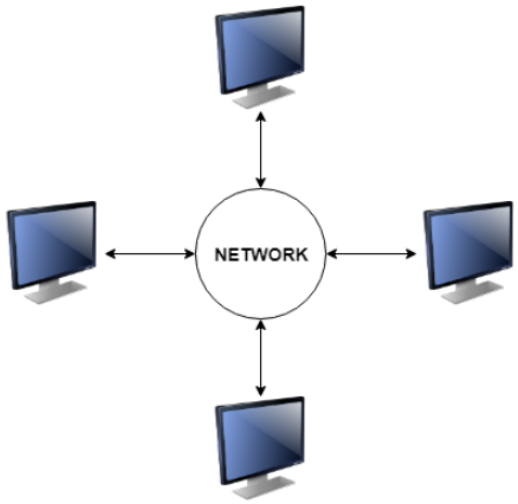
What is a distributed system?

A distributed system in its simplest definition is a group of computers working together as to appear as a single computer to the end-user. These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.



DEFINITION

A distributed system contains multiple nodes that are physically separate but linked together using the network. All the nodes in this system communicate with each other and handle processes in tandem. Each of these nodes contains a small part of the distributed operating system software.



DISTRIBUTED OPERATING SYSTEM

TYPES OF DISTRIBUTED SYSTEMS

The nodes in the distributed systems can be arranged in the form of client/server systems or peer to peer systems. Details about these are as follows −

1. Client/Server Systems

In client server systems, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server. Both the client

and server usually communicate via a computer network and so they are a part of distributed systems.

2. **Peer to Peer Systems**

The peer to peer systems contains nodes that are equal participants in data sharing. All the tasks are equally divided between all the nodes. The nodes interact with each other as required as share resources. This is done with the help of a network.

ADVANTAGES OF DISTRIBUTED SYSTEMS

Some advantages of Distributed Systems are as follows −

- All the nodes in the distributed system are connected to each other. So nodes can easily share data with other nodes.
- More nodes can easily be added to the distributed system i.e. it can be scaled as required.
- Failure of one node does not lead to the failure of the entire distributed system. Other nodes can still communicate with each other.
- Resources like printers can be shared with multiple nodes rather than being restricted to just one.

DISADVANTAGES OF DISTRIBUTED SYSTEMS

Some disadvantages of Distributed Systems are as follows −

- It is difficult to provide adequate security in distributed systems because the nodes as well as the connections need to be secured.
- Some messages and data can be lost in the network while moving from one node to another.

- The database connected to the distributed systems is quite complicated and difficult to handle as compared to a single user system.
- Overloading may occur in the network if all the nodes of the distributed system try to send data at once.

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1. ***Concurrency*:** In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example computers) to the network. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

2. ***No global clock:*** When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network.

3. ***Independent failures*:** All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running. The consequences of this characteristic of distributed systems will be a recurring theme throughout the book.

DISTRIBUTED SYSTEM MODELS

1. Architectural Models
2. Interaction Models
3. Fault Models

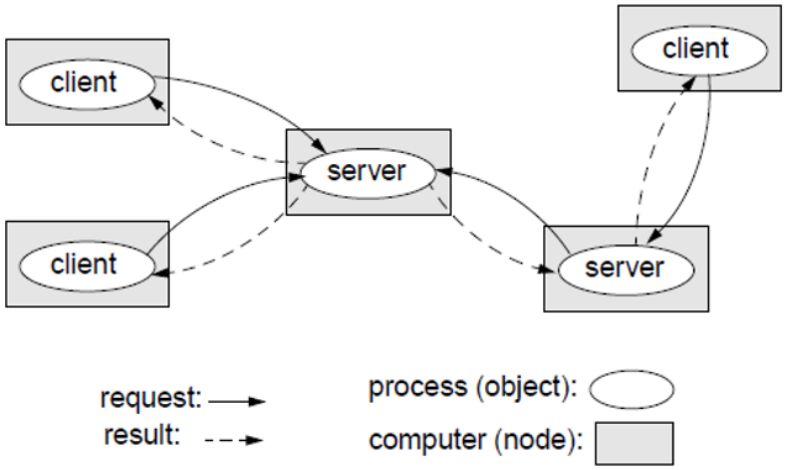
1. Architectural Models

Architectural model describes responsibilities distributed between system components and how are these components placed.

a)Client-server model

☞ The system is structured as a set of processes, called servers, that offer services to the users, called clients.

- The client-server model is usually based on a simple request/reply protocol, implemented with send/receive primitives or using remote procedure calls (RPC) or remote method invocation (RMI):
- The client sends a request (invocation) message to the server asking for some service;
- The server does the work and returns a result (e.g. the data requested) or an error code if the work could not be performed.



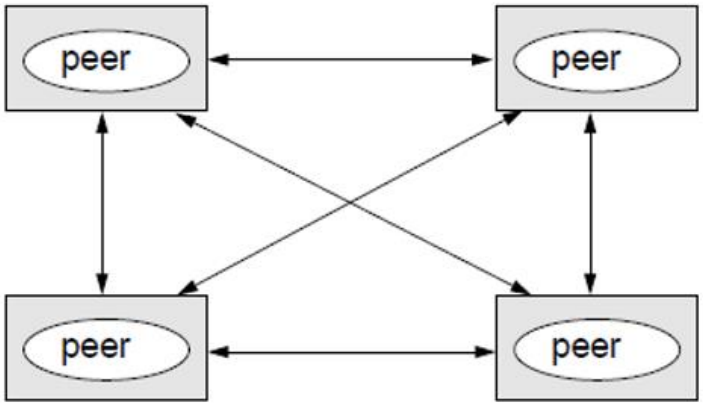
A server can itself request services from other servers; thus, in this new relation, the server itself acts like a client.

b)Peer-to-peer

☞ All processes (objects) play similar role.

- Processes (objects) interact without particular distinction between clients and servers.
- The pattern of communication depends on the particular application.
- A large number of data objects are shared; any individual computer holds only a small part of the application database.

- Processing and communication loads for access to objects are distributed across many computers and access links.
- This is the most general and flexible model.



- Peer-to-Peer tries to solve some of the above
- It distributes shared resources widely -> share computing and communication loads.

☞ Problems with peer-to-peer:

- High complexity due to
  - Cleverly place individual objects
  - retrieve the objects
  - maintain potentially large number of replicas.

2. Interaction Model

Interaction model are for handling time i. e. for process execution, message delivery, clock drifts etc.

- Synchronous distributed systems

Main features:

- Lower and upper bounds on execution time of processes can be set.

- Transmitted messages are received within a known bounded time.
- Drift rates between local clocks have a known bound.

Important consequences:

1. In a synchronous distributed system there is a notion of global physical time (with a known relative precision depending on the drift rate).
2. Only synchronous distributed systems have a predictable behavior in terms of timing. Only such systems can be used for hard real-time applications.
3. In a synchronous distributed system it is possible and safe to use timeouts in order to detect failures of a process or communication link.

☞ It is difficult and costly to implement synchronous distributed systems.

- Asynchronous distributed systems

☞ Many distributed systems (including those on the Internet) are asynchronous. - No bound on process execution time (nothing can be assumed about speed, load, and reliability of computers). - No bound on message transmission delays (nothing can be assumed about speed, load, and reliability of interconnections) - No bounds on drift rates between local clocks.

Important consequences:

1. In an asynchronous distributed system there is no global physical time. Reasoning can be only in terms of logical time (see lecture on time and state).
2. Asynchronous distributed systems are unpredictable in terms of timing.

3. No timeouts can be used.

☞ Asynchronous systems are widely and successfully used in practice.

In practice timeouts are used with asynchronous systems for failure detection.

However, additional measures have to be applied in order to avoid duplicated messages, duplicated execution of operations, etc.

3. Fault Models

☞ Failures can occur both in processes and communication channels. The reason can be both software and hardware faults.

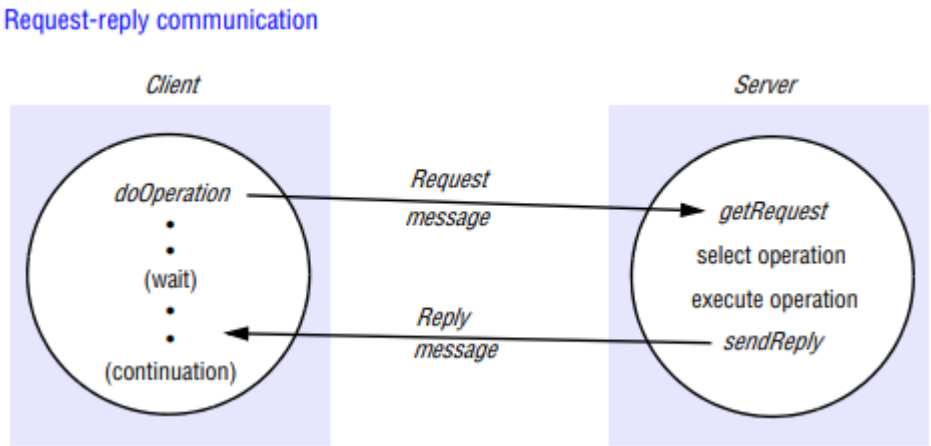
☞ Fault models are needed in order to build systems with predictable behavior in case of faults (systems which are fault tolerant).

☞ such a system will function according to the predictions, only as long as the real faults behave as defined by the “fault model”.

REMOTE INVOCATION

REQUEST-REPLY PROTOCOLS

The protocol we describe here is based on a trio of communication primitives, doOperation, getRequest and sendReply, as shown in Figure



The doOperation method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply. It is assumed that the client calling doOperation marshals the arguments into an array of bytes and unmarshals the results from the array of bytes that is returned. The doOperation method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument. After sending the request message, doOperation invokes receive to get a reply message, from which it extracts the result and returns it to the caller. The caller of doOperation is blocked until the server performs the requested operation and transmits a reply message to the client process.

The doOperation method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument. After sending the request message, doOperation invokes receive to get a reply message, from which it extracts the result and returns it to the caller. The caller of doOperation is blocked until the server performs the requested operation and transmits a reply message to the client process.

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

The information to be transmitted in a request message or a reply message. Request-reply message structure messageType int (0=Request, 1= Reply) requestId int remoteReference RemoteRef operationId int or Operation arguments // array of bytes . The first field indicates whether the message is a Request or a Reply message. The second field, requestId, contains a message identifier. A doOperation in the client generates a requestId for each request message, and the server copies these IDs into the corresponding reply messages. This enables doOperation to check that a reply message is the result of the current request, not a delayed earlier call. The third field is a remote reference. The fourth field is an identifier for the operation to be invoked. For example, the operations in an interface might be numbered 1, 2, 3, ... , if the client and server use a common language that supports reflection, a representation of the operation itself may be put in this field.

REMOTE PROCEDURE CALL

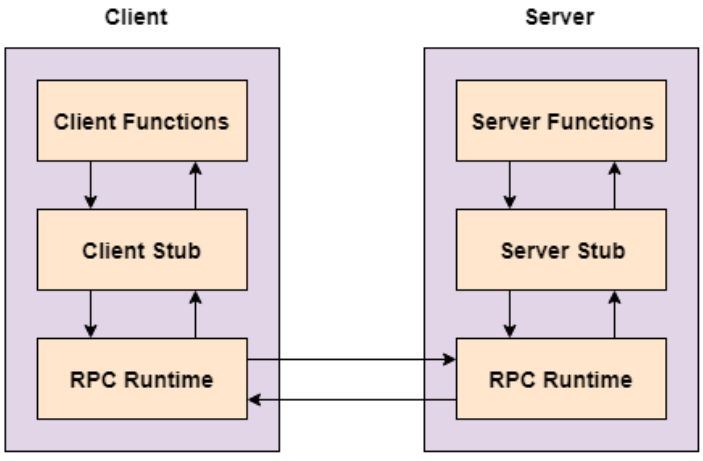
A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the

required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequence of events in a remote procedure call are given as follows –

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows −

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows −

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
- There is no flexibility in RPC for hardware architecture. It is only interaction based.
- There is an increase in costs because of remote procedure call.

REMOTE METHOD INVOCATION

In a distributed computing environment, distributed object communication realizes communication between distributed objects. The main role is to allow objects to access data and invoke methods on remote objects (objects residing in non-local memory space). Invoking a method on a remote object is known as remote method invocation (RMI) or remote invocation, and is the object-oriented programming analog of a remote procedure call (RPC).

RMI (Remote Method Invocation)

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

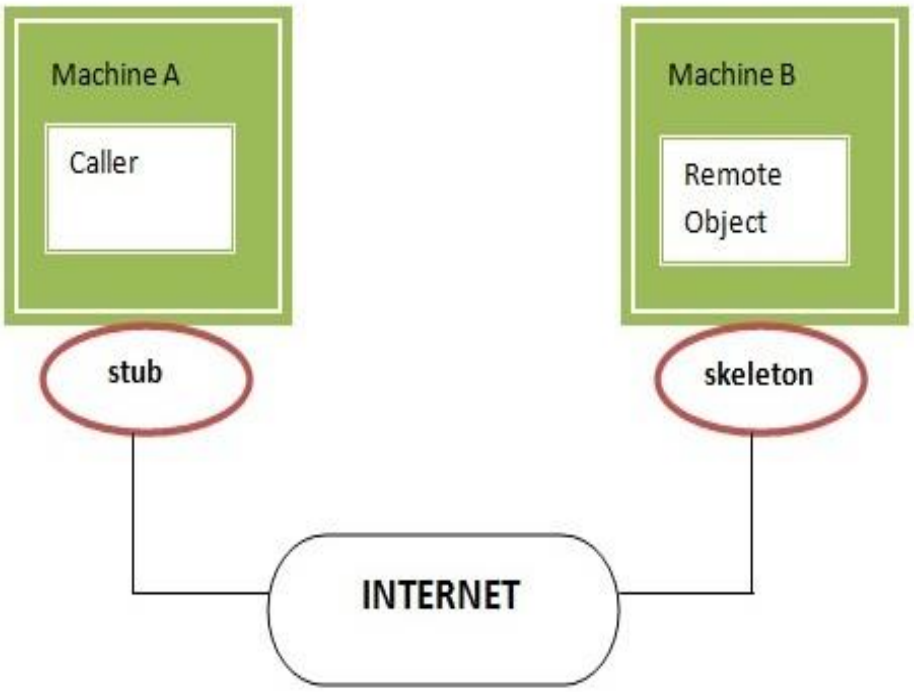
The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

1. The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks: It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, a stub protocol was introduced that eliminates the need for skeletons



Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application

1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

Java RMI Example

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmi c tool
4. Start the registry service by rmi registry tool
5. Create and start the remote application
6. Create and start the client application

Java RMI Program

// A.JAVA

```
import java.rmi.*;
public interface A extends Remote
{
        public int add(int a,int b)throws RemoteException;
}
```

//B.JAVA

```
import java.rmi.*;
```

```
import java.rmi.server.UnicastRemoteObject;
public class B extends UnicastRemoteObject implements A
{
	public B() throws RemoteException{}
	public int add(int a,int b) throws RemoteException
	{
		int t=a+b;
		return t;
	}
}
```

//C.JAVA

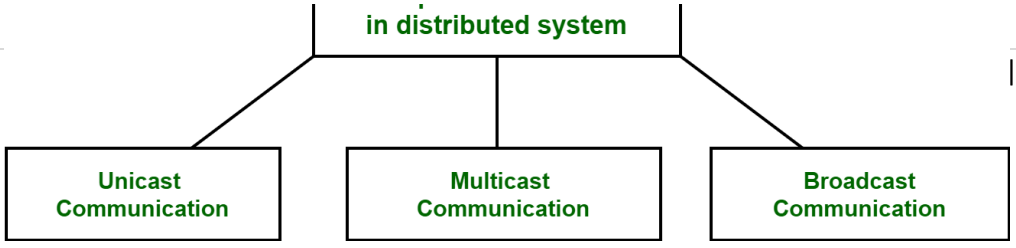
```
import java.rmi.*;
import java.rmi.server.*;
public class C
{
	public static void main(String args[])
	{
		try
		{
			B b=new B();
			Naming.rebind("AdditionApplication",b);
		}
		catch(Exception e)
		{
		}
}
}
```

//D.JAVA

```
import java.rmi.*;
public class D
{
	public static void main(String args[])
	{
		try
		{
			String s="rmi://localhost/AdditionApplication";
			A a1=(A) Naming.lookup(s);
			int result=a1.add(12,4);
			System.out.println("The value is:"+result);
		}
		catch(Exception e){}
}
}
```

GROUP COMMUNICATION

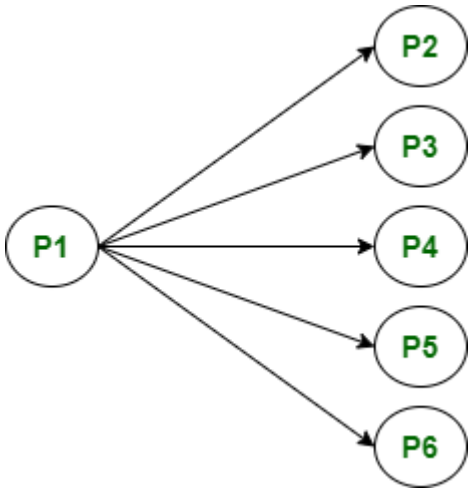
Communication between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes. When one source process tries to communicate with multiple processes at once, it is called **Group Communication**. A group is a collection of interconnected processes with abstraction. This abstraction is to hide the message passing so that the communication looks like a normal procedure call. Group communication also helps the processes from different hosts to work together and perform operations in a synchronized manner, therefore increases the overall performance of the system.



Types of Group Communication in a Distributed System:

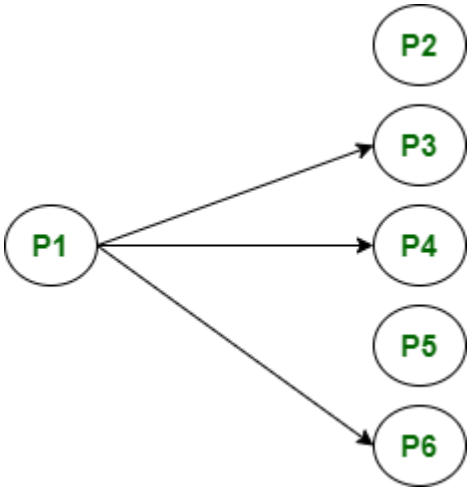
1. Broadcast Communication :

When the host process tries to communicate with every process in a distributed system at same time. Broadcast communication comes in handy when a common stream of information is to be delivered to each and every process in most efficient manner possible. Since it does not require any processing whatsoever, communication is very fast in comparison to other modes of communication. However, it does not support a large number of processes and cannot treat a specific process individually.



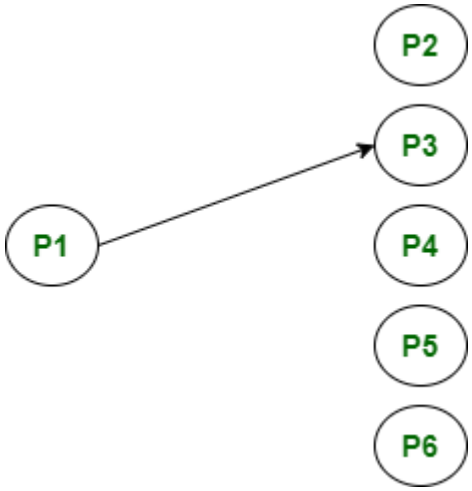
2. Multicast Communication :

When the host process tries to communicate with a designated group of processes in a distributed system at the same time. This technique is mainly used to find a way to address problem of a high workload on host system and redundant information from process in system. Multitasking can significantly decrease time taken for message handling.



3. Unicast Communication :

When the host process tries to communicate with a single process in a distributed system at the same time. Although, same information may be passed to multiple processes. This works best for two processes communicating as only it has to treat a specific process only. However, it leads to overheads as it has to find exact process and then exchange information/data.



COORDINATION IN GROUP COMMUNICATION

Coordination in group communication is how to achieve the desired reliability and ordering properties across all members of a group. We are particularly seeking reliability in terms of the properties of validity, integrity and agreement, and ordering in terms of FIFO ordering, causal ordering and total ordering. The system under consideration contains a collection of processes, which can communicate reliably over one-to-one channels. As before, processes may fail only by crashing.

The operation multicast(g, m) sends the message m to all members of the group g of processes. Correspondingly, there is an operation deliver(m) that delivers a message sent by multicast to the calling process. We use the term deliver rather than receive to make clear that a multicast message is not always handed to the application layer inside the process as soon as it is received at the process's node. This is explained when we discuss multicast delivery semantics shortly. Every message m carries the unique identifier of the process sender(m) that sent it, and the unique destination group identifier group(m). We assume that processes do not lie about the origin or destinations of messages.

System models are

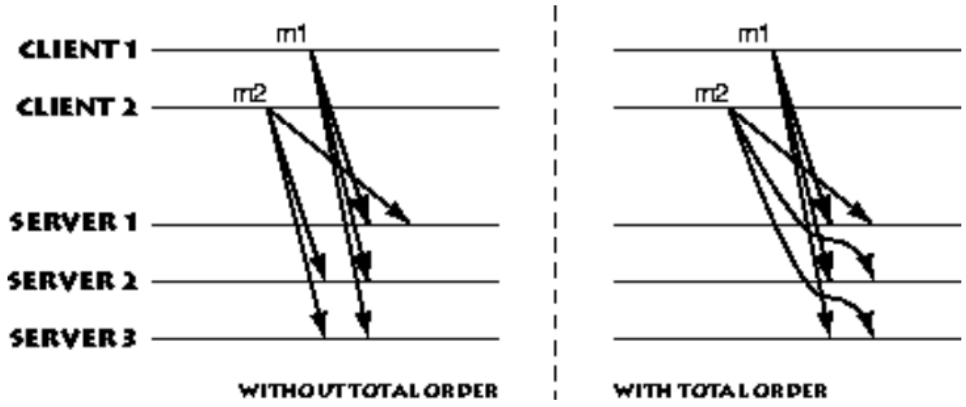
1. Basic Multicast – Simply send message m to process p in a group.
2. Reliable Multicast – Ensure that the message m is delivered in p or not. Must have the following properties: –

(i) Integrity: A working process p in group g delivers m at most once, and m was multicast by some working process –

(ii) Agreement: If a working process delivers m then all other working processes in group g will deliver m

3. Ordered Multicast

ORDERED MULTICAST



It is wanted that messages are delivered in "correct" (intended, consistent etc) order

Common ordering requirements are:

1. FIFO ordering: If a correct process issues multicast(g, m) and then multicast(g, m'), then every correct process that delivers mc will deliver m before m' .
2. Causal ordering: If multicast(g, m) -> multicast(g, m'), where -> is the happened-before relation induced only by messages sent between the members of g, then any correct process that delivers m' will deliver m before m' .

3. Total ordering: If a correct process delivers message m before it delivers m' , then any other correct process that delivers m' will deliver m before m' .

Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by happened-before. Note that FIFO ordering and causal ordering are only partial orderings: not all messages are sent by the same process, in general; similarly, some multicasts are concurrent (not ordered by happened-before).

TOTAL ORDERING

- Requires messages are delivered same order by each process
- But this order may have no relation to causality or message sending order
- Can be modified to be FIFO, total or Causal total orders

FIFO ordered multicast

Our reliable multicast implements FIFO

- Assuming the Bmulticast sends to group members in same order & channels are FIFO
- Sequence numbers can be used to implement FIFO otherwise

Causally ordered Multicast

- Each process has a Vector clock
- Suppose p sends a multicast m
- q receives m and holds it until: -
  - It has delivered any earlier message by p

- delivered any multicast message that has been delivered by p (to its application) before p multicast m

- These are easy to check using vector timestamps

Total ordered multicast

- Using sequencer process
  - p wants to multicast
  - It asks sequencer process for a sequence number
  - Sends multicast tagged with the sequence number
  - All processes deliver messages by sequence number
- Simple
- Single point of failure and bottleneck
- Using collective agreement
- p first sends Bmulticast to the group
- Each process in group picks a sequence number
- Processes run a distributed protocol to agree on a sequence number for the message
- Messages delivered according to sequence number

TIME ORDERING

- Physical Clock Synchronization
- Logical Time and Logical Clocks

PHYSICAL CLOCK SYNCHRONIZATION

In the distributed system, the hardware and software components communicate and coordinate their actions by message passing. Each node in distributed systems can share their resources with other nodes. So, there is need of proper allocation of resources to preserve the state of resources and help coordinate between the several processes. To resolve such conflicts, synchronization is used. Synchronization in distributed systems is achieved via clocks.

The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system.

The clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

1. External clock synchronization is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.

2. Internal clock synchronization is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

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

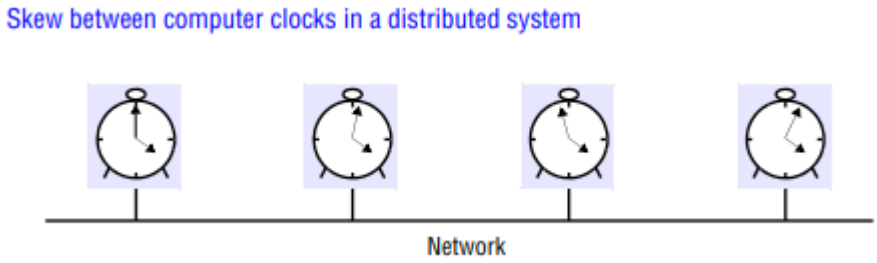
1. Centralized is the one in which a time server is used as a reference. The single time server propagates its time to the nodes and all the nodes adjust the time accordingly. It is dependent on single time server so if that node fails, the whole system will lose synchronization. Examples of centralized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

LOGICAL TIME AND LOGICAL CLOCKS

Logical Clocks refer to implementing a protocol on all machines within your distributed system, so that the machines are able to maintain consistent ordering of events within some virtual timespan. A logical clock is a mechanism for capturing

chronological and causal relationships in a distributed system. Distributed systems may have no physically synchronous global clock, so a logical clock allows global ordering on events from different processes in such systems.

Suppose, we have more than 10 PCs in a distributed system and every PC is doing it's own work but then how we make them work together. There comes a solution to this i.e. LOGICAL CLOCK.



Method-1:

To order events across process, try to sync clocks in one approach. This means that if one PC has a time 2:00 pm then every PC should have the same time which is quite not possible. Not every clock can sync at one time. Then we can't follow this method.

Method-2:

Another approach is to assign Timestamps to events.

Taking the example into consideration, this means if we assign the first place as 1, second place as 2, third place as 3 and so on. Then we always know that the first place will always come first and then so on. Similarly, If we give each PC their individual number than it will be organized in a way that 1st PC will complete its process first and then second and so on.

BUT, Timestamps will only work as long as they obey causality.

What is causality ?

Causality is fully based on HAPPEN BEFORE RELATIONSHIP.

- Taking single PC only if 2 events A and B are occurring one by one then TS(A) < TS(B). If A has timestamp of 1, then B should have timestamp more than 1, then only happen before relationship occurs.
- Taking 2 PCs and event A in P1 (PC.1) and event B in P2 (PC.2) then also the condition will be TS(A) < TS(B). Taking example- suppose you are sending message to someone at 2:00:00 pm, and the other person is receiving it at 2:00:02 pm.Then it's obvious that TS(sender) < TS(receiver).

Properties Derived from Happen Before Relationship –

Transitive Relation –

If, TS(A) <TS(B) and TS(B) <TS(C), then TS(A) < TS(C)

Causally Ordered Relation –

a->b, this means that a is occurring before b and if there is any changes in a it will surely reflect on b.

Concurrent Event –

This means that not every process occurs one by one, some processes are made to happen simultaneously i.e., A || B.