**Distributed Computing (DC)**

**Name: Ganesh Borana**

**Roll No: 67**

**Semester: 8**

**Course: Computer Engineering**

**Year: B.E. (2021-22)**

**Professor: Diksha Bhaleroa**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Title** | **PageNo.** |
| 1 | Design calculator using RMI. |  |
| 2 | Implementation of multithread application . |  |
| 3 | Implement Inter-Process Communication. |  |
| 4 | Implementation of Group Communication. |  |
| 5 | Implement Load Balancing algorithm. |  |
| 6 | Implement Name Resolution Protocol (nslookup). |  |
| 7 | Implement Bully Election algorithm. |  |
| 8 | Implement Clock Synchronization algorithm. |  |
| 9 | Implement mutual exclusion algorithm. |  |
| 10 | Case study: Distributed File System – Hadoop |  |
| 11 | Case study: CORBA. |  |
| 12 | Assignment 01 |  |
| 13 | Assignment 02 |  |

**Experiment No: 1**

**Aim:** Design calculator using RMI.

**Theory:**

Remote Method Invocation (RMI) is an API which allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine. Through RMI, object running in a JVM present on a computer (Client side) can invoke methods on an object present in another JVM (Server side). RMI creates a public remote server object that enables client and server side communications through simple method calls on the server object.

**Working of RMI**

The communication between client and server is handled by using two intermediate objects: Stub object (on client side) and Skeleton object (on server side).

**Stub Object**

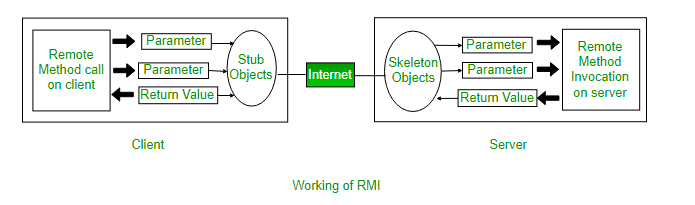
The stub object on the client machine builds an information block and sends this information to the server. The block consists of

* An identifier of the remote object to be used
* Method name which is to be invoked
* Parameters to the remote JVM

**Skeleton Object**

The skeleton object passes the request from the stub object to the remote object.

* It performs following tasks.
* It calls the desired method on the real object present on the server.
* It forwards the parameters received from the stub object to the method.



**Steps to implement Interface**

* Defining a remote interface
* Implementing the remote interface
* Creating Stub and Skeleton objects from the implementation class using rmic (rmi complier)
* Start the rmiregistry
* Create and execute the server application program
* Create and execute the client application program.

Using the RMI concept, Calculator are designed. For calculator, only basic arithmetic operation are implemented i.e addition, subtraction, division, multiplication.

Additional to the program, arithmetic operation computed. From client option are given, on selection of option, client sent to server side to compute that option. After computing, server sends the result to client. On client side, result displays.

**Conclusion:**RMI calculator are successfully implemented, by selecting the appropriate option from client, server computers and pass the result to client.

**Code:**

**Calculator.java:**

public interface Calculator extends java.rmi.Remote {

public long add(long a, long b) throws java.rmi.RemoteException;

public long sub(long a, long b) throws java.rmi.RemoteException;

public long mul(long a, long b) throws java.rmi.RemoteException;

public long div(long a, long b) throws java.rmi.RemoteException;

}

**CalculatorClient.java:**

import java.rmi.Naming;

import java.rmi.RemoteException;

import java.net.MalformedURLException;

import java.rmi.NotBoundException;

import java.util.\*;

public class CalculatorClient {

public static void main(String[] args) {

long num1 = Integer.parseInt(args[0]);

long num2 = Integer.parseInt(args[1]);

try {

Calculator c = (Calculator)

Naming.lookup("rmi://localhost/CalculatorService");

System.out.println("1: sub \t 2: add \t 3: mul \t 4: div ");

System.out.println("Enter the choice :");

int choice;

Scanner sc=new Scanner(System.in);

choice=sc.nextInt();

switch(choice){

case 1:

System.out.println( "The substraction of "+num1 +" and "+num2 +" is: "+ c.sub(num1, num2) );

break;

case 2:

System.out.println( "The addition of "+num1 +" and "+ num2 +"is: "+c.add(num1, num2) );

break;

case 3:

System.out.println( "The multiplication of "+num1 +" and "+num2 +" is: "+c.mul(num1, num2) );

break;

case 4:

System.out.println( "The division of "+num1 +" and "+ num2 +"is: "+c.div(num1, num2) );

break;

}

}

catch (MalformedURLException murle) {

System.out.println();

System.out.println("MalformedURLException");

System.out.println(murle);

}

catch (RemoteException re) {

System.out.println();

System.out.println("RemoteException");

System.out.println(re);

}

catch (NotBoundException nbe) {

System.out.println();

System.out.println("NotBoundException");

System.out.println(nbe);

}

catch (java.lang.ArithmeticException ae) {

System.out.println();

System.out.println("java.lang.ArithmeticException");

System.out.println(ae);

}

}

}

**CalculatorImpl.java:**

public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject implements Calculator {

// Implementations must have an explicit constructor

// in order to declare the RemoteException exception

public CalculatorImpl() throws java.rmi.RemoteException {

super();

}

public long add(long a, long b) throws java.rmi.RemoteException {

return a + b;

}

public long sub(long a, long b) throws java.rmi.RemoteException {

return a - b;

}

public long mul(long a, long b) throws java.rmi.RemoteException {

return a \* b;

}

public long div(long a, long b) throws java.rmi.RemoteException {

return a / b;

}

}

**CalculatorServer.java:**

import java.rmi.Naming;

public class CalculatorServer{

public CalculatorServer(){

try{

Calculator c = new CalculatorImpl();

Naming.rebind("rmi://localhost:1099/CalculatorService", c);

}catch (Exception e){

System.out.println("Trouble: " + e);

}

}

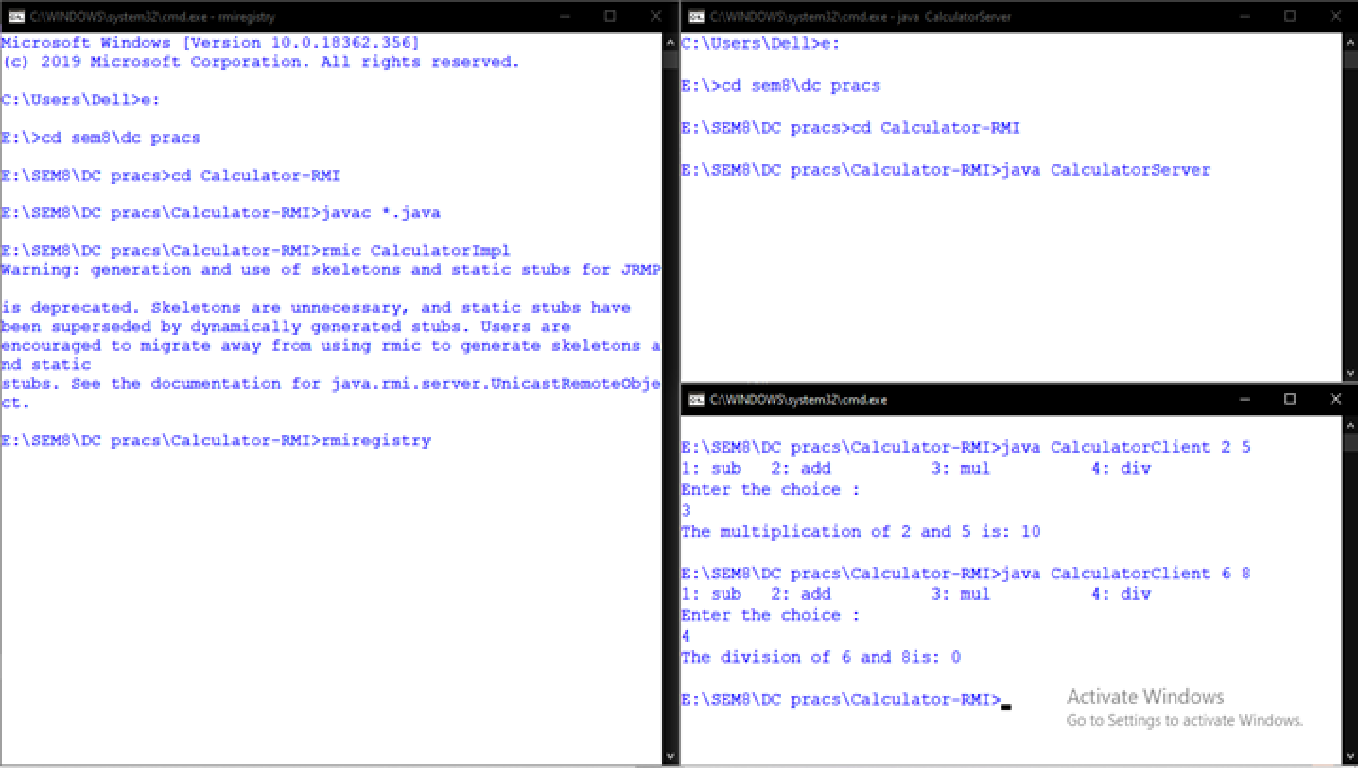
public static void main(String args[]){

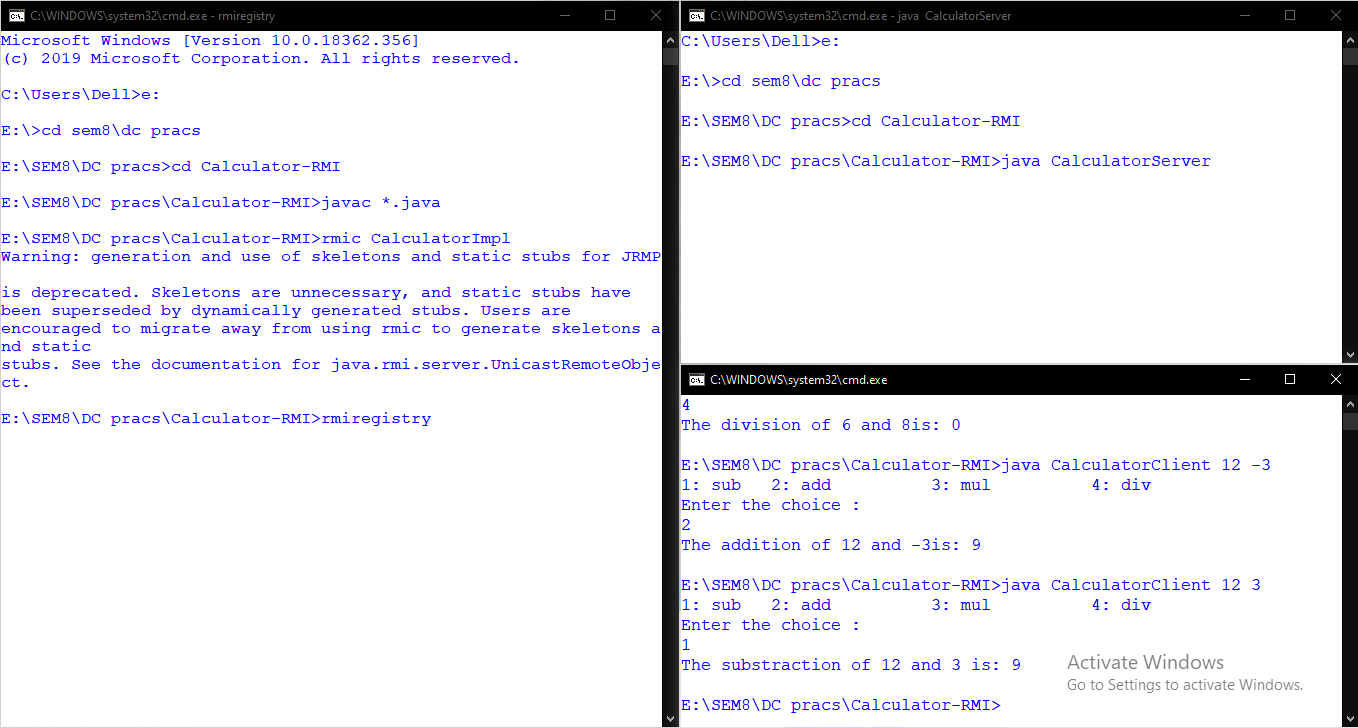
new CalculatorServer();

}

}

OUTPUT:





**Experiment No: 2**

**Aim:** Implementation of multithread application.

**Theory:**

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms:

**Extending the Thread class:**

We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

**Implementing the Runnable Interface:**

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

Here we implemented three sorting algorithms using multithreading.

* Insertion sort : Insertion Sort in java is an efficient sorting algorithm, that creates the final sorted array one element at a time. An element from the input data is removed after every iteration. It is compared to the largest value present in the array and is then moved to the correct position.
* Selection sort : We can create a java program to sort array elements using selection sort. In selection sort algorithm, we search for the lowest element and arrange it to the proper location. We swap the current element with the next lowest number.
* Bubble sort : In bubble sort algorithm, array is traversed from first element to last element. Here, current element is compared with the next element. If current element is greater than the next element, it is swapped.

**Advantages of multithreading:**

* Enhanced performance by decreased development time
* Simplified and streamlined program coding
* Improvised GUI responsiveness
* Simultaneous and parallelized occurrence of tasks

**Disadvantages of multithreading:**

* Complex debugging and testing processes
* Overhead switching of context
* Increased potential for deadlock occurrence
* Increased difficulty level in writing a program
* Unpredictable results

**Conclusion :**

Implementing multithreaded Java code is reasonably straightforward.For application operations where it's okay to interleave user actions, making a GUI multithreaded can be very useful. It can even help to improve end user productivity by allowing a number of application operations to proceed in parallel.

**Code:**

class Bubblesort extends Thread{

public void run(){

sort();

}

void sort()

{

int arr[] = {64,25,12,22,11};

int n = arr.length;

for (int i = 0; i < n-1; i++)

{

int min\_idx = i;

for (int j = i+1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

printArray(arr);

}

void printArray(int arr[])

{

int n = arr.length;

System.out.print("Bubble sort:");

for (int i=0; i<n; ++i)

System.out.print(arr[i]+" ");

System.out.println();

}

}

class Selectionsort extends Thread{

public void run(){

ssort();

}

void ssort()

{

int arr[] = {64,25,12,22,11};

int n = arr.length;

for (int i = 0; i < n-1; i++)

{

int min\_idx = i;

for (int j = i+1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

printArray(arr);

}

void printArray(int arr[])

{

System.out.print("Selection sort:");

int n = arr.length;

for (int i=0; i<n; ++i)

System.out.print(arr[i]+" ");

System.out.println();

}

}

class Insertionsort extends Thread{

public void run(){

Isort();

}

public static void Isort()

{

int a[] = {64,25,12,22,11};

int n=a.length,i,j,p,temp;

try{

for (i = 1;i < n; i++)

{

for (j=i-1; j >=0 && a[j+1]<a[j]; j--)

{

temp=a[j+1];

a[j+1]=a[j];

a[j]=temp;

}

}

Thread.sleep(500);

}

catch(Exception ex){

System.out.println("Exception has " +

"been caught" + ex);

}

printarray(a);

}

public static void printarray(int a[])

{

System.out.print("Insertion sort:");

for(int i=0; i < a.length; i++)

{

System.out.print(a[i]+" ");

}

}

}

class Multi{

public static void main(String args[]){

Bubblesort t1=new Bubblesort();

Selectionsort t2=new Selectionsort();

Insertionsort t3=new Insertionsort();

t1.start();

try

{

t1.join();

}

catch(Exception ex)

{

System.out.println("Exception has " +

"been caught" + ex);

}

t2.start();

try

{

t2.join();

t2.sleep(5000);

}

catch(Exception ex)

{

System.out.println("Exception has " +

"been caught" + ex);

}

t3.start();

try

{

t3.join();

t3.sleep(9000);

}

catch(Exception ex)

{

System.out.println("Exception has " +

"been caught" + ex);

}

}

}

**Output:**

Bubble sort:11 12 22 25 64

Selection sort:11 12 22 25 64

Insertion sort:11 12 22 25 64

**Experiment No: 3**

**Aim:** Implement the Interprocess Communication using socket.

**Theory:**

A process can be of two type:

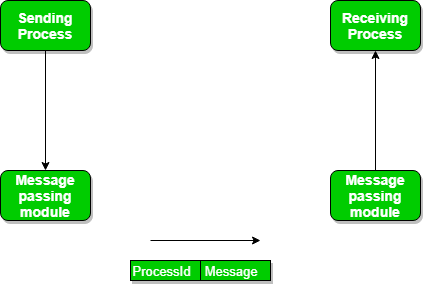
* Independent process.
* Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

* Shared Memory
* Message passing

**Message passing:**

The communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:



* Establish a communication link (if a link already exists, no need to establish it again.)
* Start exchanging messages using basic primitives.  
  We need at least two primitives:  
  – **send**(message, destination) or **send**(message)  
  – **receive**(message, host) or **receive**(message)

**Direct and Indirect Communication link**

* Direct Communication links are implemented when the processes use specific process identifier for the communication but it is hard to identify the sender ahead of time.For example: the print server.
* In-directed Communication is done via a shred mailbox (port), which consists of queue of messages. Sender keeps the message in mailbox and receiver picks them up.

**Examples of IPC systems**

1. Posix: uses shared memory method.
2. Mach: uses message passing
3. Windows XP: uses message passing using local procedural calls

**Conclusion:**

Interprocess communication are successfully implemented using socket.

**OUTPUT:**

**Server :**

import java.io.\*;

import java.net.\*;

public class Server extends Thread {

public final static int DEFAULT\_PORT = 8000;

protected int port;

protected ServerSocket listen\_socket;

public static void fail(Exception e, String msg) {

System.err.println(msg + ": " + e);

System.exit(1);

}

public Server(int port) {

if (port == 0) port = DEFAULT\_PORT;

this.port = port;

try { listen\_socket = new ServerSocket(port); }

catch (IOException e) { fail(e, "Exception creating server socket"); }

System.out.println("Server: listening on port " + port);

this.start();

}

public void run() {

try {

while(true) {

Socket client\_socket = listen\_socket.accept();

Connection c = new Connection(client\_socket);

}

}

catch (IOException e) {

fail(e, "Exception while listening for connections");

}

}

public static void main(String[] args) {

int port = 0;

if (args.length == 1) {

try { port = Integer.parseInt(args[0]); }

catch (NumberFormatException e) { port = 0; }

}

new Server(port);

}

}

class Connection extends Thread {

protected Socket client;

protected BufferedReader in;

protected PrintStream out;

public Connection(Socket client\_socket) {

client = client\_socket;

try {

in = new BufferedReader(new InputStreamReader(client.getInputStream()));

out = new PrintStream(client.getOutputStream());

}

catch (IOException e) {

try { client.close(); } catch (IOException e2) { ; }

System.err.println("Exception while getting socket streams: " + e);

return;

}

this.start();

}

public void run() {

String line;

StringBuffer revline;

int len;

try {

for(;;) {

line = in.readLine();

if (line == null) break;

len = line.length();

revline = new StringBuffer(len);

for(int i = len-1; i >= 0; i--)

revline.insert(len-1-i, line.charAt(i));

out.println(revline);

}

}

catch (IOException e) { ; }

finally { try {client.close();} catch (IOException e2) {;} }

}

}

**Client :**

import java.io.\*;

import java.net.\*;

public class Client {

public static final int DEFAULT\_PORT = 8000;

public static void usage() {

System.out.println("Usage: java Client <hostname> [<port>]");

System.exit(0);

}

public static void main(String[] args) {

int port = DEFAULT\_PORT;

Socket s = null;

if ((args.length != 1) && (args.length != 2)) usage();

if (args.length == 1) port = DEFAULT\_PORT;

else {

try { port = Integer.parseInt(args[1]); }

catch (NumberFormatException e) { usage(); }

}

try {

s = new Socket(args[0], port);

BufferedReader sin = new BufferedReader(new InputStreamReader(s.getInputStream()));

PrintStream sout = new PrintStream(s.getOutputStream());

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Connected to " + s.getInetAddress() + ":"+ s.getPort());

String line;

while(true) {

System.out.print("> ");

System.out.flush();

line = in.readLine();

if (line == null) break;

sout.println(line);

line = sin.readLine();

if (line == null) {

System.out.println("Connection closed by server.");

break;

}

System.out.println(line);

}

}

catch (IOException e) { System.err.println(e); }

finally {

try { if (s != null) s.close(); } catch (IOException e2) { ; }

}

}

}

**Output:**

**Server:**

ganesh@Ganesh:~$ javac Server.java

ganesh@Ganesh:~$ java Server

Server: listening on port 8000

**Client:**

ganesh@Ganesh:~$ javac Client.java

ganesh@Ganesh:~$ java Client localhost 8000

Connected to localhost/127.0.0.1:8000

> this is IPC client server program

margorp revres tneilc CPI si siht>

**Experiment No: 4**

**Aim:** Implementation of Group Communication

**Theory:**

A group chat application using MulticastSocket class is discussed. A MulticastSocket is a (UDP) DatagramSocket, with additional capabilities for joining “groups” of other multicast hosts on the internet.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

* **Fault tolerance based on replicated services:** A replicated service consists of a group ofservers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
* **Discovering services in spontaneous networking:** Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
* **Better performance through replicated data:** Data are replicated to increase the performanceof a service – in some cases replicas of the data are placed in users’ computers. Each time the data changes, the new value is multicast to the processes managing the replicas.
* **Propagation of event notifications:** Multicast to a group may be used to notify processes whensomething happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

**Group communication** is the process of exchanging messages between two systems continuously. Anyone can break the communication. Both systems come with the following same responsibilities.

* **Reading from keyboard**. Uses an input stream like BufferedReader connected to System.in.
* **Sending data** to the other system what is read from keyboard. Uses an output stream like PrintWriter connected to getOutputStream() method of Socket.
* **Receiving data** from the other system. Uses an input stream like BufferedReader connected to getInputStream() method of Socket.

As the responsibilities are same, both client and server programs contain the same stream objects and same code. The order of using stream objects varies in the **while loop**.

**Conclusion:**  A group chat application using MulticastSocket class is implemented. A MulticastSocket is a (UDP) DatagramSocket, with additional capabilities for joining “groups” of other multicast hosts on the internet.

**CODE:**

import java.net.\*;

import java.io.\*;

import java.util.\*;

public class group\_chat {

private static final String TERMINATE = "Exit";

static String name;

static volatile boolean finished = false;

public static void main(String[] args) {

if (args.length != 2)

System.out.println("Two arguments required: <multicast-host> <port-number>");

else{

try{

InetAddress group = InetAddress.getByName(args[0]);

int port = Integer.parseInt(args[1]);

Scanner sc = new Scanner(System.in);

System.out.print("Enter your name: ");

name = sc.nextLine();

MulticastSocket socket = new MulticastSocket(port);

// Since we are deploying

socket.setTimeToLive(0);

//this on localhost only (For a subnet set it as 1)

socket.joinGroup(group);

Thread t = new Thread(new

ReadThread(socket,group,port));

// Spawn a thread for reading messages

t.start();

// sent to the current group

System.out.println("Start typing messages...\n");

while(true) {

String message;

message = sc.nextLine();

if(message.equalsIgnoreCase(group\_chat.TERMINATE)) {

finished = true;

socket.leaveGroup(group);

socket.close();

break;

}

message = name + ": " + message;

byte[] buffer = message.getBytes();

DatagramPacket datagram = new

DatagramPacket(buffer,buffer.length,group,port);

socket.send(datagram);

}

}

catch(SocketException se) {

System.out.println("Error creating socket");

se.printStackTrace();

}

catch(IOException ie) {

System.out.println("Error reading/writing from/to socket");

ie.printStackTrace();

}

}

}

}

class ReadThread implements Runnable {

private MulticastSocket socket;

private InetAddress group;

private int port;

private static final int MAX\_LEN = 1000;

ReadThread(MulticastSocket socket,InetAddress group,int port) {

this.socket = socket;

this.group = group;

this.port = port;

}

@Override

public void run() {

while(!group\_chat.finished) {

byte[] buffer = new byte[ReadThread.MAX\_LEN];

DatagramPacket datagram = new

DatagramPacket(buffer,buffer.length,group,port);

String message;

try{

socket.receive(datagram);

message = new

String(buffer,0,datagram.getLength(),"UTF-8");

if(!message.startsWith(group\_chat.name))

System.out.println(message);

}

catch(IOException e) {

System.out.println("Socket closed!");

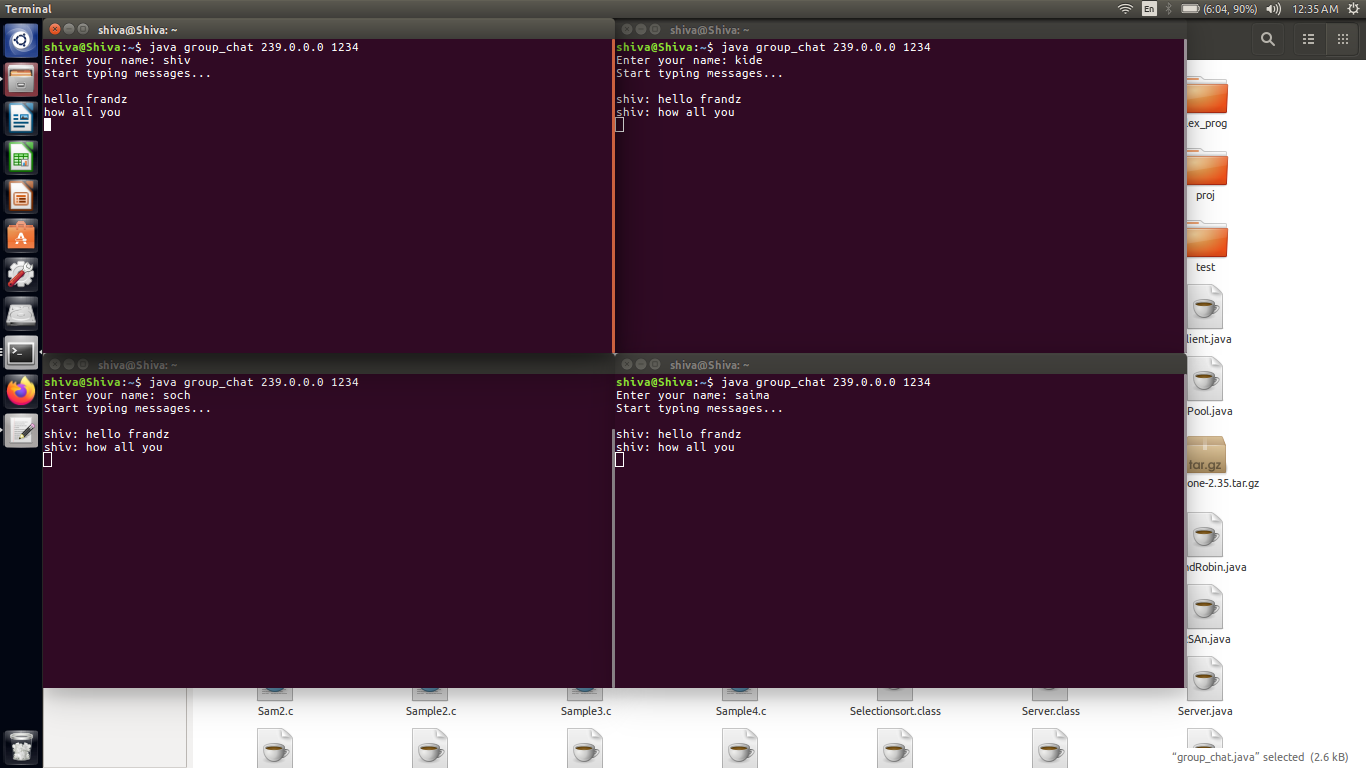
}

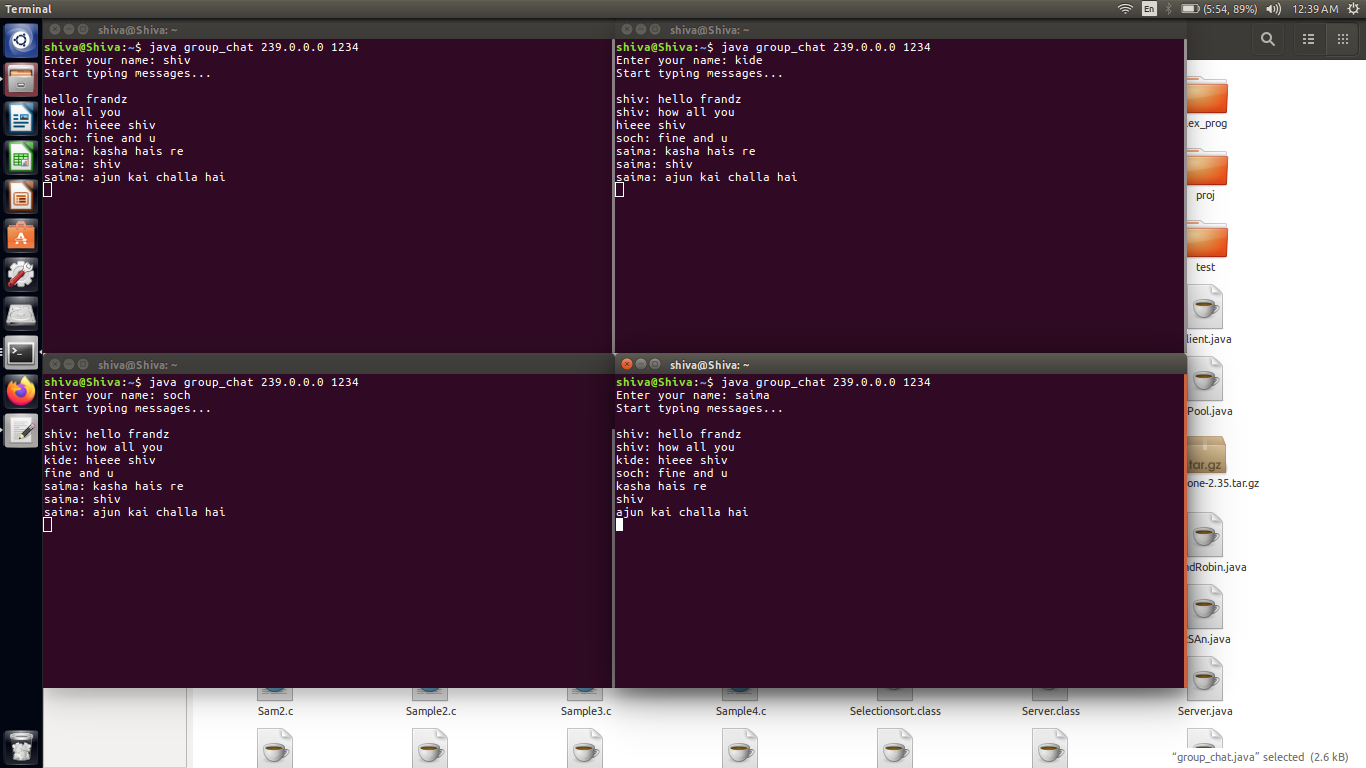
}

}

}

**OUTPUT:**





**Experiment No: 5**

**Aim:** Load Balancing Algorithm

**Theory:**

**Load balancing** refers to the process of distributing a set of [tasks](https://en.wikipedia.org/wiki/Task_(computing)" \o "Task (computing)) over a set of [resources](https://en.wikipedia.org/wiki/System_resource" \o "System resource) (computing units), with the aim of making their overall processing more efficient. Load balancing techniques can optimise the response time for each task, avoiding unevenly overloading compute nodes while other compute nodes are left idle.

Load balancing is the subject of research in the field of [parallel computers](https://en.wikipedia.org/wiki/Parallel_computers" \o "Parallel computers). Two main approaches exist: static algorithms, which do not take into account the state of the different machines, and dynamic algorithms, which are usually more general and more efficient, but require exchanges of information between the different computing units, at the risk of a loss of efficiency.

**Type of load-balancing algorithms**

**Static versus Dynamic:**

Static algorithms use only information about the average behavior of the system Static algorithms ignore the current state or load of the nodes in the system Dynamic algorithms collect state information and react to system state if it changed Static algorithms are much more simpler Dynamic algorithms are able to give significantly better performance

**Load-balancing approach Type of static load-balancing algorithms**

**Deterministic versus Probabilistic**

Deterministic algorithms use the information about the properties of the nodes and the characteristic of processes to be scheduled Probabilistic algorithms use information of static attributes of the system (e.g. number of nodes, processing capability, topology) to formulate simple process placement rules Deterministic approach is difficult to optimize Probabilistic approach has poor performance

**Load-balancing approach Type of dynamic load-balancing algorithms**

**Centralized versus Distributed**

Centralized approach collects information to server node and makes assignment decision Distributed approach contains entities to make decisions on a predefined set of nodes Centralized algorithms can make efficient decisions, have lower fault-tolerance Distributed algorithms avoid the bottleneck of collecting state information and react faster

**Load-balancing approach Type of distributed load-balancing algorithms**

**Cooperative versus Non-cooperative**

In Non-cooperative algorithms entities act as autonomous ones and make scheduling decisions independently from other entities

In Cooperative algorithms distributed entities cooperate with each other Cooperative algorithms are more complex and involve larger overhead Stability of Cooperative algorithms are better.

**Issues in load balancing:**

* **Load estimation policy:** determines how to estimate the workload of a node
* **Process transfer policy:** determines whether to execute a process locally or remote
* **State information exchange policy:** determines how to exchange load information among nodes
* **Location policy:** determines to which node the transferable process should be sent
* **Priority assignment policy:** determines the priority of execution of local and remote processes
* **Migration limiting policy:** determines the total number of times a process can migrate

**Conclusion:**

In addition to efficient problem solving through parallel computations, load balancing algorithms are widely used in HTTP request management where a site with a large audience must be able to handle requests per second.

**CODE:**

import java.io.\*;

import java.util.Scanner;

class LoadBalancer{

static void printLoad(int servers, int processes){

int each = processes/servers;

int extra = processes%servers;

int[] total = new int[servers];

for(int i=0;i<servers;i++){

total[i]=each;

}

if(extra>0 & extra<each){

total[0]+=extra;

}else{

int i=0;

while(extra!=0){

total[i]+=each;

extra-=each;

i++;

}

}

for (int i=0;i<servers ;i++ ) {

System.out.println("Server "+(char)('A'+i)+" has "+total[i]+" processes");

}

}

public static void main(String args[]){

Scanner sc=new Scanner(System.in);

System.out.print("Enter The number of servers and processes: ");

int servers = sc.nextInt();

int processes = sc.nextInt();

while(true){

printLoad(servers,processes);

System.out.println("\n1.Add Servers\n2.Remove Servers\n3.Add Processes\n4.Exit\n");

switch(sc.nextInt()){

case 1:

System.out.print("How many more Servers:- ");

servers+=sc.nextInt();

break;

case 2:

System.out.print("How many Servers to remove:- ");

servers-=sc.nextInt();

break;

case 3:

System.out.print("How many more Processes:- ");

processes+=sc.nextInt();

break;

case 4:

return;

}

}

}

}

**OUTPUT:**

C:\Users\Dell\Desktop>javac LoadBalancer.java

C:\Users\Dell\Desktop>java LoadBalancer

Enter The number of servers and processes: 2

5

Server A has 3 processes

Server B has 2 processes

1.Add Servers

2.Remove Servers

3.Add Processes

4.Exit

1

How many more Servers:- 1

Server A has 2 processes

Server B has 2 processes

Server C has 1 processes

1.Add Servers

2.Remove Servers

3.Add Processes

4.Exit

3

How many more Processes:- 1

Server A has 2 processes

Server B has 2 processes

Server C has 2 processes

**Experiment No: 6**

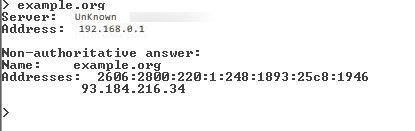
**Aim:** Name Resolution Protocol

**Theory:**

nslookup is a simple but very practical command-line tool, which is principally used to find the **IP address** that corresponds to a host, or the domain name that corresponds to an IP address (a process called “Reverse DNS Lookup”). nslookup allows itself to be used in the command-line of the operating system in question; Windows users start the service via the **command prompt**, and Unix users via the **terminal window**. Additionally, there are now a number of services that make it possible to use nslookup online.

nslookup retrieves the relevant address information directly from the DNS cache of name servers, a process which can be achieved through two different modes that the user can choose from. In the **noninteractive mode**, the tool inspects the **resource records** (which is what the address entries in the DNS cache are called) that are stored in the local name server, in a standard way. This mode is especially well suited for simple queries, for which a single domain entry needs to be looked up. When you want to use a different DNS server for the research and complete more complex search processes, you need the **interactive mode**, in which the command-line program needs to be started separately at first.

Simply enter the domain name for which you want to find out the IP address (or vice versa) into the command-line and confirm your choice by pressing the enter key. nslookup, for example, presents the following result in the search for the address to the example.org domain:



**Conclusion:**

nslookup lets itself be used in interactive mode online as well. This way, users can enter the name server of their choice instead of using the provider’s standard DNS driver, switch the **port**, or determine the **query type**.

**CODE:**

import java.io.\*;

import java.net.\*;

public class NameRes{

public static void main(String args[]) throws IOException{

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("\nEnter The Website URL :- \n");

String name = br.readLine();

try{

InetAddress ip = InetAddress.getByName(name);

System.out.println("\nIP Address:\n"+ip.getHostAddress());

}

catch(UnknownHostException e){

System.out.println("NO SUCH HOST");

}

}

}

**OUTPUT:**

E:\SEM8\DC pracs\code>javac NameRes.java

E:\SEM8\DC pracs\code>java NameRes

Enter The Website URL :-

www.facebook.com

IP Address:

31.13.79.35

E:\SEM8\DC pracs\code>java NameRes

Enter The Website URL :-

www.instagram.com

IP Address:

31.13.79.174

E:\SEM8\DC pracs\code>java NameRes

Enter The Website URL :-

www.telegram.com

IP Address:

45.60.159.62

**Experiment No: 7**

**Aim:** Implement Bully election algorithm.

**Theory:**

Distributed Algorithm is a algorithm that runs on a distributed system. 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 other machine. Many algorithms used in 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 group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted.

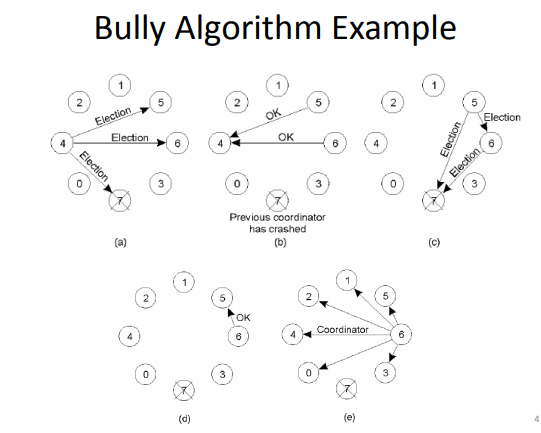
Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then this number is send to every active process in the distributed system.

**The Bully Algorithm –**  
 This algorithm applies to 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.

1. If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
2. Now process P sends election message to every process with high priority number.
3. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
4. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
5. However, if an answer is received within time T from any other process Q,

* Process P again waits for time interval T’ to receive another message from Q that it has been elected as coordinator.
* If Q doesn’t responds within time interval T’ then it is assumed to have failed and algorithm is restarted.



**Conclusion:**

Java implementation of the distributed computing algorithm, i.e bully election algorithm which aims to elect an coordinator or a leader from the nodes active in the computing environment

**OUTPUT:**

**CODE:**

import java.io.\*;

import java.util.Scanner;

class Bully{

static int n;

static int pro[] = new int[100];

static int sta[] = new int[100];

static int co;

public static void main(String args[])throws IOException

{

System.out.println("Enter the number of process");

Scanner in = new Scanner(System.in);

n = in.nextInt();

int i,j,k,l,m;

for(i=0;i<n;i++)

{

System.out.println("For process "+(i+1)+":");

System.out.print("Status: ");

sta[i]=in.nextInt();

System.out.print("Priority: ");

pro[i] = in.nextInt();

}

System.out.println("Which process will initiate election?");

int ele = in.nextInt();

elect(ele);

System.out.println("Final coordinator is "+co);

}

static void elect(int ele)

{

ele = ele-1;

co = ele+1;

for(int i=0;i<n;i++)

{

if(pro[ele]<pro[i])

{

System.out.println("Election message is sent from "+(ele+1)+" to "+(i+1));

if(sta[i]==1)

elect(i+1);

}

}

}

}

**OUTPUT:**

C:\Users\Dell\Desktop>javac Bully.java

C:\Users\Dell\Desktop>java Bully

Enter the number of process

7

For process 1:

Status: 1

Priority: 1

For process 2:

Status: 1

Priority: 2

For process 3:

Status: 1

Priority: 3

For process 4:

Status: 1

Priority: 4

For process 5:

Status: 1

Priority: 5

For process 6:

Status: 1

Priority: 6

For process 7:

Status: 0

Priority: 7

Which process will initiate election?

4

Election message is sent from 4 to 5

Election message is sent from 5 to 6

Election message is sent from 6 to 7

Election message is sent from 5 to 7

Election message is sent from 4 to 6

Election message is sent from 6 to 7

Election message is sent from 4 to 7

Final coordinator is 6

**Experiment No: 8**

**Aim**: Implementation of Clock synchronization Algorithm.

**Theory:**

Lamport’s Algorithm is a permission-based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems. In permission-based timestamp is used to order critical section requests and to resolve any conflict between requests. In Lamport’s Algorithm critical section requests are executed in the increasing order of timestamps i.e a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.

In this algorithm:

* Three type of messages (REQUEST, REPLY and RELEASE) are used and communication channels are assumed to follow FIFO order.
* A site send a REQUEST message to all other site to get their permission to enter critical section.
* A site send a REPLY message to requesting site to give its permission to enter the critical section.
* A site send a RELEASE message to all other site upon exiting the critical section.
* Every site Si, keeps a queue to store critical section requests ordered by their timestamps. request\_queuei denotes the queue of site Si
* A timestamp is given to each critical section request using Lamport’s logical clock.
* Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

**Algorithm:**

To enter Critical section:

* When a site Si wants to enter the critical section, it sends a request message Request(tsi, i) to all other sites and places the request on request\_queuei. Here, Tsi denotes the timestamp of Site Si
* When a site Sj receives the request message REQUEST(tsi, i) from site Si, it returns a timestamped REPLY message to site Si and places the request of site Si on request\_queuej

To execute the critical section:

* A site Si can enter the critical section if it has received the message with timestamp larger than (tsi, i) from all other sites and its own request is at the top of request\_queuei

To release the critical section:

* When a site Si exits the critical section, it removes its own request from the top of its request queue and sends a timestamped RELEASE message to all other sites
* When a site Sj receives the timestamped RELEASE message from site Si, it removes the request of Si from its request queue

**Message Complexity:**

Lamport’s Algorithm requires invocation of 3(N – 1) messages per critical section execution. These 3(N – 1) messages involves

* (N – 1) request messages
* (N – 1) reply messages
* (N – 1) release messages

**Drawbacks of Lamport’s Algorithm:**

* Unreliable approach: failure of any one of the processes will halt the progress of entire system.
* High message complexity: Algorithm requires 3(N-1) messages per critical section invocation.

**Performance:**

* Synchronization delay is equal to maximum message transmission time
* It requires 3(N – 1) messages per CS execution.
* Algorithm can be optimized to 2(N – 1) messages by omitting the REPLY message in some situations.

**CONCLUSION:**

Lamport algorithm is a simple algorithm used to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method.

**Code:**

import java.util.\*;

import java.util.Scanner;

import javax.swing.\*;

import java.awt.\*;

import java.awt.geom.\*;

public class lamport{

int e[][]=new int[10][10];

int en[][]=new int[10][10];

int ev[]=new int[10];

int i,p,j,k;

HashMap<Integer,Integer> hm=new HashMap<Integer,Integer>();

int xpoints[] =new int[5];

int ypoints[] =new int[5];

class draw extends JFrame{

private final int ARR\_SIZE = 4;

void drawArrow(Graphics g1, int x1, int y1, int x2, int y2) {

Graphics2D g = (Graphics2D) g1.create();

double dx = x2 - x1, dy = y2 - y1;

double angle = Math.atan2(dy, dx);

int len = (int) Math.sqrt(dx\*dx + dy\*dy);

AffineTransform at = AffineTransform.getTranslateInstance(x1, y1);

at.concatenate(AffineTransform.getRotateInstance(angle));

g.transform(at);

// Draw horizontal arrow starting in (0, 0)

g.drawLine(0, 0, len, 0);

g.fillPolygon(new int[] {len, len-ARR\_SIZE, len-ARR\_SIZE, len},

new int[] {0, -ARR\_SIZE, ARR\_SIZE, 0}, 4);

}

public void paintComponent(Graphics g) {

for (int x = 15; x < 200; x += 16)

drawArrow(g, x, x, x, 150);

drawArrow(g, 30, 300, 300, 190);

}

public void paint(Graphics g){

int h1,h11,h12;

Graphics2D go=(Graphics2D)g;

go.setPaint(Color.black);

for(i=1;i<=p;i++){

go.drawLine(50,100\*i,450,100\*i);

}

for(i=1;i<=p;i++){

for(j=1;j<=ev[i];j++){

k=i\*10+j;

go.setPaint(Color.blue);

go.fillOval(50\*j,100\*i-3,5,5);

go.drawString("e"+i+j+"("+en[i][j]+")",50\*j,100\*i-5);

h1=hm.get(k);

if(h1!=0){

h11=h1/10;

h12=h1%10;

go.setPaint(Color.red);

drawArrow(go,50\*h12+2,100\*h11,50\*j+2,100\*i);

}

}

}

}

}

public void calc(){

Scanner sc=new Scanner(System.in);

System.out.println("Enter the number of process:");

p=sc.nextInt();

System.out.println("Enter the no of events per process:");

for(i=1;i<=p;i++)

{

ev[i]=sc.nextInt();

}

System.out.println("Enter the relationship:");

for(i=1;i<=p;i++)

{

System.out.println("For process:"+i);

for(j=1;j<=ev[i];j++)

{

System.out.println("For event:"+(j));

int input=sc.nextInt();

k=i\*10+j;

hm.put(k,input);

if(j==1)

en[i][j]=1;

}

}

for(i=1;i<=p;i++){

for(j=2;j<=ev[i];j++){

k=i\*10+j;

if(hm.get(k)==0){

en[i][j]=en[i][j-1]+1;

}

else{

int a=hm.get(k);

int p1=a/10;

int e1=a%10;

if(en[p1][e1]>en[i][j-1])

en[i][j]=en[p1][e1]+1;

else

en[i][j]=en[i][j-1]+1;

}

}

}

JFrame jf=new draw();

jf.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

jf.setSize(500,500);

jf.setVisible(true);

}

public static void main(String[] args){

lamport lam=new lamport();

lam.calc();

}

}

**OUTPUT:**

C:\Users\Dell\Desktop>java lamport

Enter the number of process:

2

Enter the no of events per process:

7

5

Enter the relationship:

For process:1

For event:1

0

For event:2

0

For event:3

0

For event:4

0

For event:5

22

For event:6

0

For event:7

24

For process:2

For event:1

0

For event:2

0

For event:3

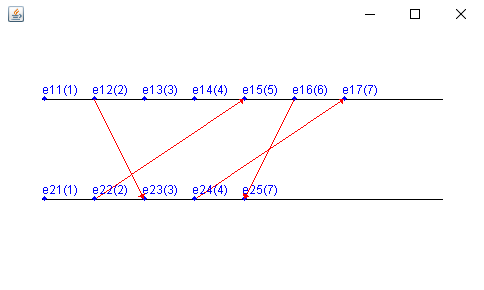
12

For event:4

0

For event:5

16



**Experiment No: 9**

**Aim**: Implementation of Mutual Exclusion Algorithm.

**Theory:**

**Mutual exclusion in distributed system**

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 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 variable (For example: Semaphores) mutual exclusion problem can be easily solved.In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.A site in distributed system do not have complete information of 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 site should not endlessly wait for any message that will never arrive.
* No Starvation: Every site who wants to execute 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 critical section. Any request to execute 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 number to order requests for the critical section.
* Each requests for 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

**CONCLUSION:**

The token-based algorithm uses the sequences to order the request for the Computer Systems and to resolve the conflict for the simultaneous requests for the System. Token-based algorithms are more scalable as they can free your server from having to store session state and also they contain all the necessary information which they need for authentication.

**OUTPUT:**

**Code:**

**EchoServer.java**

import java.io.\*;

import java.net.\*;

public class EchoServer implements Runnable{

Socket socket=null; static ServerSocket ss;

EchoServer(Socket newSocket){

this.socket=newSocket;

}

public static void main(String args[]) throws IOException{

ss=new ServerSocket(7000);

System.out.println("Server Started");

while(true){

Socket s = ss.accept();

EchoServer es = new EchoServer(s);

Thread t = new Thread(es);

t.start();

}

}

public void run(){

try {

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

while(true){

System.out.println(in.readLine());

}

}

catch(Exception e){

}

}

}

**EchoClientOne.java**

import java.io.\*;

import java.net.\*;

public class EchoClientOne{

public static void main(String args[])throws IOException{

Socket s=new Socket("localhost",7000);

PrintStream out = new PrintStream(s.getOutputStream());

ServerSocket ss = new ServerSocket(7001);

Socket s1 = ss.accept();

BufferedReader in1 = new BufferedReader(new

InputStreamReader(s1.getInputStream()));

PrintStream out1 = new PrintStream(s1.getOutputStream());

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

String str="Token";

while(true){

if(str.equalsIgnoreCase("Token")){

System.out.println("Do you want to send some data");

System.out.println("Enter Yes or No");

str=br.readLine();

if(str.equalsIgnoreCase("Yes")){

System.out.println("Enter the data");

str=br.readLine();

out.println(str);

}

out1.println("Token");

}

System.out.println("Waiting for Token");

str=in1.readLine();

}

}

}

**EchoClientTwo.java**

import java.io.\*;

import java.net.\*;

public class EchoClientTwo{

public static void main(String args[])throws IOException{

Socket s=new Socket("localhost",7000);

PrintStream out = new PrintStream(s.getOutputStream());

Socket s2=new Socket("localhost",7001);

BufferedReader in2 = new BufferedReader(new

InputStreamReader(s2.getInputStream()));

PrintStream out2 = new PrintStream(s2.getOutputStream());

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

String str;

while(true){

System.out.println("Waiting for Token");

str=in2.readLine();

if(str.equalsIgnoreCase("Token")){

System.out.println("Do you want to send some data");

System.out.println("Enter Yes or No");

str=br.readLine();

if(str.equalsIgnoreCase("Yes")){

System.out.println("Enter the data");

str=br.readLine();

out.println(str);

}

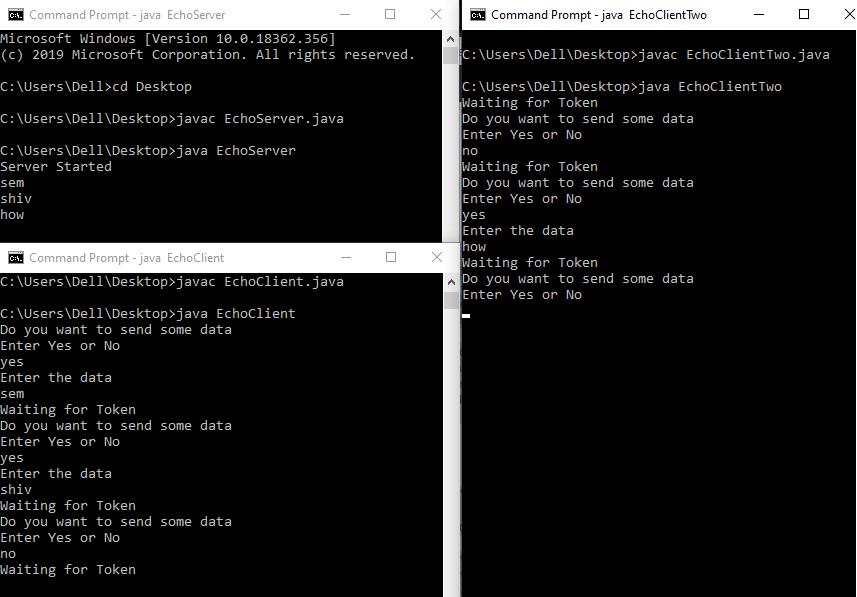
out2.println("Token");

}

}

}

}



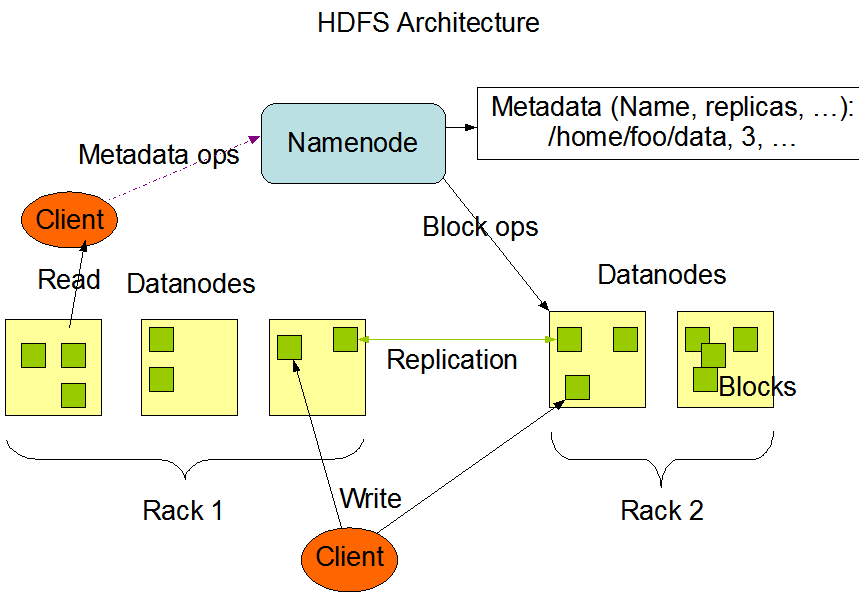
**Experiment No: 10**

**Case study of Distributed file system- Hadoop**

Hadoop is a software platform for processing and storing large data sets on cluster. Hadoop makes it possible to manage thousands of terabytes of data. The system can continue operating uninterrupted in case of a node failure. Hadoop is actually implemented in Jan, 2008. Hadoop framework is used by major companies such as Microsoft, Yahoo, IBM, Amazon. Hadoop has number of components such as HDFS, MapReduce, Hive, Pig, HBase, ZooKeeper etc. and all components perform different functions for Hadoop.

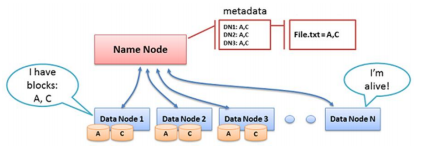
HDFS has a master/slave architecture. An HDFS cluster consists of a single Namenode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of Datanodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.

HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of Datanodes. The Namenode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to Datanodes. The Datanodes are responsible for serving read and write requests from the file system’s clients. The Datanodes also perform block creation, deletion, and replication upon instruction from the Namenode.



1. **Name Node**

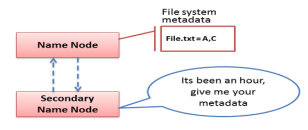
It provides the file metadata information such as name of Data Node, name of block, name of file. The Name Node is the central controller of HDFS. It does not hold any cluster data itself. The Name Node only knows what blocks make up a file and where those blocks are located in the cluster shown in figure.3.



During normal operation Data Nodes send heartbeats to the Name Node to confirm that the Data Node is operating. The default heartbeat interval is 3 seconds and each Data Node sends heartbeat after every 3 seconds. If Name Node does not receive heartbeat from Data Node in 10 min then Name Node consider that Data Node is unavailable & make decision for managing the data blocks of that Data Node. Every tenth heartbeat is a Block Report in which the Data Node tells the Name Node about all the blocks it has. The block reports allow the Name Node build its metadata. Metadata maintained by the Name Node which contains the name of files, name of blocks of each file and name of Data Nodes which contains the blocks of the file. The metadata is used for referring the Data Nodes.

1. **Secondary Name Node**

The Secondary Name Node occasionally connects to the Name Node (by default, one hour) and gets a copy of the Name Node’s metadata. The Secondary Name Node combines this information in a recent set of files. If Name Node getting failure then the files retained by the Secondary Name Node can be used to recover the Name Node. The administrator may configure the Secondary Name Node setting for connecting with Name Node. During the failing condition of the Name Node, the Secondary Name Node provides the name of all files, name of blocks of each file and name of the Data Nodes for each block of a file. The time duration for communication is provided at the configuration time.



**Operations in HDFS**

HDFS is designed for distributed data storage and for storage there are two possible operations one is read operation and another is write operation. In read operation the client want some data for their use from the cluster which is already in a cluster and after analyzing the metadata the Name Node provides the needed data to the client. In write operation, the client wants to store data on cluster. The Name Node must update the metadata information for future references and provides the Data Nodes under the control of the client.

i. Read Operation

When a Client wants to retrieve a file from HDFS it again consults the Name Node and asks for the block locations of the file. The Name Node reads metadata information and control of that Data Node is transferred to client for their purpose. The Client reads one block at a time. the client wants to read the blocks of the results file and send query to the Name Node. Then Name Node performs the analysis of the metadata information. After analyzing the metadata the Name Node provides the location of the Data Node which has the needed data block. Then client can read block from the Data Node through Name Node. The Name Node is actually used for creating the connection between client and the accurate Data Node.

ii. Write operation

Multiple copies of blocks are placed in cluster to avoid data loss and for increasing the reliability. The default no. of copies is three, two on same rack & one on different rack. During write operation the client sends query to the Name Node and then Name Node transfer the control of Data Nodes to the client through Name Node. The Name Node automatically provides the Data Nodes in a way that will be reliable and can be retrieved faster.

The Hadoop distributed file system is widely used in cloud computing. This technology is providing the fast access to the data. The user can easily work with cloud Hadoop cluster. This technology has advanced features such as reliability, efficiency and security. HDFS architecture makes the read and write operations efficient with the use of Hadoop cluster.

**Experiment No: 11**

**Case study on Common Object Request Broker Architecture (CORBA)**.

The Common Object Request Broker Architecture (CORBA) for ORB technology and Java applet technology for Web technology. We are using exemplars to make the discussion more concrete. We are using these particular exemplars because they are the most visible and popular representatives of their respective classes. While this decision may somewhat color our discussion of DOT with the peculiarities of CORBA and Java, our intent is to address the broader aspects of DOT. To provide the necessary perspective, we will give a brief explanation of the origins and current states of these exemplars. They arose from two broad classes of DOT progenitors: Operating systems and distributed systems infrastructures influenced CORBA, and programming languages and the Web gave way to Java.

As object technology became more popular through the 1980s there was more interest in bundling the concept of objects with the concept of transparent distributed computing. Objects, with their inherent combination of data and behavior and their strict separation of interface from implementation, offer an ideal package for distributing data and processes to end-user applications. Objects became an enabling technology for distributed processing. In the early1990s an international trade association called the Object Management Group (OMG) [OMG 97] defined a standard for the distribution of objects. The OMG defined the Common Object Request Broker Architecture (CORBA), which provided a standard by which OT could be used in distributed computing environments. The latest version of this standard, CORBA 2.0, addresses issues related to interface, registration, databases, communication, and error handling. When combined with other object services defined by the OMG Object Management Architecture OMA), CORBA becomes (a middleware that facilitates full exploitation of object technology in a distributed system. However, if we were to characterize CORBA technology in the simplest possible language, it would be to say it is an object-oriented RPC.

