योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

B.Tech Programme: Computer Science & Engineering

Course Title: Distributed Systems and Cloud Computing

Course Code: CIE-407P

Semester: 7th

Submitted To:
Mr. Akshit Thakur, Assistant Professor

Submitted By:

Name: Rudraksha Tyagi

Enrollment No: 04317702722

Branch & Section: CSE-A (G2)

VISION OF INSTITUTE

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

MISSION OF INSTITUTE

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building capabilities to enable them to make significant contributions to the world.

योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

INDEX

S.No	EXP.	Allotment Date	Submission Date	Marks			Remark	Updated Marks	Faculty Signature
				Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			
1	Write a Program in Java to implement RPC.								
2	Implement the concept of Remote Method Invocation in Java.								
3	Write a java program to implement Lamport's Logical clock								
4	Implement mutual exclusion service using Lamport's Mutual Exclusion Algorithm								

5	Install Hadoop on Windows								
6	Run a simple application on single node Hadoop Cluster.								
7	Install Google App Engine / AWS and develop a simple web application.								
8	Launch Web application using Google App Engine.								
9	Install VirtualBox / VMware Workstation with different flavours of Linux on Windows								
10	Simulate a cloud scenario using CloudSim and run a scheduling algorithm.								

EXPERIMENT 1

Aim:

Write a Program in Java to implement RPC.

Theory:

Remote Procedure Call (RPC) is a protocol that allows a program to execute procedures or functions on a remote server as if they were local function calls. It enables distributed computing by hiding the complexity of network communication from the programmer.

Key Concepts

1. Client-Server Model
   - Client: Makes procedure calls to remote server
   - Server: Executes procedures and returns results
   - Communication happens over network using sockets

2. RPC Components
   - Stub: Client-side proxy that handles network communication
   - Skeleton: Server-side component that receives requests
   - Marshalling: Converting data into network-transmissible format
   - Unmarshalling: Converting received data back to original format

3. RPC Process Flow
   1. Client calls a remote procedure
   2. Client stub marshalls parameters and sends request
   3. Server skeleton receives and unmarshalls request
   4. Server executes the procedure
   5. Server marshalls result and sends response
   6. Client stub receives and unmarshalls response
   7. Client gets the result

Advantages of RPC
- Transparency: Remote calls appear like local calls
- Simplicity: Hides network complexity from programmer
- Language Independence: Different languages can communicate
- Distributed Computing: Enables distributed applications

Disadvantages of RPC
- Network Dependency: Fails if network connection is lost
- Performance: Slower than local procedure calls
- Complexity: Error handling for network failures
- Security: Data transmission over network needs security

Program:

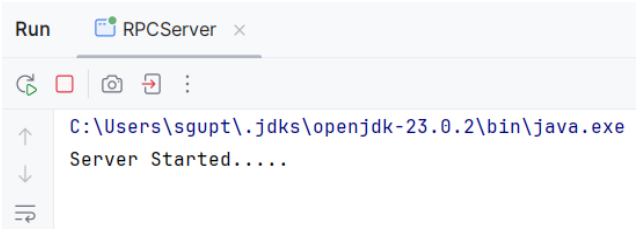
RPCServer.java

```java
import java.io.*;
import java.net.*;

public class RPCServer {
    public static void main(String[] args) throws Exception {
        ServerSocket serverSocket = new ServerSocket( port: 5000);
        System.out.println("Server Started.....");

        while(true){
            try(Socket socket = serverSocket.accept()){
                BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);

                String request = in.readLine();
                String[] parts = request.split( regex: ",");
                String method = parts[0].trim();
                int a = Integer.parseInt(parts[1].trim());
                int b = Integer.parseInt(parts[2].trim());

                int result = 0;
                if (method.equals("add")){
                    result = a + b;
                }

                out.println(result);
            }
        }
    }
}
```

RPCClient.java

```java
import java.io.*;
import java.net.*;

public class RPCClient {
    public static void main(String[] args) throws Exception {
        try (Socket socket = new Socket( host: "localhost", port: 5000)){
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            out.println("add, 10, 20");
            String response = in.readLine();
            System.out.println("Result from server: " + response);
        }
    }
}
```

Output:



Learning Outcomes:

EXPERIMENT 2

Aim:

Implement the concept of Remote Method Invocation in Java.

Theory:

Remote Method Invocation (RMI) in Java

RMI enables distributed computing by allowing objects in different JVMs to call methods on each other.

Core Components

- Stub (Client-side): Proxy for remote objects, marshalls parameters, and sends requests.
- Skeleton (Server-side): Receives requests, unmarshalls parameters, invokes methods, and sends results.
- RMI Registry: Registers server objects by name. Clients use lookup() (default port 1099).
- Remote Reference Layer (RRL): Manages client references and virtual connections.
- Transport Layer: Handles network connections and data transmission.

Process Flow

1. Client calls a remote method.
2. Stub marshalls and sends the request.
3. Skeleton unmarshalls, invokes method, and sends result.
4. Stub unmarshalls, and client receives the result.

Marshalling & Unmarshalling

- Marshalling: Converts data into a transmit-friendly format (serialization).
- Unmarshalling: Converts received data back to its original format.

Requirements

- Remote Interface: Extends java.rmi.Remote, methods throw RemoteException, and parameters/returns must be Serializable or Remote.
- Remote Implementation: Implements Remote, extends UnicastRemoteObject, and registers with RMI Registry.

Program:

RemoteInterface.java

```java
// RemoteInterface.java
import java.rmi.Remote;
import java.rmi.RemoteException;

interface Calculator extends Remote {  3 usages  1 implementation
    public int add(int x, int y) throws RemoteException;  1 usage  1 implementation
    public int subtract(int x, int y) throws RemoteException;  1 usage  1 implementation
    public int multiply(int x, int y) throws RemoteException;  1 usage  1 implementation
    public double divide(int x, int y) throws RemoteException;  1 usage  1 implementation
}
```

CalculatorImpl.java

```java
// CalculatorImpl.java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {  2 usages

    public CalculatorImpl() throws RemoteException {  1 usage
        super();
    }
    @Override  1 usage
    public int add(int x, int y) throws RemoteException {
        return x + y;
    }
    @Override  1 usage
    public int subtract(int x, int y) throws RemoteException {
        return x - y;
    }
    @Override  1 usage
    public int multiply(int x, int y) throws RemoteException {
        return x * y;
    }
    @Override  1 usage
    public double divide(int x, int y) throws RemoteException {
        if (y == 0) {
            throw new RemoteException("Division by zero not allowed");
        }
        return (double) x / y;
    }
}
```

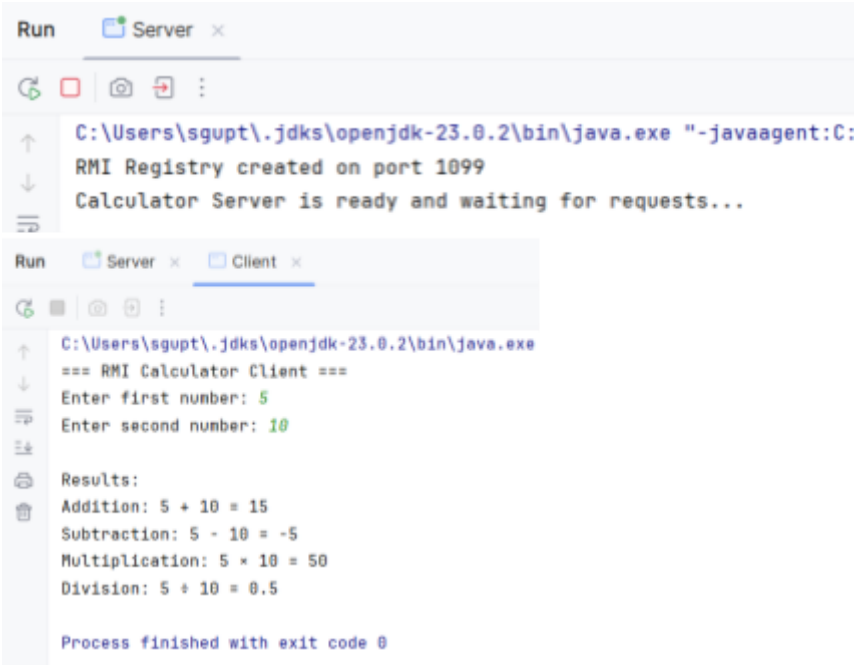
Server.java

```java
// Server.java
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;

public class Server {
    public static void main(String[] args) {
        try {
            try {
                LocateRegistry.createRegistry( port: 1099);
                System.out.println("RMI Registry created on port 1099");
            } catch (RemoteException e) {
                System.out.println("RMI Registry already exists");
            }

            CalculatorImpl calculator = new CalculatorImpl();

            Naming.rebind( name: "//localhost:1099/Calculator", calculator);

            System.out.println("Calculator Server is ready and waiting for requests...");

        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

```

Client.java

```java
// Client.java
import java.rmi.Naming;
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        try {
            Calculator calculator = (Calculator) Naming.lookup( name: "//localhost:1099/Calculator");
            Scanner scanner = new Scanner(System.in);
            System.out.println("=== RMI Calculator Client ===");
            System.out.print("Enter first number: ");
            int num1 = scanner.nextInt();

            System.out.print("Enter second number: ");
            int num2 = scanner.nextInt();

            System.out.println("\nResults:");
            System.out.println("Addition: " + num1 + " + " + num2 + " = " + calculator.add(num1, num2));
            System.out.println("Subtraction: " + num1 + " - " + num2 + " = " + calculator.subtract(num1, num2));
            System.out.println("Multiplication: " + num1 + " × " + num2 + " = " + calculator.multiply(num1, num2));

            if (num2 != 0) {
                System.out.println("Division: " + num1 + " ÷ " + num2 + " = " + calculator.divide(num1, num2));
            } else {
                System.out.println("Division: Cannot divide by zero");
            }
            scanner.close();

        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Output:



Learning Outcomes:

EXPERIMENT NO.-03

AIM-

Write a java program to implement Lamport's Logical clock

THEORY-

In a distributed system, there is no global clock, and different processes may have different local times. To maintain the correct order of events that occur across processes, Lamport introduced a logical clock mechanism.The main idea is to assign a numerical timestamp to each event in such a way that if one event happens before another, the timestamp reflects this ordering.

Rules for Lamport's logical clock:

1. Each process maintains a local logical clock. Initially, all clocks are set to zero.
2. Whenever an event occurs in a process, it increments its clock by 1.
3. When a message is sent from one process to another, the message carries the sender's timestamp.
4. When a process receives a message, it sets its clock to be greater than both its current clock value and the received timestamp, then increments it by 1.

This ensures that if event A happens before event B, then Clock(A) < Clock(B).

Lamport's clock only preserves the order of causally related events, not the exact physical time.

CODE-

```
import java.util.Scanner;
public class LamportClock {
   private int clock;

   public LamportClock() {
      clock = 0;
   }
   // internal event
   public void event() {
      clock++;
      System.out.println("Internal event occurred. Clock = " + clock);
   }
   // send message
   public int sendMessage() {
      clock++;
      System.out.println("Message sent. Clock = " + clock);
      return clock;
   }

   // receive message
```

```
    public void receiveMessage(int receivedTime) {
        clock = Math.max(clock, receivedTime) + 1;

        System.out.println("Message received. Updated Clock = " + clock);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        LamportClock process1 = new LamportClock();
        LamportClock process2 = new LamportClock();

        System.out.println("Initial clocks -> P1: " + 0 + "  P2: " + 0);

        System.out.println("\nP1 performs an internal event:");
        process1.event();

        System.out.println("\nP1 sends a message to P2:");
        int msgTime = process1.sendMessage();

        System.out.println("\nP2 receives the message:");
        process2.receiveMessage(msgTime);

        System.out.println("\nP2 performs an internal event:");
        process2.event();

        System.out.println("\nFinal Clock values:");
        System.out.println("P1 clock: " + process1.clock);
        System.out.println("P2 clock: " + process2.clock);
    }
}
```

OUTPUT-

```
Initial clocks -> P1: 0  P2: 0

P1 performs an internal event:
Internal event occurred. Clock = 1

P1 sends a message to P2:
Message sent. Clock = 2

P2 receives the message:
Message received. Updated Clock = 3

P2 performs an internal event:
Internal event occurred. Clock = 4

Final Clock values:
P1 clock: 2
P2 clock: 4

Process finished with exit code 0
```

LEARNING OUTCOME-

EXPERIMENT NO.-04

AIM-

Implement mutual exclusion service using Lamport's Mutual Exclusion Algorithm

THEORY-

In a distributed system, multiple processes may need to access a shared resource such as a file or printer. Mutual exclusion ensures that only one process can use the shared resource at a time.

Lamport's Mutual Exclusion Algorithm is a distributed approach based on message passing and Lamport's logical clocks.
It guarantees that all processes agree on the order of requests for the critical section without using a centralized coordinator.

The algorithm works as follows:

1. Each process maintains a logical clock.
2. When a process wants to enter its critical section, it sends a request message containing its process ID and current clock value to all other processes.
3. Each process that receives the request:
    - Places the request in its local request queue.
    - Sends an acknowledgment message back to the requesting process.
4. A process can enter its critical section only when:
    - It has received an acknowledgment from every other process, and
    - Its request is the smallest (earliest timestamp) in its queue.
5. After leaving the critical section, the process sends a release message to all others, and they remove that request from their queues.

This algorithm ensures mutual exclusion, fairness, and absence of deadlock.

CODE-

```
import java.util.*;

class LamportProcess implements Comparable<LamportProcess> {
   int processId;
   int timestamp;

   LamportProcess(int processId, int timestamp) {
      this.processId = processId;
      this.timestamp = timestamp;
   }

   @Override
   public int compareTo(LamportProcess other) {
```

```
        if (this.timestamp == other.timestamp)
            return this.processId - other.processId;
        return this.timestamp - other.timestamp;
    }
}

public class LamportMutualExclusion {
    private int clock;
    private int processId;
    private PriorityQueue<LamportProcess> requestQueue;

    public LamportMutualExclusion(int processId) {
        this.clock = 0;
        this.processId = processId;
        this.requestQueue = new PriorityQueue<>();
    }

    // simulate sending a request
    public void requestCS() {
        clock++;
        LamportProcess request = new LamportProcess(processId, clock);
        requestQueue.add(request);
        System.out.println("Process " + processId + " sends request with timestamp " + clock);
    }

    // simulate receiving request from another process
    public void receiveRequest(int senderId, int senderTime) {
        clock = Math.max(clock, senderTime) + 1;
        requestQueue.add(new LamportProcess(senderId, senderTime));
        System.out.println("Process " + processId + " received request from P" + senderId + " (time=" +
senderTime + ")");
    }

    // check if this process can enter the critical section
    public boolean canEnterCS(int totalProcesses) {
        LamportProcess first = requestQueue.peek();
        return (first != null && first.processId == processId && requestQueue.size() == totalProcesses);
    }

    // release critical section
    public void releaseCS() {
        requestQueue.poll();
        clock++;
        System.out.println("Process " + processId + " releases critical section (clock=" + clock + “)");

    }
    public static void main(String[] args) {
```

```
        int totalProcesses = 3;

        LamportMutualExclusion p1 = new LamportMutualExclusion(1);
        LamportMutualExclusion p2 = new LamportMutualExclusion(2);
        LamportMutualExclusion p3 = new LamportMutualExclusion(3);

        System.out.println("=== Lamport Mutual Exclusion Simulation ===\n");

        // P1 requests critical section
        p1.requestCS();
        p2.receiveRequest(1, p1.clock);
        p3.receiveRequest(1, p1.clock);

        // P2 requests after some time
        p2.requestCS();
        p1.receiveRequest(2, p2.clock);
        p3.receiveRequest(2, p2.clock);

        // P3 requests after some time
        p3.requestCS();
        p1.receiveRequest(3, p3.clock);
        p2.receiveRequest(3, p3.clock);

        System.out.println("\nChecking who can enter critical section...\n");

        if (p1.canEnterCS(totalProcesses)) {
            System.out.println("Process 1 enters Critical Section");
            p1.releaseCS();
        }
        if (p2.canEnterCS(totalProcesses)) {
            System.out.println("Process 2 enters Critical Section");
            p2.releaseCS();
        }
        if (p3.canEnterCS(totalProcesses)) {
            System.out.println("Process 3 enters Critical Section");
            p3.releaseCS();
        }
    }
}
```

OUTPUT-

```
=== Lamport Mutual Exclusion Simulation ===

Process 1 sends request with timestamp 1
Process 2 received request from P1 (time=1)
Process 3 received request from P1 (time=1)
Process 2 sends request with timestamp 3
Process 1 received request from P2 (time=3)
Process 3 received request from P2 (time=3)
Process 3 sends request with timestamp 5
Process 1 received request from P3 (time=5)
Process 2 received request from P3 (time=5)

Checking who can enter critical section...

Process 1 enters Critical Section
Process 1 releases critical section (clock=7)

Process finished with exit code 0
```

LEARNING OUTCOME-

EXPERIMENT NO.-05

AIM-

Install Hadoop on Windows

THEORY-

Hadoop is an open-source framework developed by Apache that allows for distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage.

Hadoop consists mainly of two components:

1. Hadoop Distributed File System (HDFS): Provides high-throughput access to application data.
2. MapReduce: A programming model for processing large data sets in parallel.

To install Hadoop on Windows, we set up a single-node pseudo-distributed cluster. This means Hadoop runs on a single machine but simulates distributed behaviour.

Steps to Install Hadoop on Windows:-

1. Install Java

• Download and install JDK (Java 8 or Java 11).
• Set environment variables:
JAVA_HOME = C:\Program Files\Java\jdk1.8.0_xx
PATH = %JAVA_HOME%\bin

2. Download Hadoop

• Visit https://hadoop.apache.org/releases.html
• Download the latest stable Hadoop binary.
• Extract the folder to:
C:\hadoop

3. Set Hadoop Environment Variables

Open System Properties → Environment Variables → Add:
HADOOP_HOME = C:\hadoop
PATH = %HADOOP_HOME%\bin
Then edit the file:
C:\hadoop\etc\hadoop\hadoop-env.cmd
Add this line:
set JAVA_HOME=C:\Program Files\Java\jdk1.8.0_xx

4. Configure Hadoop XML Files

Open files inside `C:\hadoop\etc\hadoop` and make the following changes:

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

```
<configuration>
   <property>
      <name>dfs.replication</name>
      <value>1</value>
   </property>
   <property>
      <name>dfs.namenode.name.dir</name>
      <value>file:/C:/hadoop/data/namenode</value>
   </property>
   <property>
      <name>dfs.datanode.data.dir</name>
      <value>file:/C:/hadoop/data/datanode</value>
   </property>
</configuration>

<configuration>
   <property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
   </property>
</configuration>

<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
   </property>
</configuration>
```

5. Format the Hadoop NameNode

Open Command Prompt and run:

```
hdfs namenode –format
```

6. Start Hadoop Services

Run the following commands one by one:

```
start-dfs.cmd
start-yarn.cmd
```

EXPERIMENT NO.-06

AIM-

Run a simple application on single node Hadoop Cluster.

THEORY-

Hadoop is an open source framework used to store and process large datasets using a distributed computing model. A single node Hadoop cluster means that all the Hadoop daemons such as NameNode, DataNode, ResourceManager, and NodeManager run on a single machine. This is useful for learning, testing, and development.

To run a simple application on a single node Hadoop cluster, we first need to set up Hadoop and configure it for single node operation. Then we can write a simple MapReduce program in Java. MapReduce is a programming model used for processing large datasets in parallel. It consists of two main functions: Mapper and Reducer. Mapper processes input data and produces key-value pairs, while Reducer aggregates the values corresponding to each key.

CODE-

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import java.util.StringTokenizer;

public class WordCount {

   public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable> {

      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(Object key, Text value, Context context
      ) throws IOException, InterruptedException {
         StringTokenizer itr = new StringTokenizer(value.toString());
         while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
         }
```

```
    }
  }
  public static class IntSumReducer
       extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
    ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

OUTPUT-

Input.txt:
hello world hello hadoop

output:
hadoop 1
hello 2
world 1

LEARNING OUTCOME-

EXPERIMENT NO.-07

AIM-

Install Google App Engine / AWS and develop a simple web application.

THEORY-

Google App Engine (GAE) and AWS (Amazon Web Services) provide cloud platforms to deploy and run applications without managing physical servers. They allow scalable, reliable, and managed hosting for web applications.

A simple web application can be built using Python (Flask) or Java (Spring Boot) and deployed on GAE or AWS. The workflow is:

1. Develop the application locally using your preferred framework.
2. Set up the cloud environment (GAE or AWS Elastic Beanstalk / AWS Lambda).
3. Deploy the application to the cloud using the provided CLI tools.
4. Access the application via a public URL.

Google App Engine supports standard and flexible environments. AWS Elastic Beanstalk automates deployment, scaling, and monitoring.

CODE-

```
mkdir MyWebApp
cd MyWebApp

# Create main.py
echo "from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return \"Hello World! Welcome to my first Google App Engine web app.\"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080, debug=True)" > main.py

# Create requirements.txt
echo "Flask==2.3.2" > requirements.txt

# Create app.yaml
echo "runtime: python39

handlers:
  - url: /.*
    script: auto" > app.yaml
```

echo "Project MyWebApp created successfully!”

To deploy on Google App Engine:

gcloud auth login
gcloud config set project PROJECT_ID
gcloud app deploy
gcloud app browse

OUTPUT-

Hello World! Welcome to my first Google App Engine web app.

LEARNING OUTCOME-

EXPERIMENT NO.-08

AIM-

Launch Web application using Google App Engine.

THEORY-

Google App Engine (GAE) is a Platform-as-a-Service (PaaS) that enables developers to build, deploy, and scale web applications without managing underlying infrastructure. GAE automatically handles load balancing, application scaling, and monitoring.

After creating and deploying the application on GAE, you can launch it using the gcloud command-line tool. The deployed app becomes publicly available via a URL such as https://<project-id>.appspot.com.

The steps include:

1. Develop a simple web app using Flask (Python) or another framework.
2. Configure the app.yaml file to define the runtime environment.
3. Deploy the app using gcloud app deploy.
4. Launch the app in a browser using gcloud app browse.

CODE-

```
// Deploy to App Engine
gcloud auth login
gcloud config set project PROJECT ID
gcloud app deploy
// Launch the web app
gcloud app browse
```

OUTPUT-

Hello World! Welcome to my first Google App Engine web app.
(Application launched successfully on: https://<project-id>.appspot.com)

LEARNING OUTCOME-

EXPERIMENT NO.-09

AIM-

Install VirtualBox / VMware Workstation with different flavours of Linux on Windows.

THEORY-

Virtualization allows running multiple operating systems on a single physical machine by creating virtual machines (VMs).
Tools like Oracle VirtualBox and VMware Workstation enable users to test, develop, and deploy software across different operating systems without dual booting.
Different Linux distributions (flavours) such as Ubuntu, Fedora, CentOS, and Kali Linux can be installed inside virtual machines. This setup is widely used for learning, testing, and development environments.

STEPS

1. Download and install Oracle VirtualBox or VMware Workstation on Windows.
2. Download a Linux ISO image (Ubuntu 22.04 LTS).
3. Create a new Virtual Machine.
4. Allocate resources (RAM, CPU, Disk Space).
5. Mount the Linux ISO file and start the VM.
6. Follow on-screen instructions to install Linux.

LEARNING OUTCOME-

EXPERIMENT NO.-10

AIM-

Simulate a cloud scenario using CloudSim and run a scheduling algorithm.

THEORY-

CloudSim is a Java-based simulation toolkit for modeling and simulating cloud computing environments and evaluating resource provisioning and scheduling algorithms.
It allows researchers and students to test cloud performance without using real cloud infrastructure.
Scheduling algorithms in cloud computing determine how virtual machines (VMs) and tasks (cloudlets) are allocated to hosts for execution — examples include Time Shared, Space Shared, Round Robin, and Priority Scheduling.

CODE-

```
import org.cloudbus.cloudsim.*;
import org.cloudbus.cloudsim.core.CloudSim;

import java.util.*;

public class SimpleCloudSimExample {
public static void main(String[] args) {
int numUsers = 1;
Calendar calendar = Calendar.getInstance();
boolean traceFlag = false;

CloudSim.init(numUsers, calendar, traceF lag);
Datacenter datacenter = createDatacenter("Datacenter_1");

DatacenterBroker broker = new DatacenterBroker("Broker");

int brokerId = broker. getIdQ);

// Create VM

Vm vm = new Vm(0, brokerId, 1000, 1, 1024, 10000, 1000, "Xen", new
CloudletSchedulerTimeShared());

broker.submitVmList(List.of(vm));

// Create Cloudlet

Cloudlet cloudlet = new Cloudlet(0, 40000, 1, 300, 300, new UtilizationModelFull(),
```

```
new UtilizationModelFull(), new UtilizationModelFullQ);

cloudlet.setUserId(brokerId);
cloudlet.setVmId(0);

broker.submitCloudletList(List.of(cloudlet));

CloudSim.startSimulation();

CloudSim.stopSimulation();

System.out.println("Cloud simulation completed successfully!");

private static Datacenter createDatacenter(String name) {
List<Host> hostList = new ArrayList<>();
List<Pe> peList = List.of(new Pe(0, new PeProvisionerSimple(1000)));

hostList.add(new Host(0, new RamProvisionerSimple(2048), new
BwProvisionerSimple(10000), 1000000, peList, new VmSchedulerTimeShared(peList)));

return new Datacenter(name, new DatacenterCharacteristics("x86", "Linux", "Xen",
hostList, 10.0, 3.0, 0.05, 0.1, 0.1), new VmAllocationPolicySimple(hostList), new
LinkedList<Storage>(), 0);
}
}
```

OUTPUT-

Cloud simulation completed successfully!

Cloudlet executed using Time Shared scheduling.

LEARNING OUTCOME-