

Experiment No: 1

Aim: Write a program to implement the RPC/RMI mechanism.

Theory:

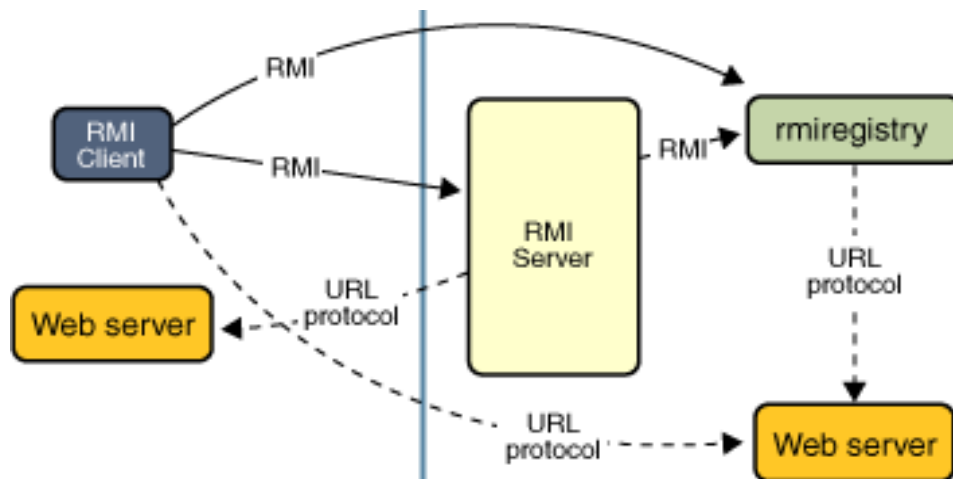
RMI (Remote Method Invocation)

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

Distributed object applications need to do the following:

- Locate remote objects. Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- Communicate with remote objects. Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

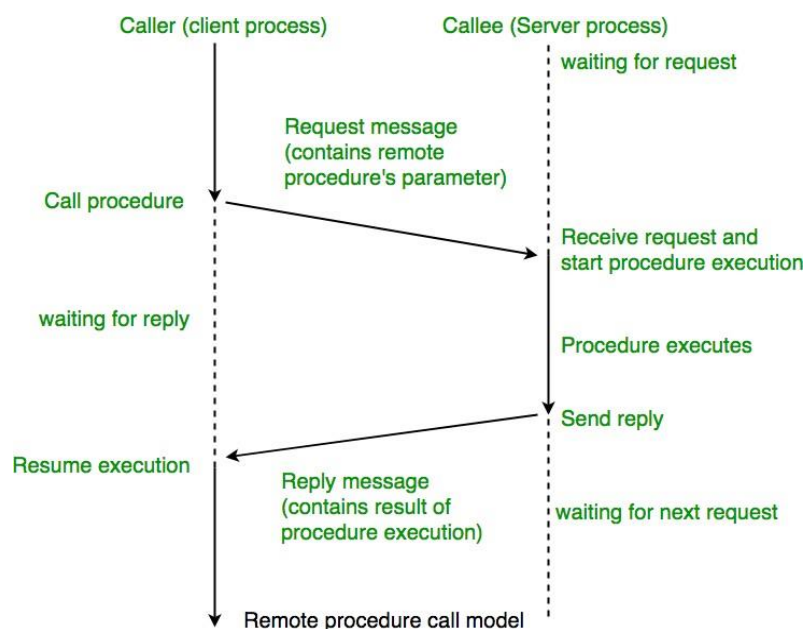
The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

When making a Remote Procedure Call:

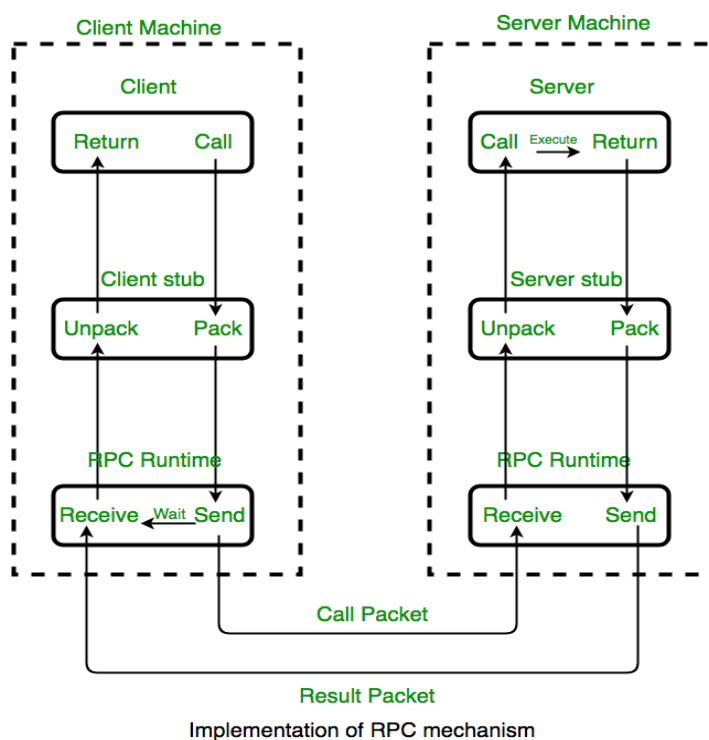


1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.

2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

NOTE: RPC is especially well suited for client-server (e.g. query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



The following steps take place during a RPC :

1. A client invokes a client stub procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub marshalls(pack) the parameters into a message. Marshaling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.

4. On the server, the transport layer passes the message to a server stub, which demarshalls(unpack) the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

Program:

Server.py

```
import socket

def server_program():

    # get the hostname

    host = socket.gethostname()

    port = 5000 # initiate port no above 1024

    server_socket = socket.socket() # get instance

    # look closely. The bind() function takes tuple as argument

    server_socket.bind((host, port)) # bind host address and port together

    # configure how many client the server can listen simultaneously

    server_socket.listen(2)

    conn, address = server_socket.accept() # accept new connection

    print("Connection from: " + str(address))

    while True:

        # receive data stream. it won't accept data packet greater than 1024 bytes

        data = conn.recv(1024).decode()

        if not data:
```

```

        # if data is not received break

        break

    print("from connected user: " + str(data))

    data = input(' -> ')

    conn.send(data.encode()) # send data to the client

    conn.close() # close the connection

if __name__ == '__main__':

    server_program()

```

Client.py

```

import socket

def client_program():

    host = socket.gethostname() # as both code is running on same pc

    port = 5000 # socket server port number

    client_socket = socket.socket() # instantiate

    client_socket.connect((host, port)) # connect to the server

    message = input(" -> ") # take input

    while message.lower().strip() != 'bye':

        client_socket.send(message.encode()) # send message

        data = client_socket.recv(1024).decode() # receive response

        print('Received from server: ' + data) # show in terminal

        message = input(" -> ") # again take input

    client_socket.close() # close the connection

if __name__ == '__main__':

    client_program()

```

Output:

```
server.py > ...
1 import socket
2 def server_program():
3     # get the hostname
4     host = socket.gethostname()
5     port = 5000 # initiate port no above 1024
6     server_socket = socket.socket() # get instance
7     # look closely. The bind() function takes tuple as argument
8     server_socket.bind((host, port)) # bind host address and port together
9     # configure how many client the server can listen simultaneously
10    server_socket.listen(2)
11    conn, address = server_socket.accept() # accept new connection
12    print("Connection from: " + str(address))
13    while True:
14        # receive data stream. it won't accept data packet greater than 1024 bytes
15        data = conn.recv(1024).decode()
16        if not data:
17            # if data is not received break
18            break
19        print("from connected user: " + str(data))
20        data = input(' -> ')
21        conn.send(data.encode()) # send data to the client
22    conn.close() # close the connection
23 if __name__ == '__main__':
24    server_program()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

python

python

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell <https://aka.ms/pscore6>

PS E:\New folder\New folder\DC> python client.py
-> Rohit Sabat
Received from server: Ragh wardayal ma rya
->

```
client.py > ...
1 import socket
2 def client_program():
3     host = socket.gethostname() # as both code is running on same pc
4     port = 5000 # socket server port number
5     client_socket = socket.socket() # instantiate
6     client_socket.connect((host, port)) # connect to the server
7     message = input(" -> ") # take input
8     while message.lower().strip() != 'bye':
9         client_socket.send(message.encode()) # send message
10        data = client_socket.recv(1024).decode() # receive response
11        print('Received from server: ' + data) # show in terminal
12        message = input(" -> ") # again take input
13    client_socket.close() # close the connection
14 if __name__ == '__main__':
15    client_program()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

python

python

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell <https://aka.ms/pscore6>

PS E:\New folder\New folder\DC> python client.py
-> Rohit Sabat
Received from server: Ragh wardayal ma rya
->

Conclusion:

Thus we have successfully completed a program to implement the RPC/RMI mechanism.

Experiment No: 2

Aim: Write a program to implement the Multiple Thread application using java.

Theory:

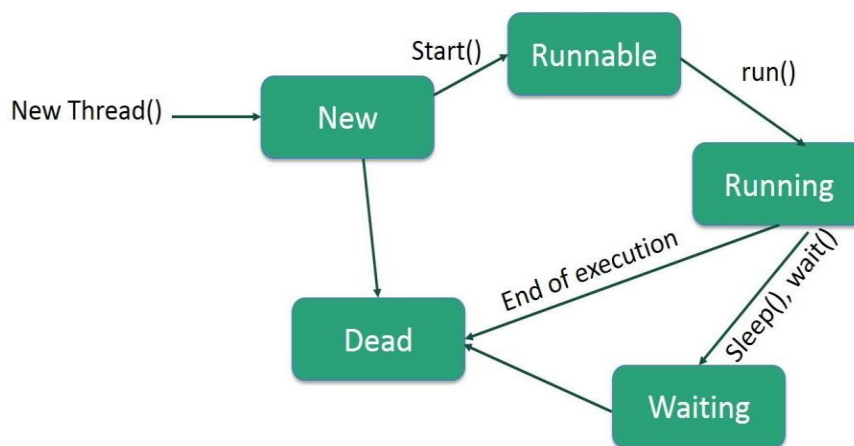
Java is a multi-threaded programming language which means we can develop multi-threaded programs using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle –

- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Program:

ThreadClass.java:

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
    ThreadDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }
    public void run() {
```



```

        System.out.println("Running " + threadName );
    try {
        for(int i = 7; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
        }
    } catch (InterruptedException e) {
        System.out.println("Thread " + threadName + " interrupted.");
    }
    System.out.println("Thread " + threadName + " exiting.");
}

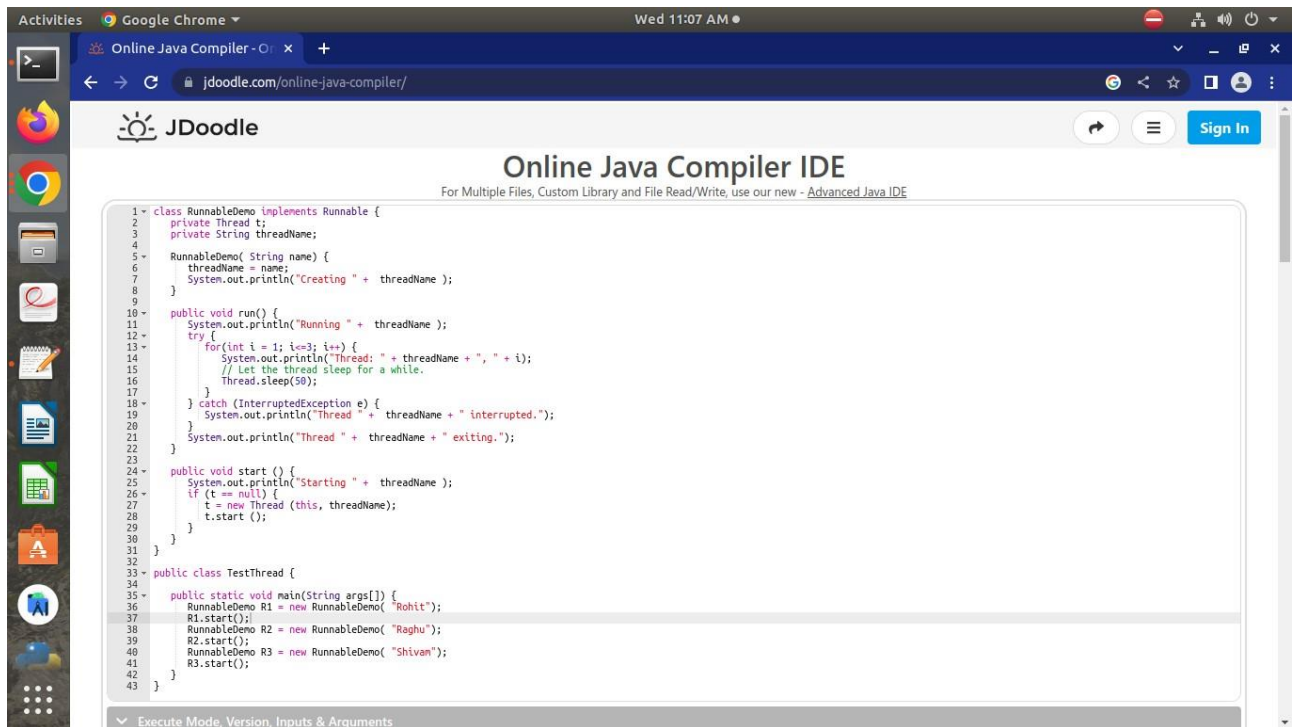
public void start () {
    System.out.println("Starting " + threadName );
    if (t == null) {
        t = new Thread (this, threadName);
        t.start ();
    }
}

}

public class TestThread {
    public static void main(String args[]) {
        ThreadDemo T1 = new ThreadDemo( "Rohit");
        T1.start();
        ThreadDemo T2 = new ThreadDemo( "Raghu");
        T2.start();
        ThreadDemo T3 = new ThreadDemo( "Shivam");
        T3.start();
    }
}

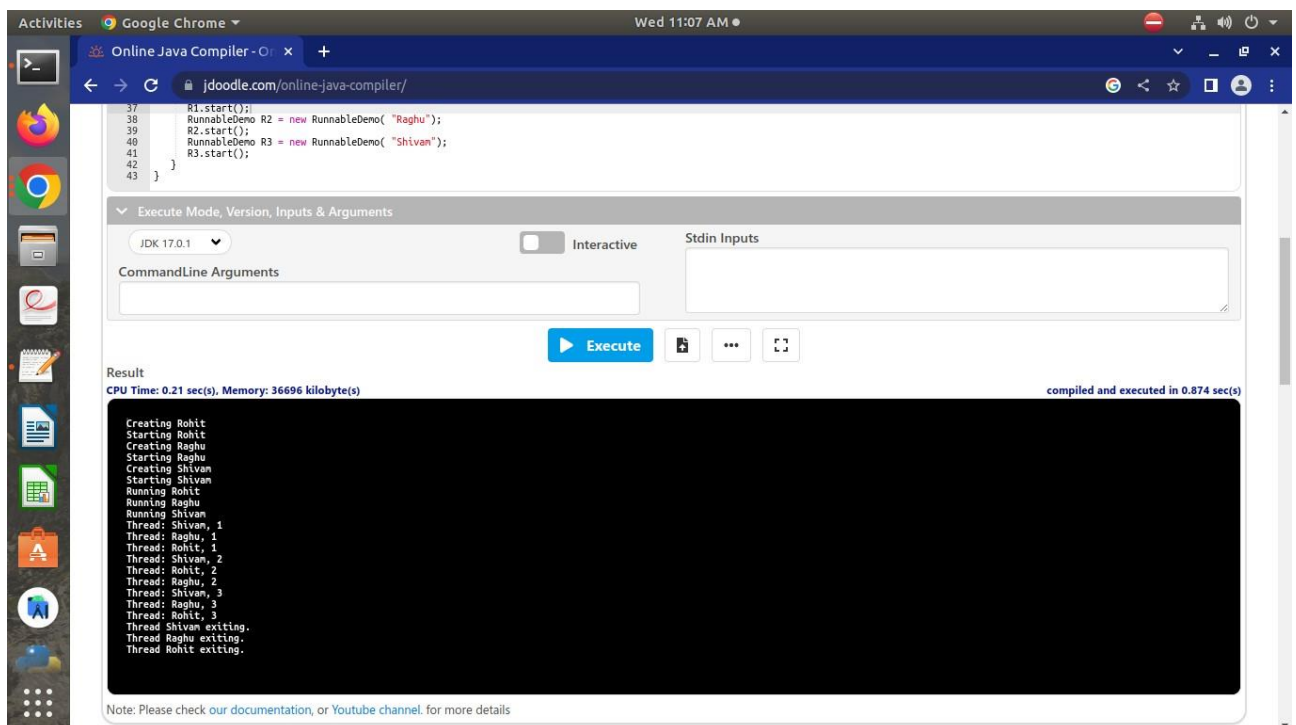
```

Output:



The screenshot shows the JDoodle Online Java Compiler IDE interface. The code defines a `RunnableDemo` class that implements the `Runnable` interface. It has a `run()` method that prints the thread name, a loop counter, and a sleep time. A `TestThread` class contains the `main` method, which creates and starts three instances of `RunnableDemo` with names "Rohit", "Raghu", and "Shivan".

```
1 class RunnableDemo implements Runnable {
2     private Thread t;
3     private String threadName;
4
5     RunnableDemo( String name) {
6         threadName = name;
7         System.out.println("Creating " + threadName);
8     }
9
10    public void run() {
11        System.out.println("Running " + threadName);
12        try {
13            for(int i = 1; i<=3; i++) {
14                System.out.println("Thread: " + threadName + " " + i);
15                // Let the thread sleep for a while.
16                Thread.sleep(50);
17            }
18        } catch (InterruptedException e) {
19            System.out.println("Thread " + threadName + " interrupted.");
20        }
21        System.out.println("Thread " + threadName + " exiting.");
22    }
23
24    public void start() {
25        System.out.println("Starting " + threadName);
26        if (t == null) {
27            t = new Thread(this, threadName);
28            t.start();
29        }
30    }
31 }
32
33 public class TestThread {
34
35     public static void main(String args[]) {
36         RunnableDemo R1 = new RunnableDemo( "Rohit");
37         R1.start();
38         RunnableDemo R2 = new RunnableDemo( "Raghu");
39         R2.start();
40         RunnableDemo R3 = new RunnableDemo( "Shivan");
41         R3.start();
42     }
43 }
```



The screenshot shows the execution output of the code. The output displays the sequence of operations for each thread: creation, starting, running (with loop iterations), and exiting. The threads are named Rohit, Raghu, and Shivan. The output also includes performance metrics: CPU Time: 0.21 sec(s), Memory: 36696 kilobyte(s), and a note that the code was compiled and executed in 0.874 sec(s).

```
37     R1.start();
38     RunnableDemo R2 = new RunnableDemo( "Raghu");
39     R2.start();
40     RunnableDemo R3 = new RunnableDemo( "Shivan");
41     R3.start();
42 }
43 }
```

Execute Mode, Version, Inputs & Arguments

JDK 17.0.1 Interactive Stdin Inputs

CommandLine Arguments

Execute

Result

CPU Time: 0.21 sec(s), Memory: 36696 kilobyte(s) compiled and executed in 0.874 sec(s)

```
Creating Rohit
Starting Rohit
Creating Raghu
Starting Raghu
Creating Shivan
Starting Shivan
Running Rohit
Running Raghu
Running Shivan
Thread: Shivan, 1
Thread: Raghu, 1
Thread: Rohit, 1
Thread: Shivan, 2
Thread: Rohit, 2
Thread: Raghu, 2
Thread: Shivan, 3
Thread: Rohit, 3
Thread: Shivan exiting.
Thread Raghu exiting.
Thread Rohit exiting.
```

Note: Please check our documentation, or Youtube channel. for more details

Conclusion:

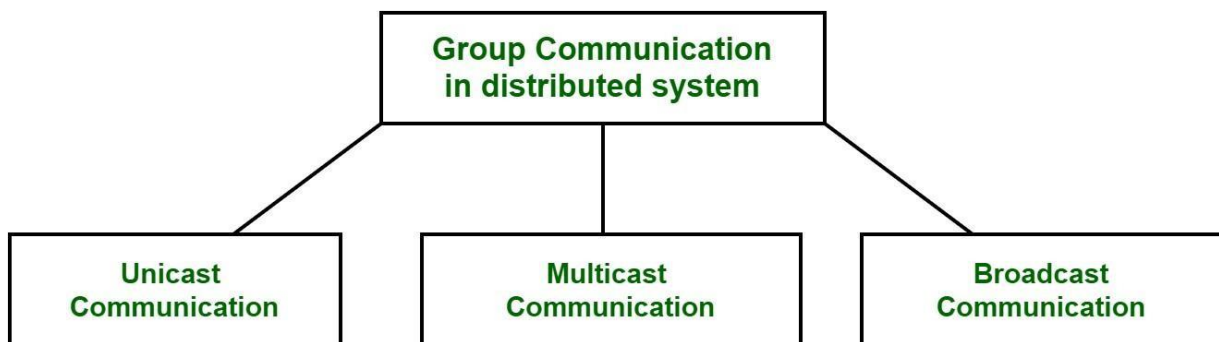
Thus we had Implemented the Multiple Thread application using java.

Experiment No: 3

Aim: Write a program to implement Group Communication in a distributed system.

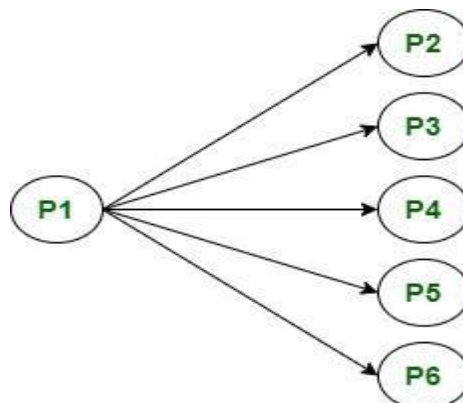
Theory:

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 increasing the overall performance of the system.



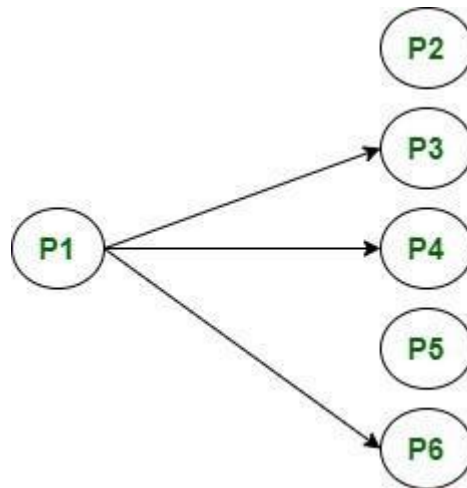
Types of Group Communication in a Distributed System:

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 the 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.



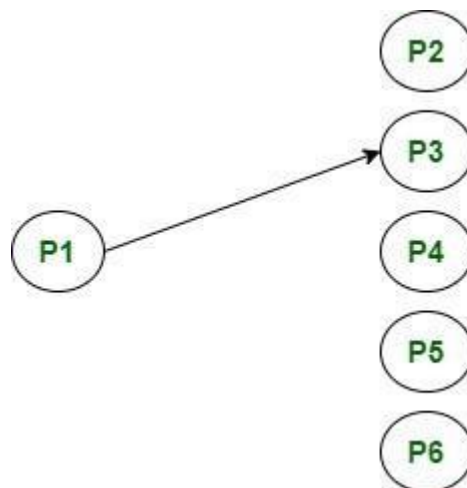
A broadcast Communication: P1 process communicating with every process in the system

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 the problem of a high workload on the host system and redundant information from processes in the system. Multitasking can significantly decrease time taken for message handling.



A multicast Communication: P1 process communicating with only a group of the process in the system

- **Unicast Communication :** When the host process tries to communicate with a single process in a distributed system at the same time. Although, the 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 the exact process and then exchange information/data.



A unicast Communication: P1 process communicating with only P3 process

Program and Output:

Server.py

```
import socket

localIP  = "127.0.0.1"
localPort = 20001
bufferSize = 1024

# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
#UDPServerSocket2 = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))
#UDPServerSocket2.bind((localIP, localPort))

print("UDP server up and listening")

# Listen for incoming datagrams
while(True):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]
    m=message.decode()

    clientMsg = "Message from Client "+m[len(m)-1]+": "+m[0:len(m)-1]
    clientIP = "Client IP Address: {}".format(address)

    print(clientMsg)
    #print(clientIP)

    msgFromServer = input("Enter your message for client "+m[len(m)-1]+": ")
    bytesToSend = str.encode(msgFromServer)

    # Sending a reply to client
    UDPServerSocket.sendto(bytesToSend, address)
```

```
server.py x client1.py client2.py client3.py
server.py > ...
1 import socket
2 localIP = "127.0.0.1"
3 localPort = 20001
4 bufferSize = 1024
5 # Create a datagram socket
6 UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
7 #UDPServerSocket2 = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
8 # Bind to address and ip
9 UDPServerSocket.bind((localIP, localPort))
10 #UDPServerSocket2.bind((localIP, localPort))
11 print("UDP server up and listening")
12 # Listen for incoming datagrams
13 while(True):
14     bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
15     message = bytesAddressPair[0]
16     address = bytesAddressPair[1]
17     m=message.decode()
18     clientMsg = "Message from Client "+m[len(m)-1]+": "+m[0:len(m)-1]
19     clientIP = "Client IP Address: {}".format(address)
20     print(clientMsg)
21     #print(clientIP)
22     msgFromServer = input("Enter your message for client "+m[len(m)-1]+": ")
23     bytesToSend = str.encode(msgFromServer)
24     # Sending a reply to client
25     UDPServerSocket.sendto(bytesToSend, address)
26
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** JUPYTER

```
python
python
python
python
PS C:\Users\Home\Downloads\Multi-Client-Socket-using-UDP-master> python server.py
UDP server up and listening
Message from Client 1: rohit
Enter your message for client 1: sabat
Message from Client 2: raghuwardayal
Enter your message for client 2: maurya
Message from Client 3: shivam
Enter your message for client 3: singh
[]
```

Client1.py

```
import socket

import time

while True:

    msgFromClient = input("Enter your message :")

    bytesToSend = str.encode(msgFromClient + "1")

    serverAddressPort = ("127.0.0.1", 20001)

    bufferSize = 1024

    # Create a UDP socket at client side

    UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

    # Send to server using created UDP socket

    UDPClientSocket.sendto(bytesToSend, serverAddressPort)

    msgFromServer = UDPClientSocket.recvfrom(bufferSize)
```

```

msg = "Message from Server :{}".format(msgFromServer[0].decode())

#time.sleep(5)

print(msg)

```

```

server.py  client1.py x  client2.py  client3.py
client1.py > ...
1  import socket
2  import time
3  while True:
4      msgFromClient    = input("Enter your message :")
5      bytesToSend      = str.encode(msgFromClient + "1")
6      serverAddressPort = ("127.0.0.1", 20001)
7      bufferSize       = 1024
8      # Create a UDP socket at client side
9      UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
10     # Send to server using created UDP socket
11     UDPClientSocket.sendto(bytesToSend, serverAddressPort)
12     msgFromServer = UDPClientSocket.recvfrom(bufferSize)
13     msg = "Message from Server :{}".format(msgFromServer[0].decode())
14     #time.sleep(5)
15     print(msg)
16
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER
python
python
python
python
PS C:\Users\Home\Downloads\Multi-Client-Socket-using-UDP-master> python client1.py
Enter your message :rohit
Message from Server :sabot
Enter your message :

```

Client2.py

```

import socket

import time

while True:

    msgFromClient    = input("Enter your message :")

    bytesToSend      = str.encode(msgFromClient + "2")

    serverAddressPort = ("127.0.0.1", 20001)

    bufferSize       = 1024

# Create a UDP socket at client side

    UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Send to server using created UDP socket

    UDPClientSocket.sendto(bytesToSend, serverAddressPort)

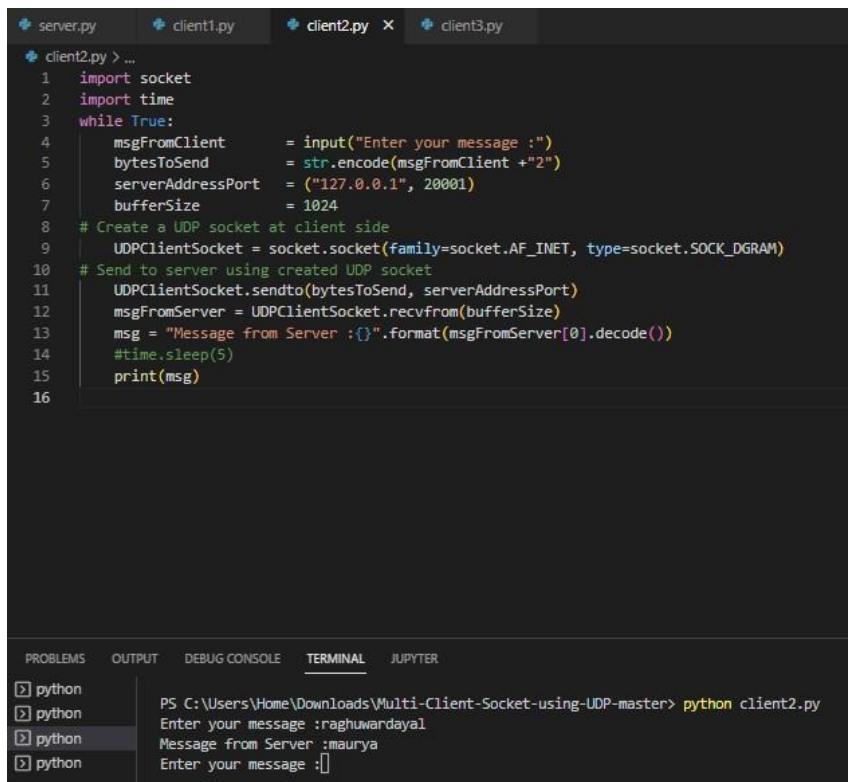
    msgFromServer = UDPClientSocket.recvfrom(bufferSize)

    msg = "Message from Server :{}".format(msgFromServer[0].decode())

    #time.sleep(5)

    print(msg)

```



The screenshot shows a code editor with four tabs: server.py, client1.py, client2.py (active), and client3.py. The client2.py file contains the following Python code:

```
1 import socket
2 import time
3 while True:
4     msgFromClient = input("Enter your message :")
5     bytesToSend = str.encode(msgFromClient + "2")
6     serverAddressPort = ("127.0.0.1", 20001)
7     bufferSize = 1024
8     # Create a UDP socket at client side
9     UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
10    # Send to server using created UDP socket
11    UDPClientSocket.sendto(bytesToSend, serverAddressPort)
12    msgFromServer = UDPClientSocket.recvfrom(bufferSize)
13    msg = "Message from Server :{}".format(msgFromServer[0].decode())
14    #time.sleep(5)
15    print(msg)
16
```

Below the code editor is a terminal window with the following output:

```
PS C:\Users\Home\Downloads\Multi-Client-Socket-using-UDP-master> python client2.py
Enter your message :raghuwardayal
Message from Server :maurya
Enter your message :

```

Client3.py

import socket

import time

while True:

msgFromClient = input("Enter your message :")

bytesToSend = str.encode(msgFromClient + "3")

serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

Create a UDP socket at client side

UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

Send to server using created UDP socket

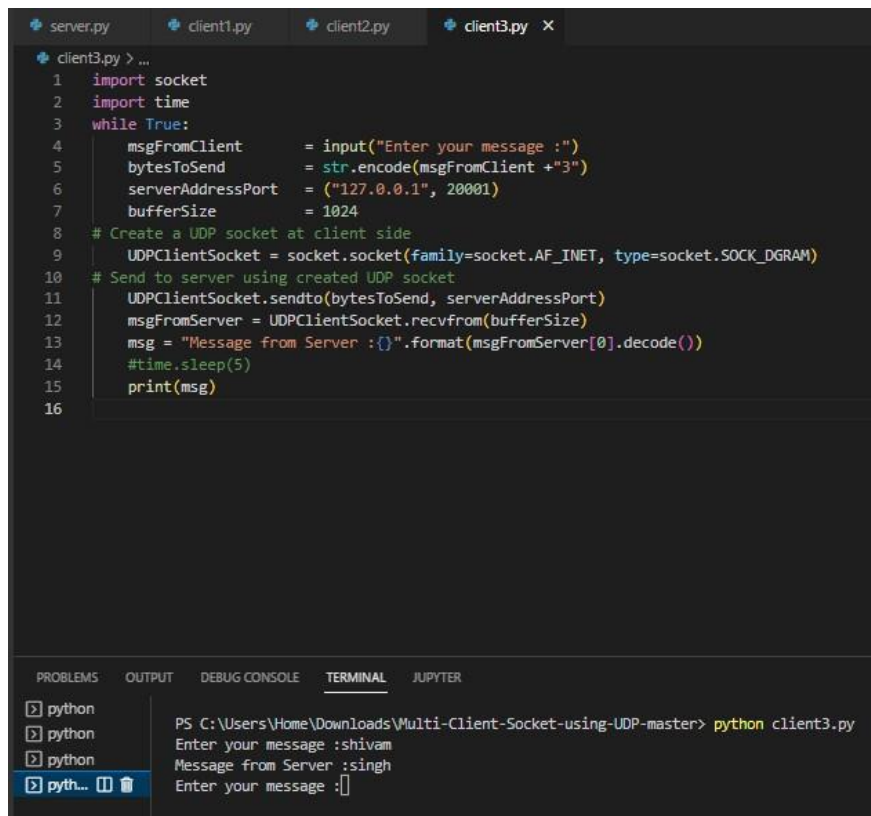
UDPClientSocket.sendto(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.recvfrom(bufferSize)

msg = "Message from Server :{}".format(msgFromServer[0].decode())

#time.sleep(5)

print(msg)



```
server.py client1.py client2.py client3.py X
client3.py > ...
1 import socket
2 import time
3 while True:
4     msgFromClient = input("Enter your message :")
5     bytesToSend = str.encode(msgFromClient + "3")
6     serverAddressPort = ("127.0.0.1", 20001)
7     bufferSize = 1024
8     # Create a UDP socket at client side
9     UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
10    # Send to server using created UDP socket
11    UDPClientSocket.sendto(bytesToSend, serverAddressPort)
12    msgFromServer = UDPClientSocket.recvfrom(bufferSize)
13    msg = "Message from Server :{}".format(msgFromServer[0].decode())
14    #time.sleep(5)
15    print(msg)
16
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** JUPYTER

- python
- python
- python
- python... [icon] [icon]

```
PS C:\Users\Home\Downloads\Multi-Client-Socket-using-UDP-master> python client3.py
Enter your message :shivam
Message from Server :singh
Enter your message :
```

Conclusion:

Thus we have successfully implemented Group Communication in a distributed system.

Experiment No: 4

Aim: Implementation of a Load Balancing Algorithm.

Theory:

Load balancing is a technique used in distributed systems to distribute workload among multiple nodes or servers to improve efficiency and avoid overloading any single node. There are several load balancing algorithms used in distributed systems, including:

- **Round Robin:** This algorithm distributes workload evenly across a set of servers in a cyclic manner. Each new request is forwarded to the next server in the rotation.
- **Least Connections:** This algorithm selects the server with the fewest active connections to distribute the workload to. This ensures that heavily loaded servers are not further burdened.
- **Weighted Round Robin:** This algorithm assigns a weight to each server based on its capacity and distributes workload in proportion to the assigned weights. Servers with higher weights receive more requests.
- **IP Hash:** This algorithm uses the client's IP address to determine which server should handle the request. Requests from the same client are consistently routed to the same server.
- **Least Response Time:** This algorithm measures the response time of each server and directs new requests to the server with the fastest response time.
- **Random:** This algorithm selects a server at random to handle each new request.

This can be useful in systems where all servers have similar capabilities.

The choice of load balancing algorithm depends on the specific requirements of the distributed system. Factors such as the number of servers, server capacity, traffic patterns, and latency can all influence the choice of algorithm.

Program and Output:

```
import itertools

class LoadBalancer:

    def __init__(self, servers):

        self.servers = itertools.cycle(servers)

    def get_server(self):

        return next(self.servers)

if __name__ == '__main__':
```

```
servers = ['server1', 'server2', 'server3']

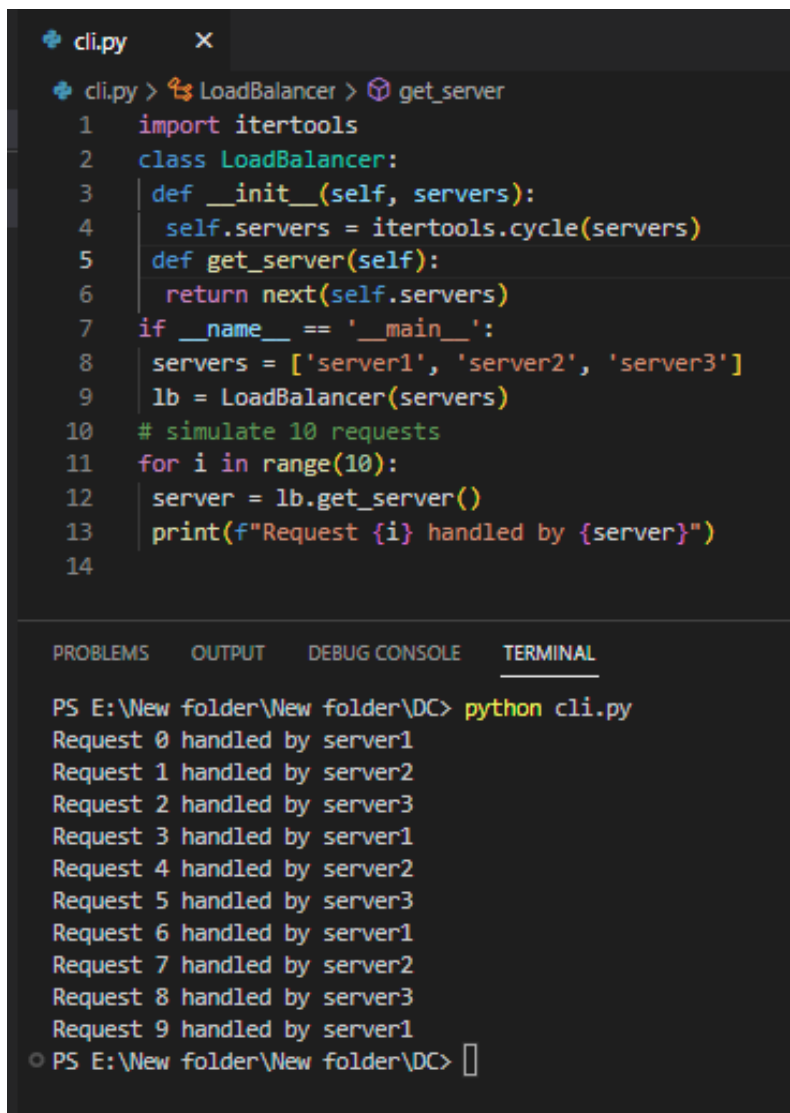
lb = LoadBalancer(servers)

# simulate 10 requests

for i in range(10):

    server = lb.get_server()

    print(f"Request {i} handled by {server}")
```



The screenshot displays a Python IDE with a file named `cli.py`. The code defines a `LoadBalancer` class that uses `itertools.cycle` to rotate through a list of servers. It then simulates 10 requests, each handled by the next server in the cycle. The terminal output shows the sequence of requests and the server that handled each one, demonstrating a round-robin load balancing algorithm.

```
cli.py  x
cli.py > LoadBalancer > get_server
1  import itertools
2  class LoadBalancer:
3      def __init__(self, servers):
4          self.servers = itertools.cycle(servers)
5      def get_server(self):
6          return next(self.servers)
7  if __name__ == '__main__':
8      servers = ['server1', 'server2', 'server3']
9      lb = LoadBalancer(servers)
10 # simulate 10 requests
11 for i in range(10):
12     server = lb.get_server()
13     print(f"Request {i} handled by {server}")
14

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS E:\New folder\New folder\DC> python cli.py
Request 0 handled by server1
Request 1 handled by server2
Request 2 handled by server3
Request 3 handled by server1
Request 4 handled by server2
Request 5 handled by server3
Request 6 handled by server1
Request 7 handled by server2
Request 8 handled by server3
Request 9 handled by server1
PS E:\New folder\New folder\DC> 
```

Conclusion:

Thus we had Successfully Implemented Load Balancing Algorithm.

Experiment No: 5

Aim: Implementation a Clock Synchronization algorithms

Theory:

A Distributed System is a collection of computers connected via the high speed communication network. 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 a need for 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.

- **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.
- **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.

- **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 a 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.
- **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.

Lamport's Logical Clock

Lamport's Logical Clock was created by Leslie Lamport. It is a procedure to determine the order of events occurring. It provides a basis for the more advanced Vector Clock Algorithm. Due to the absence of a Global Clock in a Distributed Operating System Lamport Logical Clock is needed.

Algorithm:

- Happened before relation(\rightarrow): $a \rightarrow b$, means 'a' happened before 'b'.
- Logical Clock: The criteria for the logical clocks are:
 - [C1]: $C_i(a) < C_i(b)$, [$C_i \rightarrow$ Logical Clock, If 'a' happened before 'b', then time of 'a' will be less than 'b' in a particular process.]
 - [C2]: $C_i(a) < C_j(b)$, [Clock value of $C_i(a)$ is less than $C_j(b)$]

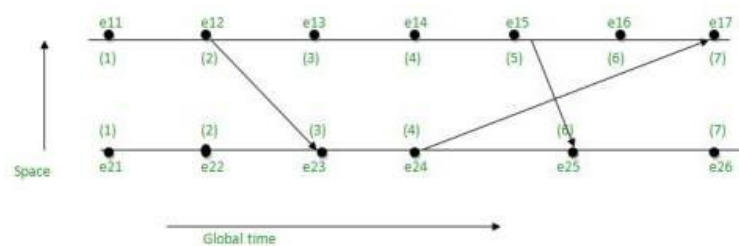
Reference:

- Process: P_i
- Event: E_{ij} , where i is the process in number and j : j th event in the i th process.
- t_m : vector time span for message m .
- C_i vector clock associated with process P_i , the j th element is $C_i[j]$ and contains P_i 's latest value for the current time in process P_j .
- d : drift time, generally d is 1.

Implementation Rules[IR]:

- [IR1]: If $a \rightarrow b$ ['a' happened before 'b' within the same process] then, $C_i(b) = C_i(a) + d$
- [IR2]: $C_j = \max(C_j, t_m + d)$ [If there's more number of processes, then $t_m = \text{value of } C_i(a)$, $C_j = \text{max value between } C_j \text{ and } t_m + d$]

For Example:



- Take the starting value as 1, since it is the 1st event and there is no incoming value at the starting point:
 - $e_{11} = 1$
 - $e_{21} = 1$
- The value of the next point will go on increasing by d ($d = 1$), if there is no incoming value i.e., to follow [IR1].
 - $e_{12} = e_{11} + d = 1 + 1 = 2$

- $e_{13} = e_{12} + d = 2 + 1 = 3$
- $e_{14} = e_{13} + d = 3 + 1 = 4$
- $e_{15} = e_{14} + d = 4 + 1 = 5$
- $e_{16} = e_{15} + d = 5 + 1 = 6$
- $e_{22} = e_{21} + d = 1 + 1 = 2$
- $e_{24} = e_{23} + d = 3 + 1 = 4$
- $e_{26} = e_{25} + d = 6 + 1 = 7$
- When there will be an incoming value, then follow [IR2] i.e., take the maximum value between C_j and $T_m + d$.
- $e_{17} = \max(7, 5) = 7$, [$e_{16} + d = 6 + 1 = 7$, $e_{24} + d = 4 + 1 = 5$, maximum among 7 and 5 is 7]
- $e_{23} = \max(3, 3) = 3$, [$e_{22} + d = 2 + 1 = 3$, $e_{12} + d = 2 + 1 = 3$, maximum among 3 and 3 is 3]
- $e_{25} = \max(5, 6) = 6$, [$e_{24} + 1 = 4 + 1 = 5$, $e_{15} + d = 5 + 1 = 6$, maximum among 5 and 6 is 6]

Program and Output:

Lamport algorithm

```
from multiprocessing import Process, Pipe
from os import getpid
from datetime import datetime

def local_time(counter):
    return '(LAMPORT_TIME={ }, LOCAL_TIME={ })'.format(counter,
                                                         datetime.now())

def calc_recv_timestamp(recv_time_stamp, counter):
    return max(recv_time_stamp, counter) + 1

def event(pid, counter):
    counter += 1
    print('Event happened in { } !'.\
          format(pid) + local_time(counter))
    return counter

def send_message(pipe, pid, counter):
    counter += 1
```

```

    pipe.send(('Empty shell', counter))
    print('Message sent from ' + str(pid) + local_time(counter))
    return counter
def recv_message(pipe, pid, counter):
    message, timestamp = pipe.recv()
    counter = calc_recv_timestamp(timestamp, counter)
    print('Message received at ' + str(pid) + local_time(counter))
    return counter
def process_one(pipe12):
    pid = getpid()
    counter = 0
    counter = event(pid, counter)
    counter = send_message(pipe12, pid, counter)
    counter = event(pid, counter)
    counter = recv_message(pipe12, pid, counter)
    counter = event(pid, counter)
def process_two(pipe21, pipe23):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe21, pid, counter)
    counter = send_message(pipe21, pid, counter)
    counter = send_message(pipe23, pid, counter)
    counter = recv_message(pipe23, pid, counter)
def process_three(pipe32):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe32, pid, counter)
    counter = send_message(pipe32, pid, counter)
if __name__ == '__main__':
    oneandtwo, twoandone = Pipe()
    twoandthree, threeandtwo = Pipe()
    process1 = Process(target=process_one,
                       args=(oneandtwo,))
    process2 = Process(target=process_two,

```

```

        args=(twoandone, twoandthree))

process3 = Process(target=process_three,
        args=(threeandtwo,))

process1.start()
process2.start()
process3.start()
process1.join()
process2.join()
process3.join()

```

The screenshot displays a web browser window with the URL `https://www.tutorialspoint.com/online_python_compiler.php`. The page title is "Online Python Compiler (Interpreter)". The code editor shows the following Python code:

```

1
2 from multiprocessing import Process, Pipe
3 from os import getpid
4 from datetime import datetime
5 def local_time(counter):
6     return ' (LAMPOR_TIME={}, LOCAL_TIME={})'.format(counter,
7                                                         datetime.now())
8 def calc_recv_timestamp(recv_time_stamp, counter):
9     return max(recv_time_stamp, counter) + 1
10 def event(pid, counter):
11     counter += 1
12     print('Event happened in {} !'.\
13           format(pid) + local_time(counter))
14     return counter
15
16 def send_message(pipe, pid, counter):
17     counter += 1
18     pipe.send(('Empty shell', counter))
19     print('Message sent from ' + str(pid) + local_time(counter))
20     return counter
21 def recv_message(pipe, pid, counter):
22     message, timestamp = pipe.recv()
23     counter = calc_recv_timestamp(timestamp, counter)
24     print('Message received at ' + str(pid) + local_time(counter))
25     return counter

```

The terminal output shows the execution of the code, displaying timestamps and process IDs for each event and message sent/received. The output is as follows:

```

Event happened in 330294 ! (LAMPOR_TIME=1, LOCAL_TIME=2023-02-15 05:22:49
.517096)
Message sent from 330294 (LAMPOR_TIME=2, LOCAL_TIME=2023-02-15 05:22:49.517428)
Event happened in 330294 ! (LAMPOR_TIME=3, LOCAL_TIME=2023-02-15 05:22:49
.517449)
Message received at 330295 (LAMPOR_TIME=3, LOCAL_TIME=2023-02-15 05:22:49
.517741)
Message sent from 330295 (LAMPOR_TIME=4, LOCAL_TIME=2023-02-15 05:22:49.517956)
Message sent from 330295 (LAMPOR_TIME=5, LOCAL_TIME=2023-02-15 05:22:49.517987)
Message received at 330294 (LAMPOR_TIME=5, LOCAL_TIME=2023-02-15 05:22:49
.518031)
Event happened in 330294 ! (LAMPOR_TIME=6, LOCAL_TIME=2023-02-15 05:22:49
.518065)
Message received at 330296 (LAMPOR_TIME=6, LOCAL_TIME=2023-02-15 05:22:49
.518148)
Message sent from 330296 (LAMPOR_TIME=7, LOCAL_TIME=2023-02-15 05:22:49.518308)
Message received at 330295 (LAMPOR_TIME=8, LOCAL_TIME=2023-02-15 05:22:49
.518393)

```

Conclusion:

Thus we had Successfully Implemented Load Balancing Algorithm.

Experiment No: 6

Aim: To implement an Election Algorithm.

Theory:

Distributed Algorithm is an algorithm that runs on a distributed system. A distributed system is a collection of independent computers that do not share their memory. Each processor has its own memory and they communicate via communication networks. Communication in networks is implemented in a process on one machine communicating with a process on another machine. Many algorithms used in the distributed system require a coordinator that performs functions needed by other processes in the system.

Election algorithms are designed to choose a coordinator.

Election Algorithms: Election algorithms choose a process from a group of processors to act as a coordinator. If the coordinator process crashes due to some reason, then a new coordinator is elected on another processor. The election algorithm basically determines where a new copy of the coordinator should be restarted. The election algorithm assumes that every active process in the system has a unique priority number. The process with the highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects the active process that has the highest priority number. Then this number is sent to every active process in the distributed system. We have two election algorithms for two different configurations of a distributed system.

The Bully Algorithm – This algorithm applies to a system where every process can send a message to every other process in the system. Algorithm – Suppose process P sends a message to the coordinator.

If the coordinator does not respond to it within a time interval T, then it is assumed that the coordinator has failed.

Now process P sends an election message to every process with a high priority number.

It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.

Then it sends a message to all lower priority number processes that it is elected as their new coordinator.

However, if an answer is received within time T from any other process Q,

(I) Process P again waits for a time interval T' to receive another message from Q that it has been elected as coordinator.

(II) If Q doesn't respond within the time interval T' then it is assumed to have failed and the algorithm is restarted.

The Ring Algorithm –

This algorithm applies to systems organized as a ring(logically or physically). In this algorithm, we assume that the link between the process is unidirectional and every process can message to the process on its right only. A data structure that this algorithm uses is an active list, a list that has a priority number of all active processes in the system.

Algorithm –

If process P1 detects a coordinator failure, it creates a new active list that is empty initially. It sends an election message to its neighbor on right and adds number 1 to its active list.

If process P2 receives a message elected from processes on left, it responds in 3 ways:

- (I) If the message received does not contain 1 in the active list then P1 adds 2 to its active list and forwards the message.
- (II) If this is the first election message it has received or sent, P1 creates a new active list with numbers 1 and 2. It then sends election message 1 followed by 2.
- (III) If Process P1 receives its own election message 1 then the active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects the highest priority number from the list and elects it as the new coordinator.

Program and Output:

```
class Pro:
    def __init__(self, id):
        self.id = id
        self.act = True

class GFG:
    def __init__(self):
        self.TotalProcess = 0
        self.process = []

    def initialiseGFG(self):
        print("No of processes 5")
        self.TotalProcess = 5
        self.process = [Pro(i) for i in range(self.TotalProcess)]

    def Election(self):
        print("Process no " + str(self.process[self.FetchMaximum()].id) + "
fails")
        self.process[self.FetchMaximum()].act = False
        print("Election Initiated by 2")
        initializedProcess = 2

        old = initializedProcess
        newer = old + 1
```

```

        while (True):
            if (self.process[newer].act):
                print("Process " + str(self.process[old].id) + " pass
Election(" + str(self.process[old].id) + ") to" + str(self.process[newer].id))
                old = newer
            newer = (newer + 1) % self.TotalProcess
            if (newer == initializedProcess):
                break

        print("Process " + str(self.process[self.FetchMaximum()].id) + "
becomes coordinator")
        coord = self.process[self.FetchMaximum()].id

        old = coord
        newer = (old + 1) % self.TotalProcess
        while (True):
            if (self.process[newer].act):
                print("Process " + str(self.process[old].id) + " pass
Coordinator(" + str(coord) + ") message to process " +
str(self.process[newer].id))
                old = newer
            newer = (newer + 1) % self.TotalProcess
            if (newer == coord):
                print("End Of Election ")
                break

    def FetchMaximum(self):
        maxId = -9999
        ind = 0
        for i in range(self.TotalProcess):
            if (self.process[i].act and self.process[i].id > maxId):
                maxId = self.process[i].id
                ind = i
        return ind

def main():
    object = GFG()
    object.initialiseGFG()
    object.Election()

if __name__ == "__main__":
    main()

```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\gayat\OneDrive\Desktop\socket> python exp06.py
No of processes 8
Process no 7 fails
Election Initiated by 4
Process 4 pass Election(4) to5
Process 5 pass Election(5) to6
Process 6 pass Election(6) to0
Process 0 pass Election(0) to1
Process 1 pass Election(1) to2
Process 2 pass Election(2) to3
Process 6 becomes coordinator
Process 6 pass Coordinator(6) message to process 0
Process 0 pass Coordinator(6) message to process 1
Process 1 pass Coordinator(6) message to process 2
Process 2 pass Coordinator(6) message to process 3
Process 3 pass Coordinator(6) message to process 4
Process 4 pass Coordinator(6) message to process 5
End Of Election
PS C:\Users\gayat\OneDrive\Desktop\socket> █
```

Conclusion:

Thus we had Successfully Implemented Load Balancing Algorithm.

Experiment No: 7

Aim: Implementation of a Mutual Exclusion Algorithm.

Theory:

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time. Mutual exclusion in single computer system Vs. distributed system:

In a single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variables (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we neither have shared memory nor a common physical clock and therefore we can not solve mutual exclusion problems using shared variables. To eliminate the mutual exclusion problem in distributed systems, an approach based on message passing is used.

A site in a distributed system does not have complete information of the state of the system due to lack of shared memory and a common physical clock. Requirements of Mutual exclusion Algorithm:

- No Deadlock:
Two or more sites should not endlessly wait for any message that will never arrive.
- No Starvation:
Every site who wants to execute a critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section
- Fairness:
Each site should get a fair chance to execute a critical section. Any request to execute a critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.
- Fault Tolerance:
In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Solution to distributed mutual exclusion:

As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

Token Based Algorithm:

- A unique token is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence numbers to order requests for the critical section.
- Each request for the critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique

Example: Suzuki-Kasami's Broadcast Algorithm

Non-token based approach:

- A site communicates with other sites in order to determine which sites should execute the critical section next. This requires exchange of two or more successive rounds of messages among sites.
- This approach uses timestamps instead of sequence numbers to order requests for the critical section.
- Whenever a site makes a request for a critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithms which follow a non-token based approach maintain a logical clock.

Logical clocks get updated according to Lamport's scheme

Example: Lamport's algorithm, Ricart-Agrawala algorithm

Program:

```
rohit.py X
rohit.py > [0] p
1  from multiprocessing import Process, Value
2  from time import sleep
3  n = 10
4  clock = [Value('i', 0) for i in range(n)] # logical clocks for each process
5  request = [Value('i', 0) for i in range(n)] # request flags for each process
6  def request_cs(pid):
7      request[pid].value = 1
8      clock[pid].value += 1
9      for j in range(n):
10         if j != pid:
11             while request[j].value == 1:
12                 if clock[j].value < clock[pid].value:
13                     clock[pid].value += 1
14                 elif (clock[j].value == clock[pid].value) and (j < pid):
15                     clock[pid].value += 1
16                 else:
17                     break
18         print(f"Process {pid} enters critical section at {clock[pid].value}")
19         sleep(1) # simulate critical section
20         print(f"Process {pid} exits critical section")
21         request[pid].value = 0
22  if __name__ == '__main__':
23     processes = [Process(target=request_cs, args=(i,)) for i in range(n)]
24     for p in processes:
25         p.start()
26     for p in processes:
27         p.join()
28
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

● PS E:\New folder\New folder\DC> python rohit.py
Process 2 enters critical section at 1
Process 8 enters critical section at 1
Process 4 enters critical section at 1
Process 1 enters critical section at 1
● Process 0 enters critical section at 1
Process 3 enters critical section at 1
Process 9 enters critical section at 1
Process 5 enters critical section at 1
Process 7 enters critical section at 1
Process 6 enters critical section at 1
Process 2 exits critical section
Process 4 exits critical section
Process 8 exits critical section
Process 1 exits critical section
Process 0 exits critical section
Process 9 exits critical section
Process 3 exits critical section
Process 5 exits critical section
Process 7 exits critical section
Process 6 exits critical section
○ PS E:\New folder\New folder\DC> |
```

Conclusion:

Thus we had Successfully Implemented a Mutual Exclusion Algorithm.

Experiment No. 8

Aim: Implement Deadlock management in Distributed systems.

Theory:

Chandy-Misra-Hass deadlock detection algorithm

Chandy-Misra-Haas's distributed deadlock detection algorithm is an edge chasing algorithm to detect deadlock in distributed systems. It is also considered one of the best deadlock detection algorithms for distributed systems.

Algorithm:

Process of sending probe:

1. If process P_i is locally dependent on itself then declare a deadlock.
2. Else for all P_j and P_k check the following condition:
 - (a) Process P_i is locally dependent on process P_j
 - (b) Process P_j is waiting on process P_k
 - (c) Process P_j and process P_k are on different sites.

If all of the above conditions are true, send probe (i, j, k) to the home site of process P_k .

On the receipt of probe (i, j, k) at home site of process P_k :

1. Process P_k checks the following conditions:
 - (a) Process P_k is blocked.
 - (b) $\text{dependent}_k[i]$ is false.
 - (c) Process P_k has not replied to all requests of process P_j

If all of the above conditions are found to be true then:

1. Set $\text{dependent}_k[i]$ to true.
2. Now, If $k == i$ then, declare the P_i is deadlocked.
3. Else for all P_m and P_n check the following conditions:
 - (a) Process P_k is locally dependent on process P_m and
 - (b) Process P_m is waiting upon process P_n and
 - (c) Process P_m and process P_n are on different sites.
4. Send probe (i, m, n) to the home site of process P_n if the above conditions satisfy.

Thus, the probe message travels along the edges of the transaction wait-for (TWF) graph and when the probe message returns to its initiating process then it is said that deadlock has been detected.

Performance:

The algorithm requires at the most exchange of $m(n-1)/2$ messages to detect deadlock. Here, m is the number of processes and n is the number of sites.

The delay in detecting the deadlock is $O(n)$.

Advantages:

- There is no need for a special data structure. A probe message, which is very small and involves only 3 integers and a two-dimensional boolean array dependent is used in the deadlock detection process.
- At each site, only a little computation is required and overhead is also low
- Unlike other deadlock detection algorithm, there is no need to construct any graph or pass nor to pass graph information to other sites in this algorithm.
- Algorithm does not report any false deadlock (also called phantom deadlock).

Disadvantages:

The main disadvantage of a distributed detection algorithms is that all sites may not aware of the processes involved in the deadlock this makes resolution difficult. Also, proof of correction of the algorithm is difficult.

Program:

```
a = [ [0, 1, 0, 0, 0],
      [0, 0, 1, 0, 0],
      [0, 0, 0, 1, 1],
      [1, 0, 0, 0, 0],
      [0, 0, 0, 0, 0] ]
flag = 0
def aman(a, i, k):
    end = 5
    for x in range(end):
        if(a[k][x] == 1):
            if(i == x):
                print(f' S{k+1} ==> S{x+1}      ({i+1}, {k+1}, {x+1}) ----- >
DEADLOCK DETECTED')
                global flag
                flag = 1
                break
            print(f' S{k+1} ==> S{x+1}      ({i+1}, {k+1}, {x+1})')
            aman(a,i,x)
    x = 0
    end = 5
    i = int(input("Enter Initiator Site No. : "))
    j = i - 1
    print()
    for k in range(end):
        if(a[j][k]==1):
            print(f' S{j+1} ==> s{k+1}      ({i}, {j+1}, {k+1})')
```

```
aman(a,j,k)
if(flag == 0):
    print("\nNO DEADLOCK DETECTED")
```

Output:

```
PS E:\New folder\New folder\DC> python rohit.py
Enter Initiator Site No. : 3
○
S3 ==> s4      (3, 3, 4)
S4 ==> S1      (3, 4, 1)
S1 ==> S2      (3, 1, 2)
S2 ==> S3      (3, 2, 3) -----> DEADLOCK DETECTED
S3 ==> s5      (3, 3, 5)
PS E:\New folder\New folder\DC> █
```

Conclusion:

Thus, we have successfully implemented the Chandy-Misra-Hass deadlock management algorithm.

Experiment No. 9

Aim: Implement a Name Resolution protocol.

Theory:

Name Resolution Protocol

In a distributed system, name resolution protocol is used to translate human-readable names or addresses to machine-readable network addresses. This allows processes or nodes in a distributed system to communicate with each other using their respective names, rather than having to remember and manually enter IP addresses.

There are several name resolution protocols used in distributed systems, including:

1. **Domain Name System (DNS):** DNS is the most widely used name resolution protocol on the internet. It maps domain names to IP addresses and vice versa.
2. **Simple Service Discovery Protocol (SSDP):** SSDP is a protocol used by network devices to discover and communicate with each other. It allows devices to advertise their services and discover other services on the network.
3. **Service Location Protocol (SLP):** SLP is another protocol used to locate network services. It allows clients to search for services based on specific criteria, such as service type, location, or protocol.
4. **Lightweight Directory Access Protocol (LDAP):** LDAP is a protocol used to access and manage directory services. It is commonly used in enterprise environments to store and retrieve information about users, groups, and other network resources.
5. **Network Information Service (NIS):** NIS is a protocol used for name resolution and authentication in Unix-based systems. It allows clients to look up network information, such as user accounts and passwords, from a central server.

Code:

```
import socket
def get_ip_address(url):
    try:
        host_name = socket.gethostbyname(url)
        host_ip = socket.gethostbyname(host_name)
        print("Hostname :", host_name)
        print("IP :", host_ip)
    except:
        print("Unable to get hostname and IP")

if __name__ == '__main__':
    url = "www.ltce.in"
    get_ip_address(url)
```

Output:

```
PS E:\New folder\New folder\DC> python rohit.py
Hostname : 192.46.211.241
IP : 192.46.211.241
PS E:\New folder\New folder\DC> █
```

Conclusion:

Thus, we have implemented a Name Resolution Protocol in Python.

Experiment No. 10

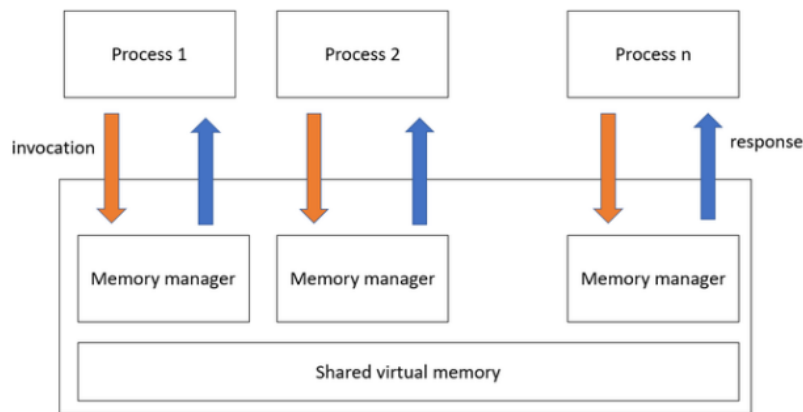
Aim: Distributed Shared Memory.

Theory:

Group

Distributed Shared Memory

DSM is a mechanism that manages memory across multiple nodes and makes inter-process communications transparent to end-users. The applications will think that they are running on shared memory. DSM is a mechanism of allowing user processes to access shared data without using inter-process communications. In DSM every node has its own memory and provides memory read and write services and it provides consistency protocols. The distributed shared memory (DSM) implements the shared memory model in distributed systems but it doesn't have physical shared memory. All the nodes share the virtual address space provided by the shared memory model. The Data moves between the main memories of different nodes.



Types of Distributed shared memory

On-Chip Memory:

- The data is present in the CPU portion of the chip.
- Memory is directly connected to address lines.
- On-Chip Memory DSM is expensive and complex.
- Bus-Based Multiprocessors:
 - A set of parallel wires called a bus act as a connection between CPU and memory.
 - accessing of same memory simultaneously by multiple CPUs is prevented by using some algorithms
- Cache memory is used to reduce network traffic.

Bus-Based Multiprocessors:

- A set of parallel wires called a bus act as a connection between CPU and memory.
- accessing of same memory simultaneously by multiple CPUs is prevented by using some algorithms
- Cache memory is used to reduce network traffic.

Ring-Based Multiprocessors:

- There is no global centralized memory present in Ring-based DSM.
- All nodes are connected via a token passing ring.
- In ring-bases DSM a single address line is divided into the shared area.

Advantages of Distributed shared memory:

- Simpler abstraction: Programmers need not concern about data movement, As the address space is the same it is easier to implement than RPC.
- Easier portability: The access protocols used in DSM allow for a natural transition from sequential to distributed systems. DSM programs are portable as they use a common programming interface.
- Locality of data: Data moved in large blocks i.e. data near to the current memory location that is being fetched, may be needed in the future so it will be also fetched.
- on-demand data movement: It provided by DSM will eliminate the data exchange phase.
- Larger memory space: It provides large virtual memory space, the total memory size is the sum of the memory size of all the nodes, paging activities are reduced.
- Better Performance: DSM improves performance and efficiency by speeding up access to data.
- Flexible communication environment: They can join and leave DSM system without affecting the others as there is no need for sender and receiver to existing,
- Process migration simplified: They all share the address space so one process can easily be moved to a different machine.

Code:

```
import threading
# Shared Memory
memory = {}
# Lock for synchronization
lock = threading.Lock()
# Function to set values in shared memory
def set_value(key, value):
    global memory
    global lock
    lock.acquire()
    memory[key] = value
    lock.release()
# Function to get values from shared memory
def get_value(key):
    global memory
    global lock
    lock.acquire()
```

```

    value = memory.get(key, None)
    lock.release()
    return value
# Function for thread 1
def thread_1():
    set_value("a", 25)
    set_value("b", 18)
    print("Thread 1 sets value of a as 25 and b as 18")
# Function for thread 2
def thread_2():
    value_a = get_value("a")
    value_b = get_value("b")
    print("Thread 2 reads value of a as {} and b as {}".format(value_a, value_b))
    set_value("c", value_a * value_b)
def main_thread():
    thread1 = threading.Thread(target=thread_1)
    thread2 = threading.Thread(target=thread_2)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    value_c = get_value("c")
    print("Main thread reads value of c as {}".format(value_c))
#Main Thread
if __name__ == "__main__":
    main_thread()

```

Output:

```

Thread 1 sets value of a as 25 and b as 18
Thread 2 reads value of a as 25 and b as 18
Main thread reads value of c as 450

```

Conclusion:

Thus, we have implemented Distributed Shared Memory.

Experiment No. 11

Aim: Case Study: CORBA.

Theory:

Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.

CORBA is a standard for distributing objects across networks so that operations on those objects can be called remotely. CORBA is not associated with a particular programming language, and any language with a CORBA binding can be used to call and implement CORBA objects. Objects are described in a syntax called Interface Definition Language (IDL).

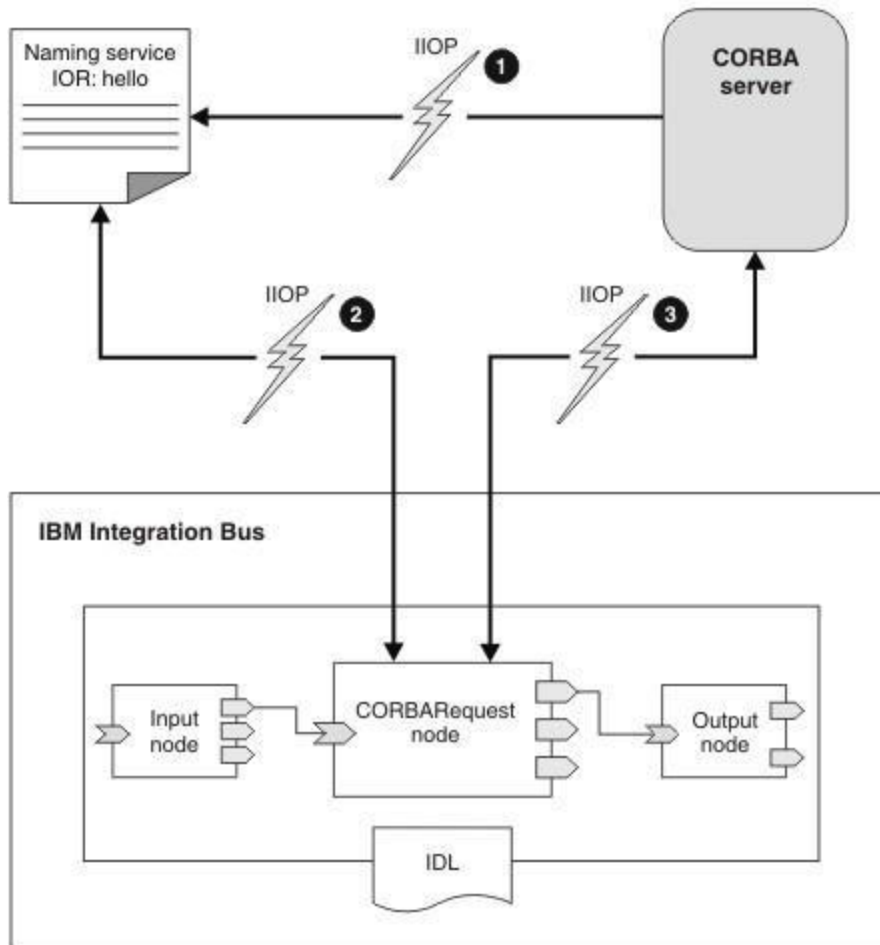
CORBA includes four components:

- **Object Request Broker (ORB)**
The Object Request Broker (ORB) handles the communication, marshaling, and unmarshaling of parameters so that the parameter handling is transparent for a CORBA server and client applications.
- **CORBA server**
The CORBA server creates CORBA objects and initializes them with an ORB. The server places references to the CORBA objects inside a naming service so that clients can access them.
- **Naming service**
The naming service holds references to CORBA objects.
- **CORBARequest node**
The CORBARequest node acts as a CORBA client.

The following diagram shows the layers of communication between IBM® Integration Bus and CORBA.

The diagram illustrates the following steps.

1. CORBA server applications create CORBA objects and put object references in a naming service so that clients can call them.
2. At deployment time, the node contacts a naming service to get an object reference.
3. When a message arrives, the node uses the object reference to call an operation on an object in the CORBA server.



CORBA nodes

CORBA is a standard for distributing objects across networks so that operations on those objects can be called remotely. CORBA objects are described in Interface Definition Language (IDL) files, and these IDL files are used to configure the CORBA message flow nodes. The IDL file is stored in a message set project, in a folder called CORBA IDLs.

An IDL importer imports the IDL file into the message set project and creates the message definition file (.mxsd) in the message set. This message definition file is used for mid-flow validation, ESQL content assist, and the Mapping node.

For each IDL file, a single message definition is created. In the message definition, two messages are created for each operation in the IDL file: one message for the request, and one for the response. The request has a child element for each in and inout parameter; the response has a child element for each input and out parameter, and a child element named “_return” for the return type of the operation.

The name of these elements is based on the interface name and operation name; for example, for the operation sayHello in the Interface Hello, the request element is called Hello.sayHello, and the response element is called Hello.sayHelloResponse. If the interface is contained in a module, the request and response element names are qualified with the names of the modules. For example, if the operation sayHello in the Interface Hello is contained in ModuleB, which in turn is contained in ModuleA, the response element would be called ModuleA.ModuleB.Hello.sayHelloResponse.

When you add a message flow that contains CORBA nodes to a BAR file, all the IDL files that are used by the nodes are added to the BAR file automatically.

CORBA naming service

A CORBA naming service holds CORBA object references.

A CORBA server puts references to CORBA objects inside a naming service so that clients can query the naming service and obtain the object reference, then call operations on the CORBA objects. Typically, a client queries the naming service once, then caches the object reference.

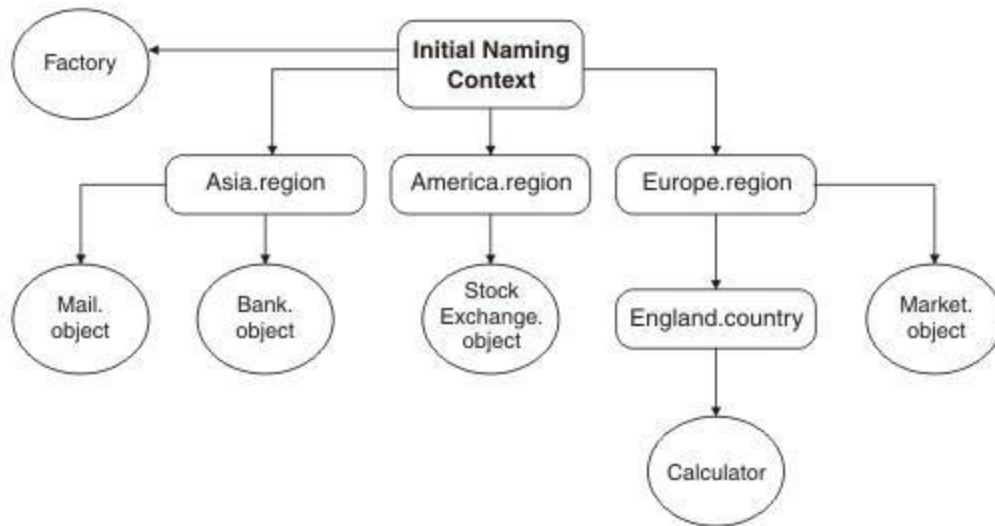
A CORBARequest node is a CORBA client; therefore, when it is deployed, the node contacts a naming service to obtain an object reference. If the object reference is not in the naming service at deployment time, or the naming service that is configured on the node is unavailable, the CORBARequest node issues a warning, and attempts to contact the naming service to get the object reference when it receives a message. If an object reference cannot be acquired from the naming service when the node receives a message, an error is issued. You can specify the location of an object reference by using the properties on the CORBARequest node, or by using the CORBA configurable service. For more information, see CORBARequest node and Defining where the CORBARequest node gets the object reference.

Identifying an object reference in a naming service

Each object in a naming service has a unique name. You must use this name when you configure the Object reference name property on the CORBARequest node.

Naming services are typically arranged in a hierarchy so that names can be given context or scope. The initial naming context is at the top of the hierarchy. Object references can be added to the initial naming context, and additional contexts can exist below it. The number of levels in the hierarchy is unlimited.

Object references and contexts can be assigned a kind to facilitate grouping. The kind is appended to the context in the format context.kind. If you are using IBM® Integration Bus to access an external CORBA application, you need to know the location of the naming service and the name of the object reference in the naming service. The following example shows how to determine the exact string representation of the name.



In the diagram, contexts are represented by squares, and object references are represented by circles.

- An object called Factory is directly attached to the initial naming context.
- Three contexts, with kind region, are also attached to the initial naming context.
- These three contexts each have one or more object references attached to them.
- The Europe context has an England context attached to it, of kind country, which has an object attached to it (Calculator).

The name that you specify when you configure the Object reference name property on the CORBARequest node reflects the position of the object in the hierarchy. The following table shows how to refer to the specific objects in the diagram.

Object	Object reference name
Factory	Factory
Bank	Asia.region/Bank.object
Mail	Asia.region/Mail.object
StockExchange	America.region/StrockExchange.object
Market	Europe.region/Market.object

Calculator	Europe.region/England.country/Calculator
------------	--

All objects in the naming service can be connected directly to the initial naming context; in which case, their names would be in the same format as the Factory object in this example.

CORBA Request node

Use the CORBA Request node to call an external CORBA application over Internet Inter-Orb Protocol (IIOP).

This topic contains the following sections:

- Purpose
- Using this node in a message flow
- Configuring the CORBARequest node
- Terminals and properties

Purpose

You can use the CORBA Request to connect IBM® Integration Bus to CORBA applications. CORBA is a standard for distributing objects across networks so that operations on those objects can be called remotely. CORBA objects are described in Interface Definition Language (IDL) files. You can create a message flow that contains a CORBA Request node, which calls a CORBA server. The message flow uses the IDL file to configure which operation is called on which interface. By using a message flow that includes a CORBA Request node, you can give existing CORBA applications a new external interface; for example, a SOAP interface. The IDL file is stored in a message set project inside a folder called CORBA IDLs, and is used to configure the CORBA Request node in the message flow.

Using this node in a message flow

One possible use of a CORBA Request node is to connect a SOAP-based Web service application to an existing CORBA IIOP application by using a synchronous style of message flow. You can achieve this connection by creating the following message flow:



In this example, the SOAPInput node receives a Web service request, the Mapping node transforms the data in the SOAP message to a CORBA request, and a request is made to the CORBA server. The second Mapping node transforms the response message back into a SOAP reply, which is propagated by the SOAPReply node.

The CORBA Request node is not transactional. After the node has made a request, it cannot roll back the request. The CORBA nodes use the DataObject domain.

The CORBA Request node is contained in the CORBA drawer of the message flow node palette, and is represented in the IBM Integration Toolkit by the following icon:



Look at the following sample to see how to use this node:

- CORBA nodes

You can view information about samples only when you use the product documentation that is integrated with the IBM Integration Toolkit or the online product documentation.

You can run samples only when you use the product documentation that is integrated with the IBM Integration Toolkit.

Configuring the CORBA Request node

When you have put an instance of the CORBA Request node into a message flow, you can configure it; see [Configuring a message flow node](#). The properties of the node are displayed in the Properties view.

All mandatory properties for which you must enter a value (properties that do not have a default value defined) are marked with an asterisk.

Conclusion:

We have successfully learned about the Common Object Request Broker Architecture (CORBA).

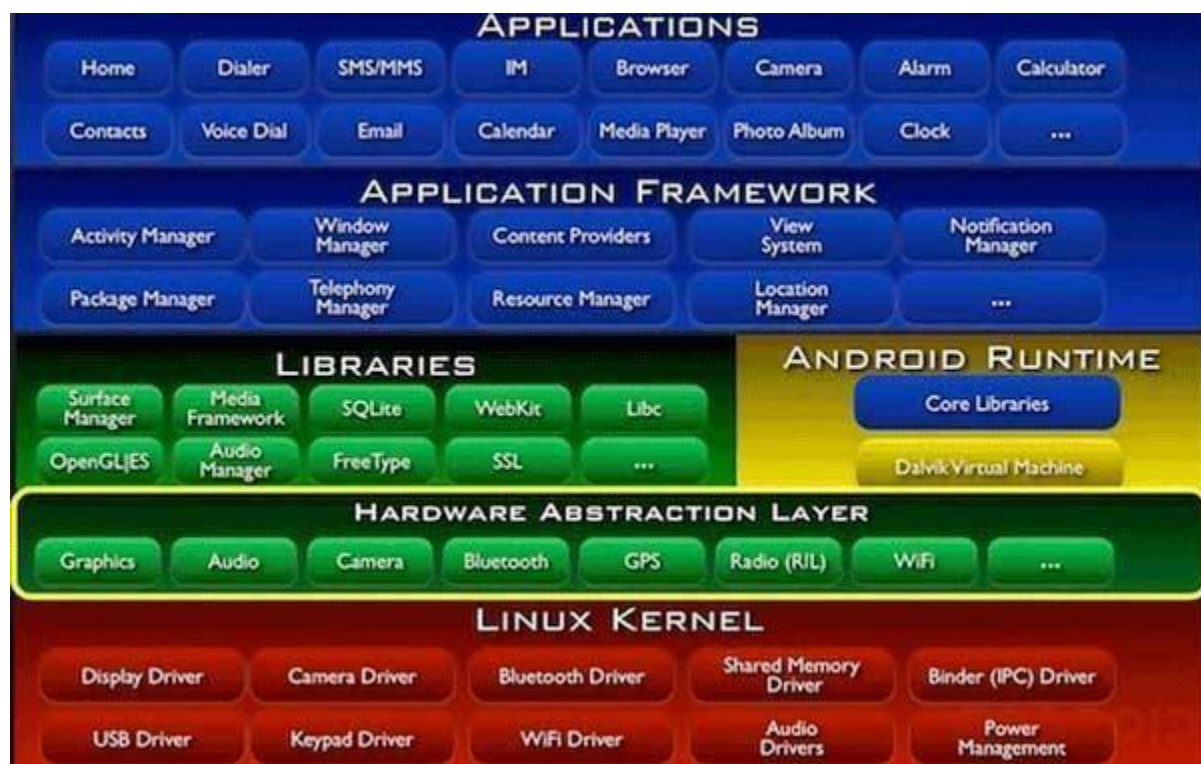
Experiment No. 12

Aim: Case Study: Android Stack .

Theory:

Introduction to Android Stack

The Android software stack generally consists of a Linux kernel and a collection of C/C++ libraries that are exposed through an application framework that provides services, and management of the applications and run time



The Android operating system is like a cake consisting of various layers. Each layer has its own characteristics and purpose— but the layers are not always cleanly separated and often seep into one another.

Although Android is based on Linux, it is not just another flavor of Linux, in the way that Ubuntu, Fedora, or Red Hat are.

Many things you'd expect from a typical Linux distribution aren't available in Android, such as the X11 window manager, the ability to add a person as a Linux user or even the glibc standard C library.

On the other hand, Android adds quite a bit to the Linux kernel, such as an improved power management that is well-suited for mobile battery-powered devices, • a very fast interprocess communication mechanisms

Mechanisms for sand- boxing applications so they are isolated from one another.

Layers in the Android Stack

The Android stack, as the folks over at Google call it, has a number of layers, and each layer groups together several programs. In this tutorial I'll walk you through the various layers in Android stack and the functions they are responsible for.

Following are the different layers in the Android stack:

- Linux Kernel Layer
- Native Layer
- Application Framework Layer
- Applications layer

Kernel Layer



At the bottom of the Android stack is the Linux Kernel. It never really interacts with the users and developers, but is at the heart of the whole system. Its importance stems

from the fact that it provides the following functions in the Android system:

- Hardware Abstraction
- Memory Management Programs
- Security Settings
- Power Management Software
- Other Hardware Drivers (Drivers are programs that control hardware devices.)
- Support for Shared Libraries
- Network Stack

With the evolution of Android, the Linux kernels it runs on have evolved too. Here is a Table highlighting the different Kernel versions.

Android Version	Linux Kernel Version
1.0	2.6.25
1.5 (Cupcake)	2.6.27
1.6 (Donut)	2.6.29
2.2 (Froyo)	2.6.32
2.3 (Gingerbread)	2.6.35
3.0 (Honeycomb)	2.6.36
4.0.x (Ice Cream Sandwich)	3.0.1
4.1./4.2 (Jelly Bean)	3.0.31

The Android system uses a binder framework for its Inter-Process Communication (IPC) mechanism. The binder framework was originally developed as OpenBinder and was used for IPC in BeOS.

Hardware Abstraction Layer

The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or bluetooth module. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.

Native Libraries Layer



The next layer in the Android architecture includes Android's native libraries. Libraries carry a set of instructions to guide the device in handling different types of data. For instance, the playback and recording of various audio and video formats is guided by the Media Framework Library.

Open Source Libraries:

- Surface Manager: composing windows on the screen
- SGL: 2D Graphics
- Open GL|ES: 3D Library

- Media Framework: Supports playback and recording of various audio, video and picture formats.
- Free Type: Font Rendering
- WebKit: Browser Engine
- libc (System C libraries)
- SQLite
- Open SSL

Application Runtime Layer

Located on the same level as the libraries layer, the Android runtime layer includes a set of core Java libraries as well. Android application programmers build their apps using the Java programming language. It also includes the Dalvik Virtual Machine.



What is Dalvik VM?

Dalvik is open-source software. Dan Bornstein, who named it after the fishing village of Dalvík in Eyjafjörður, Iceland, where some of his ancestors lived, originally wrote Dalvik VM. It is the software responsible for running apps on Android devices.

- It is a Register based Virtual Machine.
- It is optimized for low memory requirements.
- It has been designed to allow multiple VM instances to run at once.
- Relies on the underlying OS for process isolation, memory management and threading support.
- Operates on DEX files.

Application Framework Layer

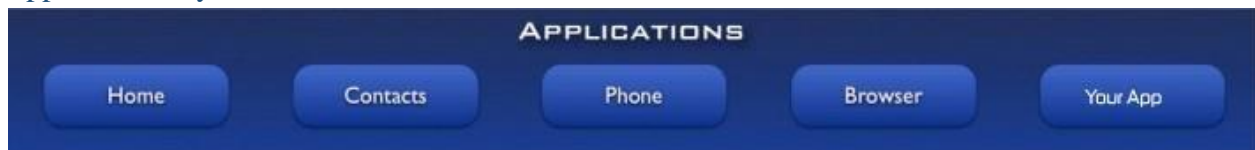


Our applications directly interact with these blocks of the Android architecture. These programs manage the basic functions of phones like resource management, voice call management etc.

Important blocks of Application Framework:

- **Activity Manager:** Manages the activity life cycle of applications. To understand the Activity component in Android in detail [click here](#).
- **Content Providers:** Manage the data sharing between applications. Our Post on Content Provider component describes this in greater detail
- **Telephony Manager:** Manages all voice calls. We use a telephony manager if we want to access voice calls in our application.
- **Location Manager:** Location management, using GPS or cell tower
- **Resource Manager:** Manage the various types of resources we use in our Application

Application Layer



The applications are at the topmost layer of the Android stack. An average user of the Android device would mostly interact with this layer (for basic functions, such as making phone calls, accessing the Web browser etc.). The layers further down are accessed mostly by developers, programmers and the likes.

Several standard applications come installed with every device, such as:

- SMS client app
- Dialer
- Web browser
- Contact manager

Conclusion:

We have successfully learned about the Android Stack. Also about layers of android stack.