Fr. Conceicao Rodrigues College of Engineering

Department of Computer Engineering

Academic Year 2022-23

Distributed Computing Lab (B.E. Computer Engineering)

Experiment List

Sr.No.	Title of Experiment
1	To implement Client Server Application.
2	To implement Remote Procedure Call.
3	To implement Remote Method Invocation.
4	To implement Message Queueing System.
5	To implement Group Communication.
6	To implement Lamport Algorithm for Logical Clock Synchronization.
7	To implement techniques for Election Algorithms.
8	To implement Mutual Exclusion / Deadlock Detection.
9	To implement Stateful and Stateless File Server.
10	To study HDFS and MapReduce.

LAB 1

Aim: To implement client server application

Lab Outcome:
Develop test and debug using Message-Oriented Communication or RPC/RMIbased client-server programs

Theory:

Client-Server Model

The client-server model is a decentralized computing architecture that involves dividing tasks or workloads between servers and clients. The servers provide centralized resources and services, while clients request and utilize these resources. This model enables clients to access shared data and services, while servers can manage and control access to resources. It helps to distribute the workload and reduce the burden on any single device or component. The client-server model is used in a variety of applications, including web services, email, and database systems, and is particularly useful for supporting large numbers of users or clients. This architecture is also flexible and scalable, making it a popular choice for many organizations.

Socket

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.
They are the real backbones behind web browsing. In simpler terms, there is a server and a client. Socket programming is started by importing the socket library and making a simple socket.

Function Call	Description
Socket()	To create a socket
Bind()	It's a socket identification like a telephone number to contact
Listen()	Ready to receive a connection
Connect()	Ready to act as a sender
Accept()	Confirmation, it is like accepting to receive a call from a sender
Write()	To send data
Read()	To receive data
Close()	To close a connection

Single-threading

Single-threading is a computer programming model that processes and executes tasks sequentially in a single thread of execution. In this model, the processor runs one task at a time and is unable to execute multiple tasks simultaneously. As a result, single-threaded applications can only perform one operation at a time and cannot take advantage of multi-core processors. Single-threading is simple and easy to implement, but it can be limited in terms of performance, particularly for complex or demanding tasks. However, it is still commonly used in applications where the task execution time is relatively short and predictable, or where the need for multi-threading is low. Single-threaded programming can also be used to ensure that tasks are executed in a specific order, which can be important in certain applications, such as financial transactions.

Multi-threading

Multithreading is the ability of a program or an operating system process to manage its useby more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer. Each user request for a program or system service(and here a user can also be another program) is kept track of as a thread with a separate identity. As programs work on behalf of the initial request for that thread and are interrupted by other requests, the status of work on behalf ofthat thread is kept track of until the work is completed.

Developing Client-Server Application

In this experiment we aim at developing Client–Server application using multithreading concepts. In this application server handles the requests from multiple clients.

For this experiment socket programing is used. In client program the parameters like port number, server name are taken in order to connect with server. In the server program it acceptsthe connection from client and assigns them to new connection handler object. Then one thread at server side handles new connections and another thread is for clients.

Multithread client server
example:Web client:

- Web browser scans an incoming HTML page, and finds that more files need to befetched
- Each file is fetched by a separate thread, each doing a (blocking) HTTP request
- As files come in, the browser displays them

A multithreaded server organized in a dispatcher/worker model.

Steps to run the application

1. Start server program. It will be ready to accept connection from the client.
2. On another terminal start client program and send some message to server.
3. Server will display the output.

(Code & Output)

1 client 1 server

Client.py

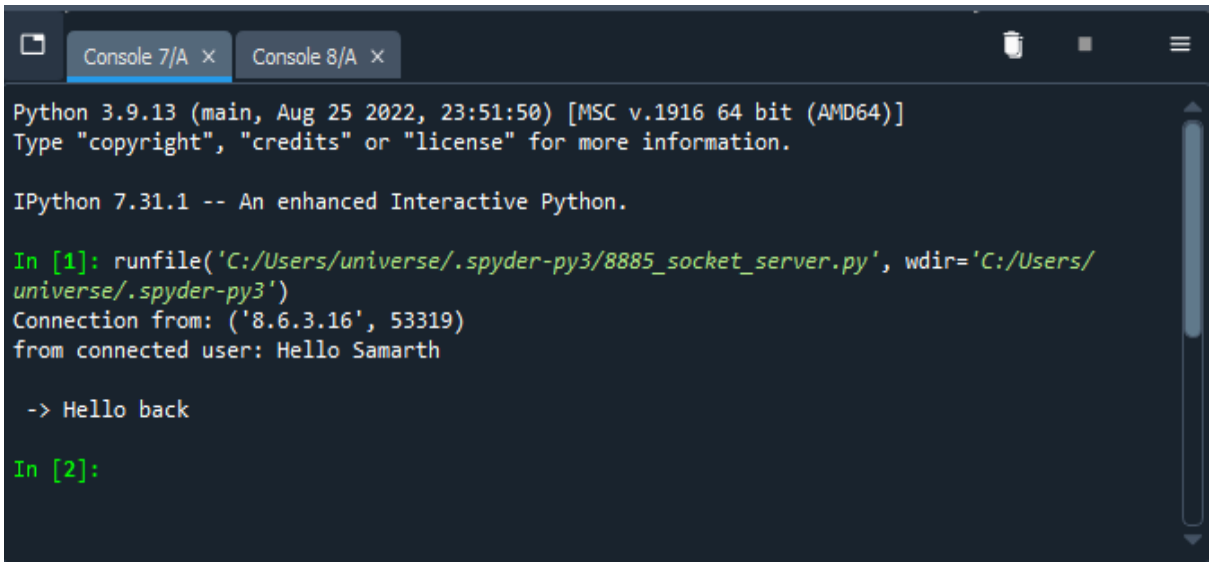
```
universe@lenovo12:~/Desktop/afdz$ cat client.py
import socket


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()
universe@lenovo12:~/Desktop/afdz$
```

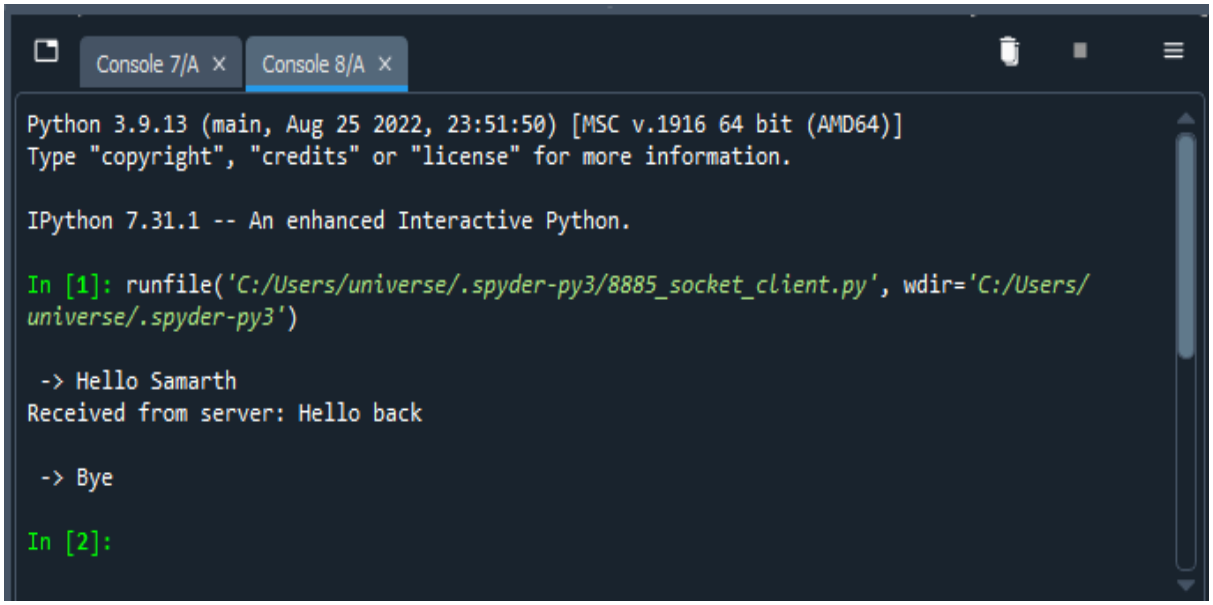
Server.py

```
universe@lenovo12:~/Desktop/afdz$ cat 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()
```

OUTPUT:





Multithreading
2 client 1 server

Client1.py

```
import socket

host = socket.gethostname()
```

```
port = 2004
BUFFER_SIZE = 2000
MESSAGE = input("tcpClientA: Enter message/ Enter exit:")

tcpClientA = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcpClientA.connect((host, port))

while MESSAGE != 'exit':
   tcpClientA.send(MESSAGE.encode())
   data = tcpClientA.recv(BUFFER_SIZE).decode()
   print (" Client2 received data:", data)
   MESSAGE = input("tcpClientA: Enter message to continue/ Enter exit:")

tcpClientA.close()
```

Client2.py

```
import socket
host = socket.gethostname()
port = 2004
BUFFER_SIZE = 2000
MESSAGE = input("tcpClientB: Enter message/ Enter exit:")

tcpClientB = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcpClientB.connect((host, port))

while MESSAGE != 'exit':
   tcpClientB.send(MESSAGE.encode())
   data = tcpClientB.recv(BUFFER_SIZE).decode()
   print (" Client received data:", data)
   MESSAGE = input("tcpClientB: Enter message to continue/ Enter exit:")

tcpClientB.close()
```

Server.py

```
import socket
from threading import Thread
from socketserver import ThreadingMixIn
# Multithreaded Python server : TCP Server Socket Thread Pool
class ClientThread(Thread):

   def __init__(self,ip,port):
      Thread.__init__(self)
      self.ip = ip
      self.port = port
      print ("[+] New server socket thread started for " + ip + ":" + str(port))
   def run(self):
      while True :
         data = conn.recv(2048).decode()
         print ("Server received data:", data)
         MESSAGE = input("Multithreaded Python server : Enter Response from Server/Enter exit:")
         if MESSAGE == 'exit':
            break
         conn.send(MESSAGE.encode())  # echo

# Multithreaded Python server : TCP Server Socket Program Stub
TCP_IP = '0.0.0.0'
TCP_PORT = 2004
BUFFER_SIZE = 20  # Usually 1024, but we need quick response

tcpServer = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcpServer.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
tcpServer.bind((TCP_IP, TCP_PORT))
threads = []

while True:
    tcpServer.listen(4)
    print ("Multithreaded Python server : Waiting for connections from TCP clients…")
    (conn, (ip,port)) = tcpServer.accept()
    newthread = ClientThread(ip,port)
    newthread.start()
    threads.append(newthread)

for t in threads:
    t.join()
```

Ouput:

```
In [4]: runfile('C:/Users/universe/.spyder-py3/temp.py', wdir='C:/Users/universe/.spyder-
py3')
Multithreaded Python server : Waiting for connections from TCP clients...
[+] New server socket thread started for 8.6.3.17:55350
Multithreaded Python server : Waiting for connections from TCP clients...
Server received data: hello samarth

Multithreaded Python server : Enter Response from Server/Enter exit:[+] New server socket
thread started for 8.6.3.17:55351
Multithreaded Python server : Waiting for connections from TCP clients...
Server received data: hello sam from client 2

Bad address (C:\ci\zeromq_1616055400030\work\src\fq.cpp:87)

Multithreaded Python server : Enter Response from Server/Enter exit:hello to server
```

```
IPython 7.31.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/universe/.spyder-py3/client1.py', wdir='C:/Users/universe/.spyder-
py3')

tcpClientA: Enter message/ Enter exit:hello samarth
 Client2 received data:

tcpClientA: Enter message to continue/ Enter exit:
```

```
In [1]: runfile('C:/Users/universe/.spyder-py3/client2.py', wdir='C:/Users/universe/.spyder-
py3')

tcpClientB: Enter message/ Enter exit:hello sam from client 2
 Client received data: hello to server

tcpClientB: Enter message to continue/ Enter exit:sup
Traceback (most recent call last):

  File "C:\Users\universe\.spyder-py3\client2.py", line 19, in <module>
    tcpClientB.send(MESSAGE.encode())
```

Conclusions :

The experiment demonstrated the difference between single threading and multithreading in a client-server model. Single threading showed limitations in handling multiple clients simultaneously, whereas multithreading improved the performance by handling multiple clients at the same time. This highlights the importance of multithreading in server design for

improved efficiency and scalability.

Post lab Questions:

1. Enlist the socket primitives.
2. What are the advantages of a Multithreaded Server?
3. With an example, explain the concept of multithreaded clients.
4. What are the motivations for using threads.
5.What are the typical models for Organizing threads.

LAB 2

Aim: To implement Remote Procedure Call

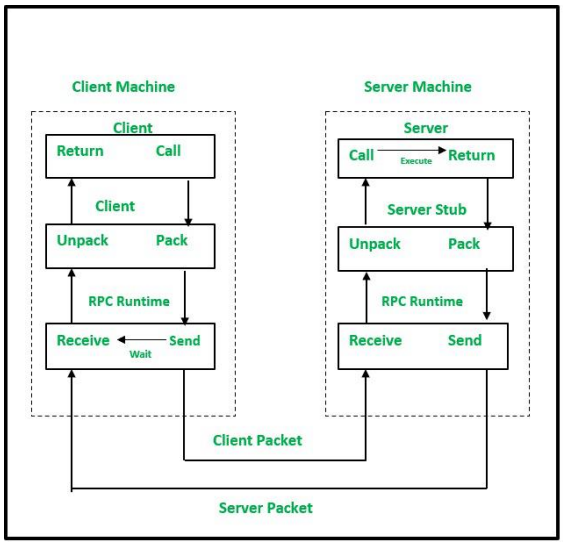
Lab Outcome:
Develop test and debug using Message-Oriented Communication or RPC/RMI based client-server programs

Theory:

RPC is an effective mechanism for building client-server systems that are distributed. RPC enhances the power and ease of programming of the client/server computing concept. It is a protocol that allows one software to seek a service from another program on another computer in a network without having to know about the network. The software that makes the request is called a client, and the program that provides the service is called a server.

There are 5 elements used in the working of RPC:

- Client
- Client Stub
- RPC Runtime
- Server Stub
- Server

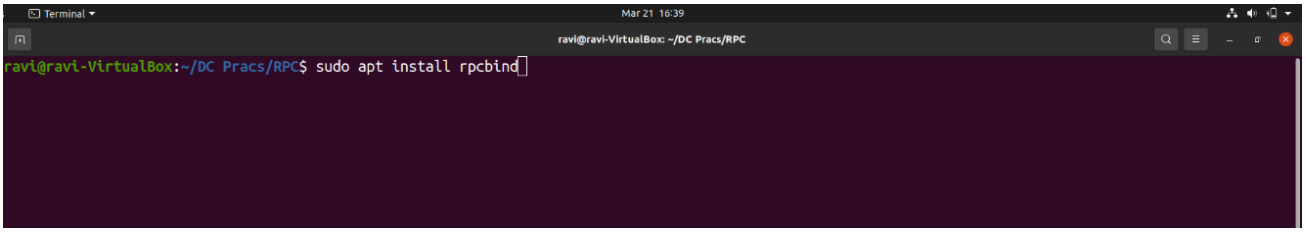


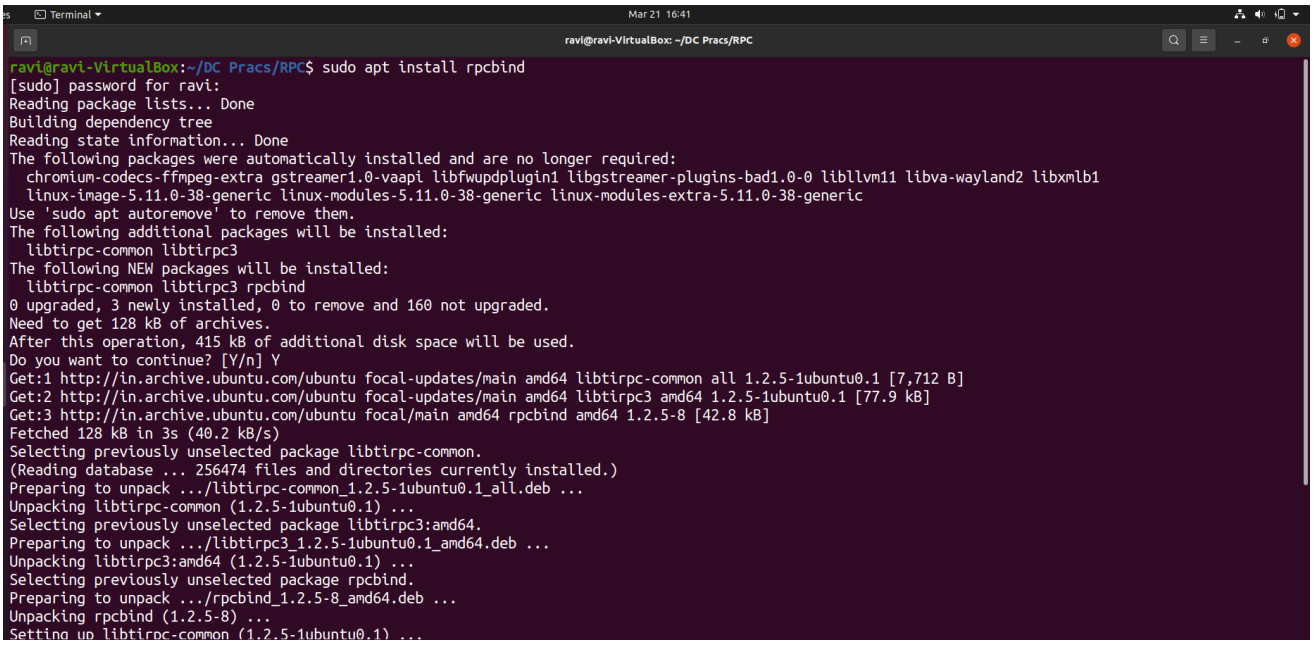
- The client, the client stub, and one instance of RPC Runtime are all running on the client machine.

- A client initiates a client stub process by giving parameters as normal. The client stub acquires storage in the address space of the client.
- At this point, the user can access RPC by using a normal Local Procedural Call. The RPC runtime oversees message transmission between client and server via the network. Retransmission, acknowledgment, routing, and encryption are all tasks performed by it.
- On the server-side, values are returned to the server stub, after the completion of server operation, which then packs (which is also known as marshalling) the return values into a message. The transport layer receives a message from the server stub.
- The resulting message is transmitted by the transport layer to the client transport layer, which then sends a message back to the client stub.
- The client stub unpacks (which is also known as unmarshalling) the return arguments in the resulting packet, and the execution process returns to the caller at this point.

Code & Output:

Installing necessary libraries: rpcbind





```
ravi@ravi-VirtualBox:~/DC Pracs/RPC$ rpcinfo
   program version netid     address                service    owner
    100000    4    tcp6      ::.0.111               portmapper superuser
    100000    3    tcp6      ::.0.111               portmapper superuser
    100000    4    udp6      ::.0.111               portmapper superuser
    100000    3    udp6      ::.0.111               portmapper superuser
    100000    4    tcp       0.0.0.0.0.111          portmapper superuser
    100000    3    tcp       0.0.0.0.0.111          portmapper superuser
    100000    2    tcp       0.0.0.0.0.111          portmapper superuser
    100000    4    udp       0.0.0.0.0.111          portmapper superuser
    100000    3    udp       0.0.0.0.0.111          portmapper superuser
    100000    2    udp       0.0.0.0.0.111          portmapper superuser
    100000    4    local     /run/rpcbind.sock      portmapper superuser
    100000    3    local     /run/rpcbind.sock      portmapper superuser
ravi@ravi-VirtualBox:~/DC Pracs/RPC$
```

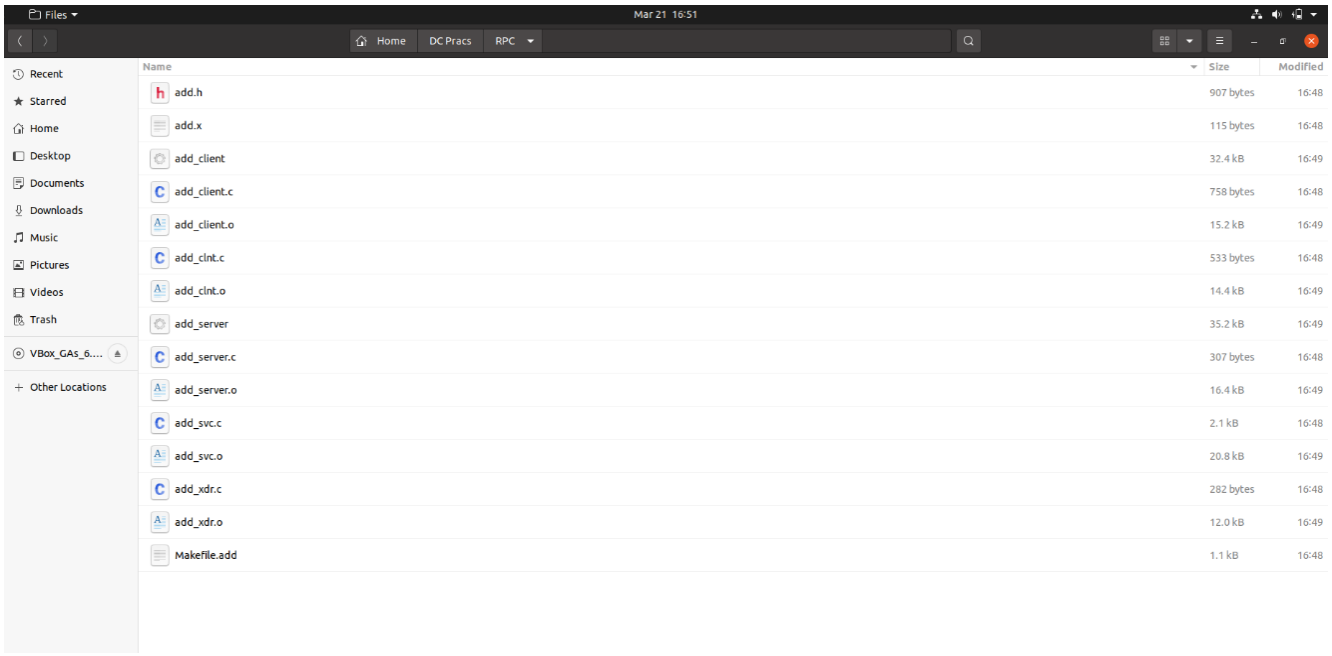
Here we specify the structure of our program using XDR

```
1 struct numbers{
2         int a;
3         int b;
4 };
5
6 program ADD_PROG{
7         version ADD_VERS{
8                 int add(numbers)=1;
9         }=1;
10 }=0x23451111;
```

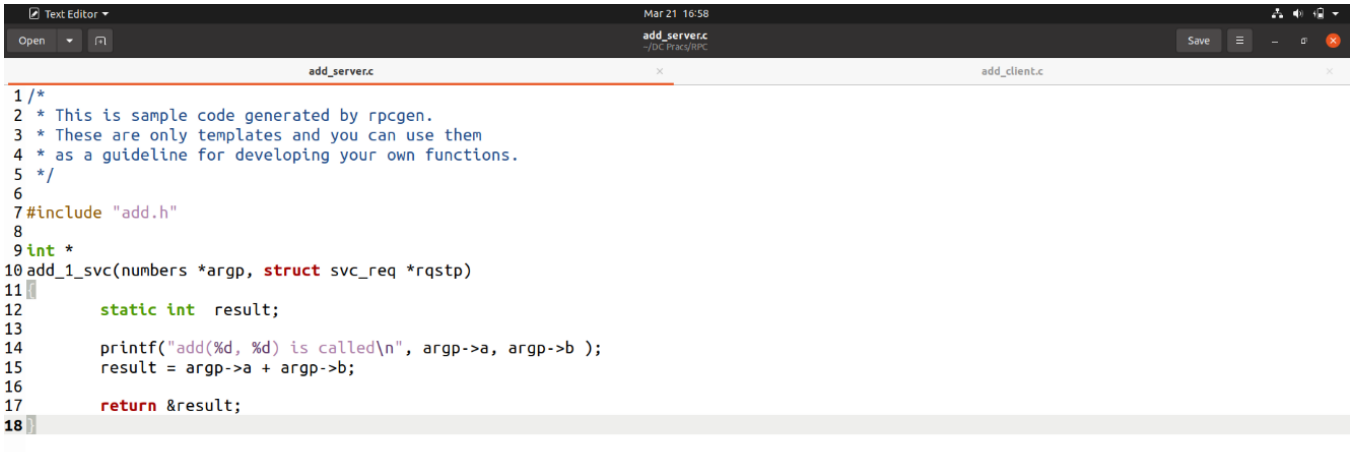
Creating other necessary files using rpcgen command

```
ravi@ravi-VirtualBox:~/DC Pracs/RPC$ rpcgen -a -C add.x
ravi@ravi-VirtualBox:~/DC Pracs/RPC$ make -f Makefile.add
cc -g    -c -o add_clnt.o add_clnt.c
cc -g    -c -o add_client.o add_client.c
cc -g    -c -o add_xdr.o add_xdr.c
cc -g     -o add_client  add_clnt.o add_client.o add_xdr.o -lnsl
cc -g    -c -o add_svc.o add_svc.c
cc -g    -c -o add_server.o add_server.c
cc -g     -o add_server  add_svc.o add_server.o add_xdr.o -lnsl
ravi@ravi-VirtualBox:~/DC Pracs/RPC$
```

Files created in the folder



Server Program File



Client Program File

Text Editor — Mar 21 17:04 — add_client.c ~/DC Pracs/RPC — Open — Save

add_server.c | add_client.c

```c
#include "add.h"


void
add_prog_1(char *host, int x, int y)
{
        CLIENT *clnt;
        int  *result_1;
        numbers  add_1_arg;

#ifndef         DEBUG
        clnt = clnt_create (host, ADD_PROG, ADD_VERS, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif  /* DEBUG */
        add_1_arg.a = x;
        add_1_arg.b = y;
        result_1 = add_1(&add_1_arg, clnt);
        if (result_1 == (int *) NULL) {
                clnt_perror (clnt, "call failed");
        }
        else{
                printf("Result: %d\n", *result_1);
        }
#ifndef         DEBUG
        clnt_destroy (clnt);
#endif   /* DEBUG */
}
```

```c

int
main (int argc, char *argv[])
{
        char *host;

        if (argc < 4) {
                printf ("usage: %s server_host NUMBER NUMBER\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        add_prog_1 (host, atoi(argv[2]), atoi(argv[3]));
exit (0);
}
```

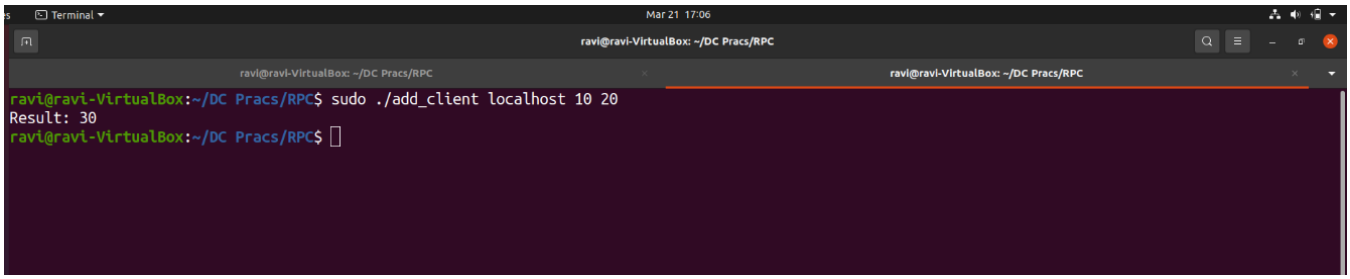
C — Tab Width: 8 — Ln 31, Col 51 — INS

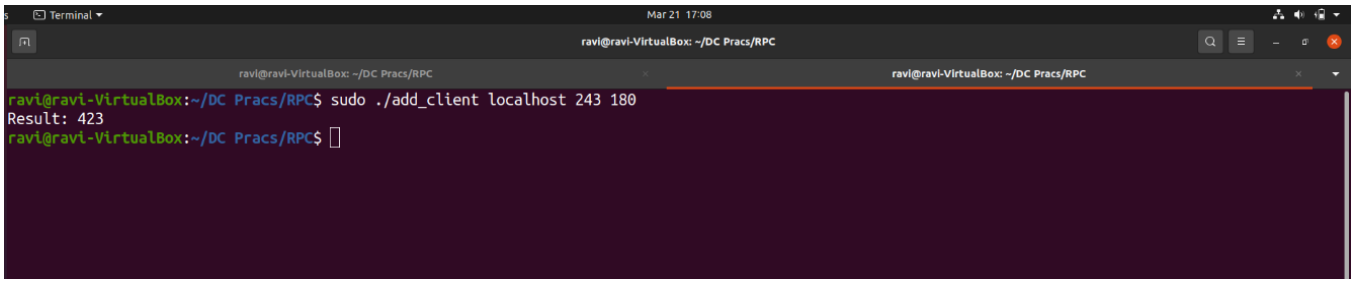
Terminal — Mar 21 17:05 — ravi@ravi-VirtualBox: ~/DC Pracs/RPC

ravi@ravi-VirtualBox: ~/DC Pracs/RPC | ravi@ravi-VirtualBox: ~/DC Pracs/RPC

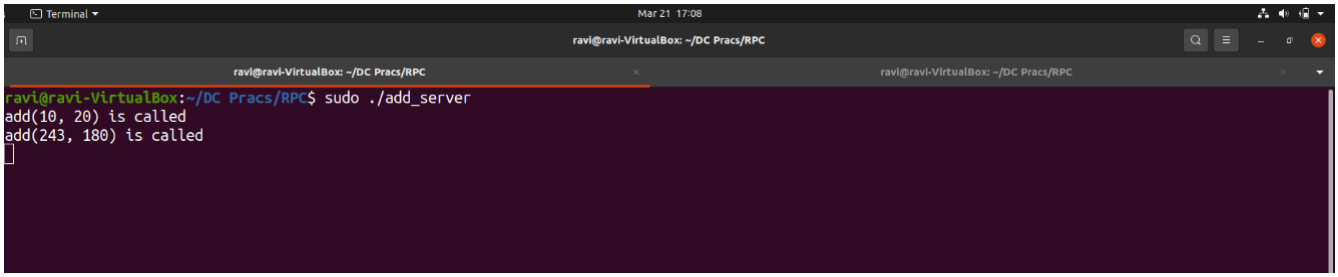
```
ravi@ravi-VirtualBox:~/DC Pracs/RPC$ make -f Makefile.add
cc -g    -c -o add_clnt.o add_clnt.c
cc -g    -c -o add_client.o add_client.c
cc -g    -c -o add_xdr.o add_xdr.c
cc -g     -o add_client  add_clnt.o add_client.o add_xdr.o -lnsl
cc -g    -c -o add_svc.o add_svc.c
cc -g    -c -o add_server.o add_server.c
cc -g     -o add_server  add_svc.o add_server.o add_xdr.o -lnsl
ravi@ravi-VirtualBox:~/DC Pracs/RPC$
```

Testing Client:





Response at the Server



Conclusions:

In conclusion, the remote procedure call (RPC) is a powerful technology that enables communication between processes running on different machines in a networked environment. The experiment performed on RPC in C language has demonstrated its ability to enable distributed computing across different machines.

Postlab Questions:

1. In which category of communication, RPC be included?
2. What are stubs? What are the different ways of stub generation?
3. What is binding?
4. Name the transparencies achieved through stubs

LAB 3

Aim: To implement Remote Method Invocation

Lab Outcome:
Develop test and debug using Message-Oriented Communication or RPC/RMI based client-server programs

Theory:

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

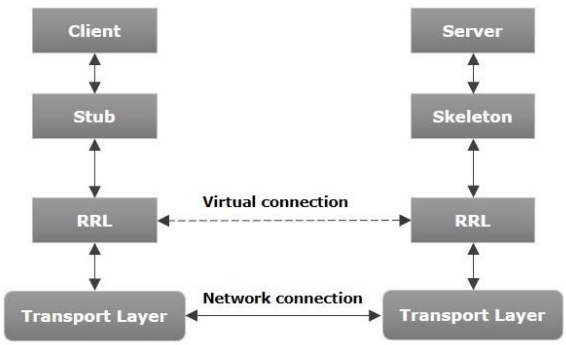
Architecture of an RMI Application

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.

RMI Architecture



To write an RMI Java application, you would have to follow the steps given below –

- Define the remote interface
- Develop the implementation class (remote object)

- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Code:

Adder.java

```
import java.rmi.*;
public interface Adder extends Remote {
	public int add(int x,int y)throws RemoteException;
}
```

AdderRemote.java

```
// Implementing the remote interface
public class AdderRemote implements Adder {
	// Implementing the interface method
	public int add(int x, int y) {
		return x+y;
	}
}
```

Client.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
   private Client() {}

public static void main(String[] args) {
      try {
         // Getting the registry
         Registry registry = LocateRegistry.getRegistry(null);

         // Looking up the registry for the remote object
         Adder stub = (Adder) registry.lookup("Hello");

         // Calling the remote method using the obtained object
```

```
         int result = stub.add(Integer.parseInt(args[0]),
Integer.parseInt(args[1]));
         System.out.println("Result From Server: " + result);

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

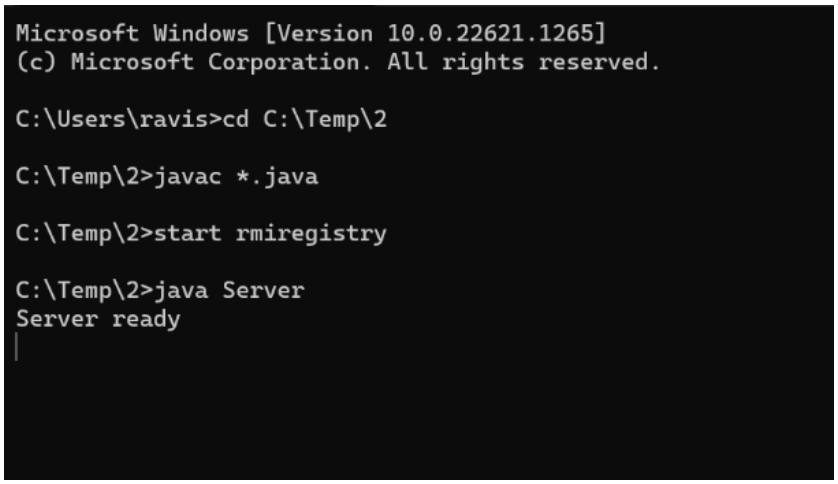
Server.java

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server extends AdderRemote {
   public Server() {}
   public static void main(String args[]) {
      try {
         // Instantiating the implementation class
         AdderRemote obj = new AdderRemote();

         // Exporting the object of implementation class
         // (here we are exporting the remote object to the stub)
         Adder stub = (Adder) UnicastRemoteObject.exportObject(obj, 0);

         // Binding the remote object (stub) in the registry
         Registry registry = LocateRegistry.getRegistry();

         registry.bind("Hello", stub);
         System.err.println("Server ready");
      } catch (Exception e) {
         System.err.println("Server exception: " + e.toString());
         e.printStackTrace();
      }
   }
```

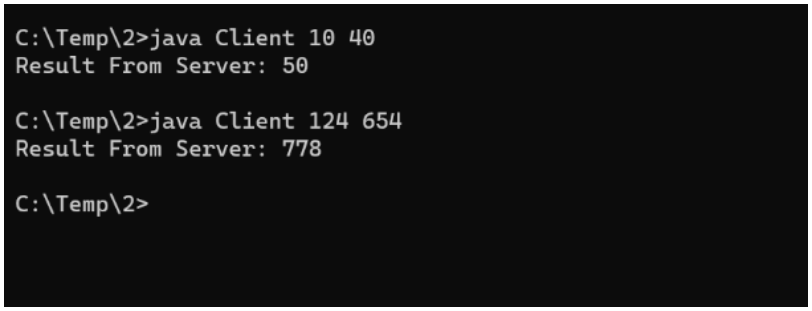
```
}
```

Output:

Compiling java files, starting rmiregistry and executing Server



Client:



Conclusions:

In conclusion, the remote method invocation (RMI) in Java is a powerful technology that enables distributed computing across different machines connected via a network. The experiment performed on RMI has shown that it is possible to invoke methods on remote objects in Java using RMI, which can facilitate communication between different Java applications.

Postlab Questions:

1. What are the different times at which a client can be bound to a server?
2. How does a binding process locate a server?
3. Name some optimization methods adopted for better performance of distributed applications using RPC and RMI.

LAB 4

Aim: To Implement a Message Queueing System

Introduction:

A message queueing system is a software architecture pattern that enables communication between different parts of a distributed system by allowing them to exchange messages. In this system, messages are stored in a queue and are retrieved by consumers when they are ready to process them. This decouples the producers and consumers of messages, allowing them to operate independently and asynchronously.

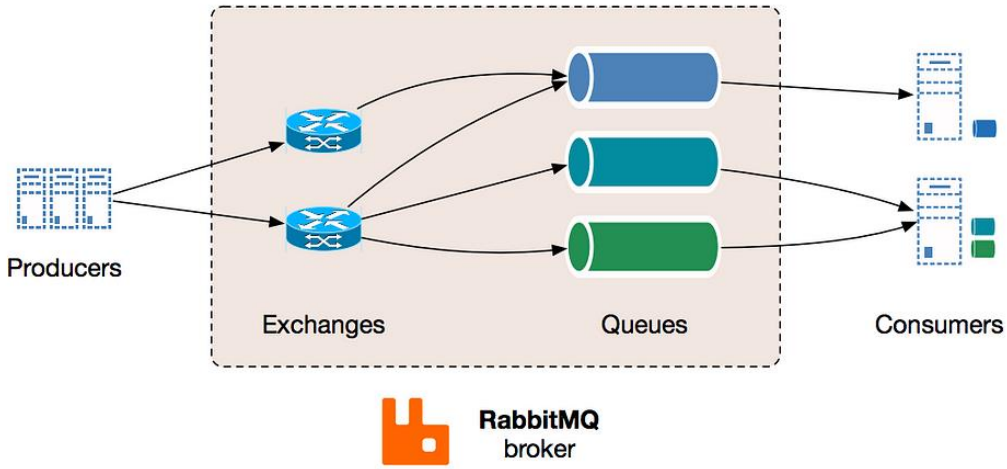
Message queueing systems can be implemented using various technologies, including open-source solutions like Apache Kafka, RabbitMQ, and ActiveMQ. In this practical we are going to implement a simple message queueing system using RabbitMQ.

RabbitMQ Model:

RabbitMQ is one of the most widely used message brokers, it acts as the message broker, "the mailman", a microservice architecture needs.

RabbitMQ consists of:

1. producer — the client that creates a message
2. consumer — receives a message
3. queue — stores messages
4. exchange — enables to route messages and send them to queues



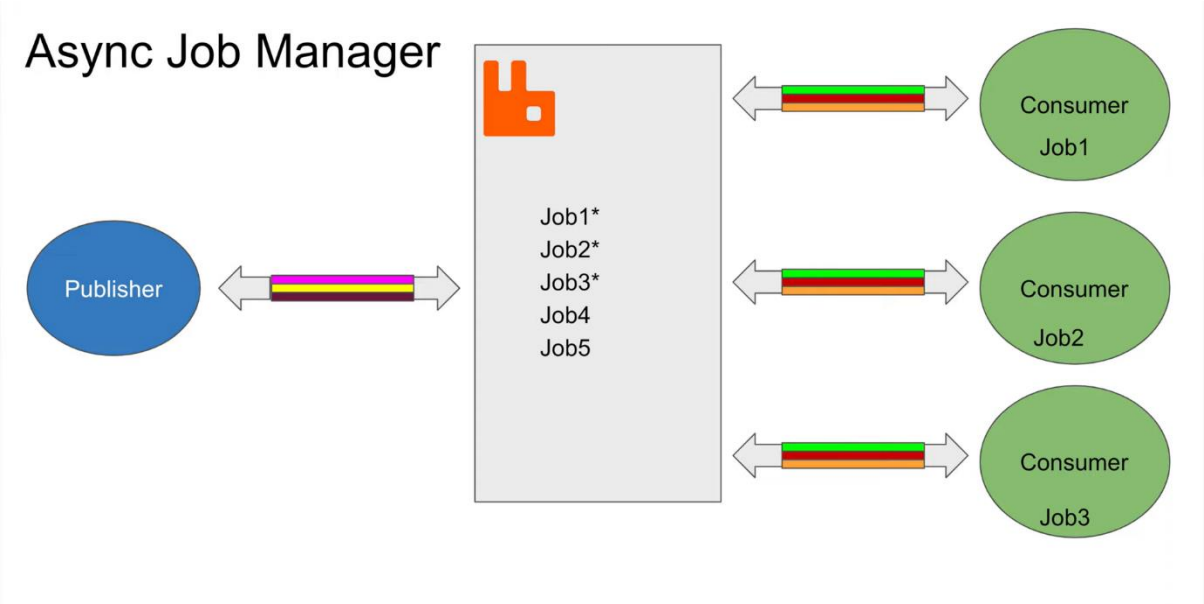
The system functions in the following way:

1. producer creates a message and sends it to an exchange
2. exchange receives a message and routes it to queues subscribed to it
3. consumer receives messages from those queues he/she is subscribed to

Implementation:

We are going to implement a job manager as described in the below figure.
Components of our message queueing system are:

- Publisher – produces jobs/messages into the queue
- Consumers – consumes the jobs
- RabbitMQ broker – contains the exchange and queue
- Connections – denoted by double-sided arrows
- Channels – denoted by colourful bands within the connections



Technologies Used:

- Docker
- RabbitMQ Image
- Node.js
- amqplib Library

Step 1: Run RabbitMQ's Docker Image

```
D:\8th Sem\DC\LAB Message Queueing System\rabbitmq>docker run --name rabbitmq -p 5672:5672 rabbitmq
Unable to find image 'rabbitmq:latest' locally
latest: Pulling from library/rabbitmq
5544ebdc0c7b: Pull complete
56fd8067e26d: Pull complete
50f617f636f4: Pull complete
dfa68bb7204a: Pull complete
c55a28a6ac3f: Pull complete
0f876b9b5491: Pull complete
c20f5fdda65c: Pull complete
c123cae549cf: Pull complete
3a78199d9e00: Pull complete
5104724e9223: Pull complete
Digest: sha256:7c74642976b61aafb7254a0762606bc8ac5ead30e96e07d3d260d73839a436ce
Status: Downloaded newer image for rabbitmq:latest
2023-03-22 11:12:22.384843+00:00 [notice] <0.44.0> Application syslog exited with reason: stopped
2023-03-22 11:12:22.406593+00:00 [notice] <0.230.0> Logging: switching to configured handler(s); following messages may not be visibl
e in this log output
2023-03-22 11:12:22.480842+00:00 [notice] <0.230.0> Logging: configured log handlers are now ACTIVE
2023-03-22 11:12:23.320836+00:00 [info] <0.230.0> ra: starting system quorum_queues
2023-03-22 11:12:23.321141+00:00 [info] <0.230.0> starting Ra system: quorum_queues in directory: /var/lib/rabbitmq/mnesia/rabbit@4bf
5eff3ee60/quorum/rabbit@4bf5eff3ee60
2023-03-22 11:12:23.571422+00:00 [info] <0.266.0> ra system 'quorum_queues' running pre init for 0 registered servers
2023-03-22 11:12:23.605560+00:00 [info] <0.267.0> ra: meta data store initialised for system quorum_queues. 0 record(s) recovered
2023-03-22 11:12:23.670433+00:00 [notice] <0.272.0> WAL: ra_log_wal init, open tbls: ra_log_open_mem_tables, closed tbls: ra_log_clos
ed_mem_tables
2023-03-22 11:12:23.692262+00:00 [info] <0.230.0> ra: starting system coordination
2023-03-22 11:12:23.692399+00:00 [info] <0.230.0> starting Ra system: coordination in directory: /var/lib/rabbitmq/mnesia/rabbit@4bf5
eff3ee60/coordination/rabbit@4bf5eff3ee60
2023-03-22 11:12:23.696248+00:00 [info] <0.279.0> ra system 'coordination' running pre init for 0 registered servers
2023-03-22 11:12:23.698733+00:00 [info] <0.280.0> ra: meta data store initialised for system coordination. 0 record(s) recovered
```

Step 2: Write a Producer Program - *publisher.js*

```
const amqp = require("amqplib");
const msg = {number : process.argv[2]}
connect()

async function connect() {
    try{
        const connection = await amqp.connect("amqp://localhost:5672");
        const channel = await connection.createChannel();
        const result = await channel.assertQueue("jobs");
        channel.sendToQueue("jobs", Buffer.from(JSON.stringify(msg)))
        console.log(`Job sent successfully ${msg.number}`)
    }
    catch(err){
        console.error(err)
    }
}
```

- A Node Library named “amqplib” is used to implement AMQP (Advanced Message Queueing Protocol)
- We then create a connection with the RabbitMQ server.
- Then a channel is created using connection’s createChannel() function
- This channel is used to create a new queue named “jobs” which resides within our RabbitMQ broker
- A new message is enqueued within the queue. In other words, a new job is produced. The content of this message is provided as a command line argument when we run our producer program

Step 3: Write a Consumer Program – *consumer.js*

```
const amqp = require("amqplib");

connect();

async function connect() {
  try {
    const connection = await amqp.connect("amqp://localhost:5672");
    const channel = await connection.createChannel();
    const result = await channel.assertQueue("jobs");

    channel.consume("jobs", message => {

        const input = JSON.parse(message.content.toString());

        console.log(`Received Job with input ${input.number}`)
        if(input.number == 22){
            // Process Job number 10
            channel.ack(message)
        }


    })

    console.log("Waiting for messages..")

  } catch (err) {
    console.error(err);
  }
}
```

- Here too, we create connection and channel the same way as in our publisher.js program
- Then we write functionality to consume the messages already present in the queue
- Let us say that our consumer only consumes message number 22. Hence, if the queue has a message number 22, it will be consumed by the consumer and an acknowledgement will be passed to the RabbitMQ server. Subsequently the message number 22 will be dequeued

Step 4: Testing our system

Running Producer – publisher.js

Publish job 10

```
PS D:\8th Sem\DC\LAB Message Queueing System\rabbitmq> npm run publish 10

> rabbitmq@1.0.0 publish D:\8th Sem\DC\LAB Message Queueing System\rabbitmq
> node publisher.js "10"

Job sent successfully 10
```

Publish job 20

```
PS D:\8th Sem\DC\LAB Message Queueing System\rabbitmq> npm run publish 20

> rabbitmq@1.0.0 publish D:\8th Sem\DC\LAB Message Queueing System\rabbitmq
> node publisher.js "20"

Job sent successfully 20
```

Publish job 35

```
PS D:\8th Sem\DC\LAB Message Queueing System\rabbitmq> npm run publish 35

> rabbitmq@1.0.0 publish D:\8th Sem\DC\LAB Message Queueing System\rabbitmq
> node publisher.js "35"

Job sent successfully 35
```

Publish job 22

```
PS D:\8th Sem\DC\LAB Message Queueing System\rabbitmq> npm run publish 22

> rabbitmq@1.0.0 publish D:\8th Sem\DC\LAB Message Queueing System\rabbitmq
> node publisher.js "22"

Job sent successfully 22
```

Running Consumer – consumer.js

All the jobs displayed:

```
PS D:\8th Sem\DC\LAB Message Queueing System\rabbitmq> npm run consume

> rabbitmq@1.0.0 consume D:\8th Sem\DC\LAB Message Queueing System\rabbitmq
> node consumer.js

Waiting for messages..
Received Job with input 10
Received Job with input 20
Received Job with input 35
Received Job with input 22
```

Job 22 consumed:

```
PS D:\8th Sem\DC\LAB Message Queueing System\rabbitmq> npm run consume

> rabbitmq@1.0.0 consume D:\8th Sem\DC\LAB Message Queueing System\rabbitmq
> node consumer.js

Waiting for messages..
Received Job with input 10
Received Job with input 20
Received Job with input 35
```

Conclusion:

- Message queueing systems, its need, architecture, and implementation were understood
- A simple message queueing system was designed and executed using RabbitMQ message broker.

Github Link: https://github.com/ravisinghk/Message-Queueing-System

References:

- *What is a Message Queue and When should you use Messaging Queue Systems Like RabbitMQ and Kafka*. (2020, May 2). YouTube. https://www.youtube.com/watch?v=W4_aGb_MOls
- *What is RabbitMQ?* (2020, November 10). YouTube. https://www.youtube.com/watch?v=7rkeORD4jSw
- *RabbitMQ Crash Course*. (2019, October 18). YouTube. https://www.youtube.com/watch?v=Cie5v59mrTg
- *8 Basic Docker Commands || Docker Tutorial 4*. (2019, October 28). YouTube. https://www.youtube.com/watch?v=xGn7cFR3ARU

- Peng Yang, L. (2022, December 4). *System Design—Message Queues*. Medium. https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22

Postlab Questions:

1.What is message Queueing?

2.What are the benefits of message Queueing?

LAB 5

Aim: To implement group communication

Lab Outcome:

Develop test and debug using Message-Oriented Communication or RPC/RMI based client-server programs.

Theory:

Group communication is a paradigm for multi-party communication that is based on the notion of groups as a main abstraction. A group is a set of parties that, presumably, want to exchange information in a reliable, consistent manner. For example:

- The participants of a message-based conferencing tool may constitute a group. Ideally, in order to have meaningful communication, each participant wants to receive all communicated messages from each other participant. Moreover, if one message is a response to another, the original message should be delivered before the response. (In this example, if two participants originate messages independently at about the same time, the order in which such independent messages are delivered is not important)

- The set of replicas of a fault-tolerant database server may constitute a group. Consider updating messages to the server. Since the contents of the database depend on the history of all update messages received, all updates must be delivered to all replicas. Furthermore, all updates must be delivered in the same order. Otherwise, inconsistencies may arise.

Group Communication Primitives

Group communication is implemented using middleware that provides two sets of primitives to the application:

- Multicast primitive (e.g., post): This primitive allows a sender to post a message to the entire group.
- Membership primitives (e.g., join, leave, query_membership): These primitives allow a process to join or leave a particular group, as well as to query the group for the list of all current participants.

Three types of group communication:

- One to many (single sender and multiple receivers)

In this scheme, there are multiple receivers for a message sent by a single sender. The one-to-many scheme is also known as multicast communication. A special case of multicast communication is broadcast communication, in which the message is sent to all processors connected to a network.

- Many to one (multiple senders and single receiver)

➔ Multiple senders send messages to a single receiver.

➔ The single receiver may be selective or nonselective.

➔ A *selective receiver* specifies a unique sender; a message exchange takes place only if that sender sends a message.

➔ A nonselective receiver specifies a set of senders, and if anyone sender in the set sends a message to this receiver, a message exchange takes place - an important issue related to the many-to-one communication scheme is nondeterminism

● Many too many (multiple senders and multiple receivers) ➔

Multiple senders send messages to multiple receivers.

➔ An important issue related to many-to-many communication scheme is that of ordered message delivery

➔ Ordered message delivery ensures that all messages are delivered to all receivers in an order acceptable to the application. This property is needed by many applications for its correct functioning.

➔ Ordered message delivery requires message sequencing.

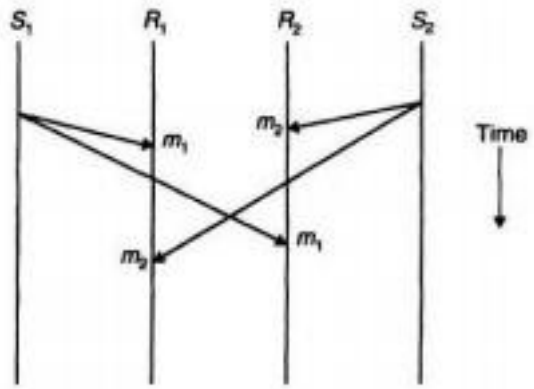


Fig. 3.14 No ordering constraint for message delivery.

➔ The commonly used semantics for ordered delivery of multicast messages are absolute ordering, consistent ordering, and causal ordering.

Steps to run the Single Client Server Communication application

1. Start GossipServer program. It will be ready to accept connections from the GossipClient.
2. On another terminal start the GossipClient program and send some message to GossipServer.
3. GossipServer will display the output.

Steps to run the Multi Client Server Communication application

1. Start Server program. It will be ready to accept connections from the Master.
2. On another terminal start the Master program followed by the Slave and send some message from the Master to the Slave.
3. Multiple Slaves can be started to depict group communication.

(Code & Output:)

Single Client Server Communication Code

GossipClient.java

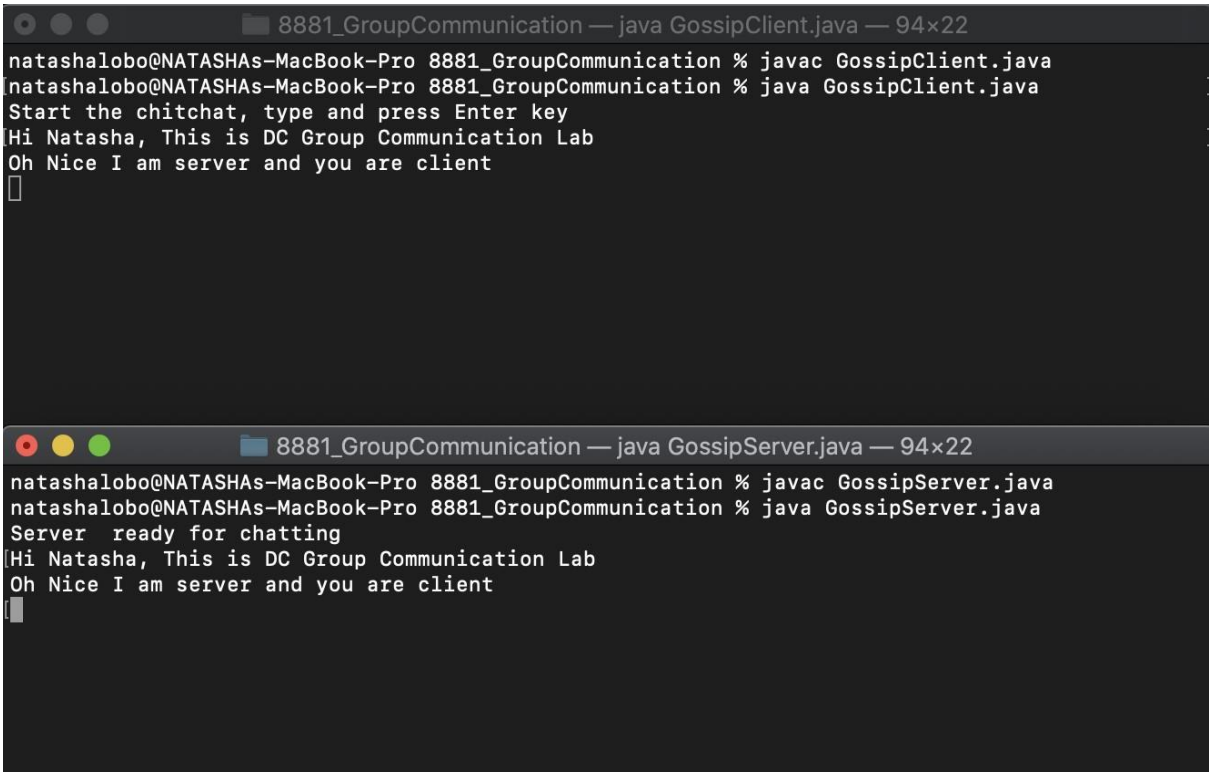
```
import java.io.*;
import java.net.*;
public class
GossipClient
{
  public static void main(String[] args) throws Exception
  {
    Socket sock = new Socket("127.0.0.1", 3000);

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

    OutputStream ostream = sock.getOutputStream();
    PrintWriter pwrite = new PrintWriter(ostream, true);

                        InputStream istream = sock.getInputStream();
    BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));

    System.out.println("Start the chitchat, type and press Enter key");

    String receiveMessage, sendMessage;
   while(true)
    {
      sendMessage = keyRead.readLine();
     pwrite.println(sendMessage);
     pwrite.flush();
      if((receiveMessage = receiveRead.readLine()) != null)
      {
         System.out.println(receiveMessage);
      }
     }
   }
}
```

GossipServer.java

```
import java.io.*;  import
java.net.*;  public class
GossipServer
```

```
{
  public static void main(String[] args) throws Exception
  {
     ServerSocket sersock = new ServerSocket(3000);
     System.out.println("Server  ready for chatting");
     Socket sock = sersock.accept( );
                        // reading from keyboard (keyRead object)
     BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
                             // sending to client (pwrite object)
     OutputStream ostream = sock.getOutputStream();
     PrintWriter pwrite = new PrintWriter(ostream, true);

                        // receiving from server ( receiveRead  object)
     InputStream istream = sock.getInputStream();
     BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));

     String receiveMessage, sendMessage;
    while(true)
     {
      if((receiveMessage = receiveRead.readLine()) != null)
      {
        System.out.println(receiveMessage);
      }
      sendMessage = keyRead.readLine();
     pwrite.println(sendMessage);
     pwrite.flush();
     }
   }
}
```

Single Client Server Communication Output



Multi Client Server Communication

Code Server.java import java.util.*;
import java.io.*; import java.net.*;

```
public class Server {
   static ArrayList<ClientHandler> clients;

   public static void main(String args[]) throws Exception {
      // Server server = new Server();
      ServerSocket MyServer = new ServerSocket(8881);
     clients = new ArrayList<ClientHandler>();
      Socket ss = null;
      Message msg = new
     Message(); int count = 0;
     while (true) { ss = null; try {
            ss = MyServer.accept();
           DataInputStream din = new DataInputStream(ss.getInputStream());
          DataOutputStream dout = new DataOutputStream(ss.getOutputStream());
           ClientHandler chlr = new ClientHandler(ss, din, dout, msg);
           Thread t = chlr;
          if (count > 0)
              clients.add(chlr);
```

```
            count++;
            // System.out.println(threads);
            t.start();
          } catch (Exception E) {
            continue;
          }
      }
    }
}

class Message {
  String msg;

    public void set_msg(String msg) {
      this.msg = msg;
    }

    public void get_msg() {
      System.out.println("\nNEW GROUP MESSAGE: " + this.msg);
      for (int i = 0; i < Server.clients.size(); i++) {
        try {
            System.out.print("Client: " + Server.clients.get(i).ip + "; ");
            Server.clients.get(i).out.writeUTF(this.msg);
            Server.clients.get(i).out.flush();
         } catch (Exception e) {
            System.out.print(e);
          }
      }
    }
}

class ClientHandler extends Thread {
    DataInputStream in;
    DataOutputStream
   out;  Socket socket;  int
   sum;  float res;
    boolean conn;
    Message msg;
    String ip;
```

```
    public ClientHandler(Socket s, DataInputStream din, DataOutputStream dout,
Message  msg) {  this.socket = s;  this.in = din;  this.out = dout;  this.conn = true;
this.msg = msg;  this.ip = (((InetSocketAddress)
 this.socket.getRemoteSocketAddress()).getAddress()).toString().replace("/", "");
    }

    public void run() {
      while (conn == true)
      {  try {
             String input = this.in.readUTF();
             // System.out.println("From host " + this.ip + ':' +
            input);  // String msg = "From host " + this.ip + ':' +
            input;  this.msg.set_msg(input);  this.msg.get_msg();
          } catch (Exception E) {
            conn = false;
             System.out.println(E);
          }
       }
       closeConn();
    }

    public void closeConn() {
      try {
          this.out.close();
          this.in.close();
          this.socket.close();
       } catch (Exception E) {
          System.out.println(E);
       }
    }
 }
```

Master.java

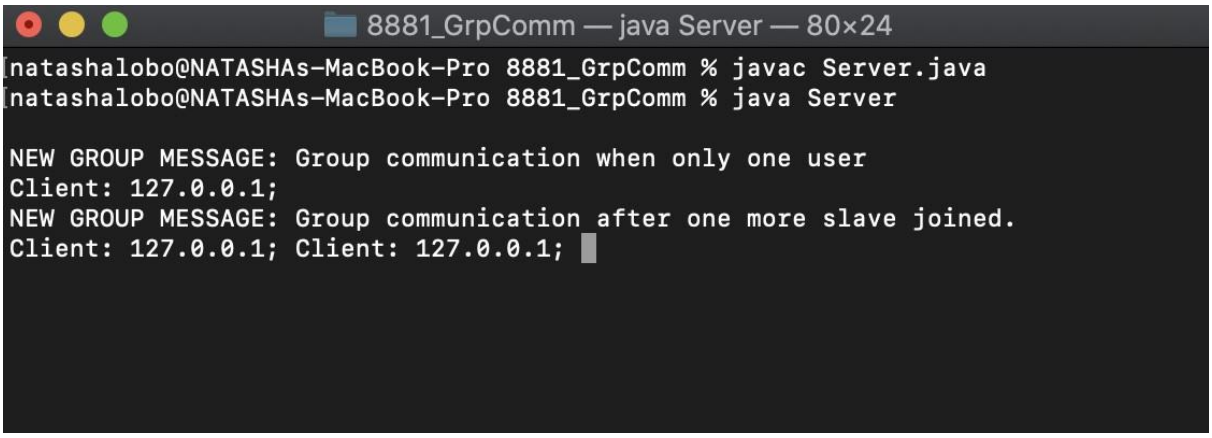
```
import java.io.*;
import java.util.*;
import java.net.*;

 public class Master {
    public static void main(String args[]) throws Exception {
       String send = "", r = "";
       Socket MyClient = new Socket("127.0.0.1", 8881);
       System.out.println("Connected as Master");
```

```
      DataInputStream din = new DataInputStream(MyClient.getInputStream());
      DataOutputStream dout = new
     DataOutputStream(MyClient.getOutputStream());  Scanner sc = new
     Scanner(System.in);  do {
         System.out.print("Message('close' to stop):
        ");  send = sc.nextLine();
        dout.writeUTF(send);  dout.flush();
      } while
     (!send.equals("stop"));
     dout.close();  din.close();
      MyClient.close();
   }
}
```

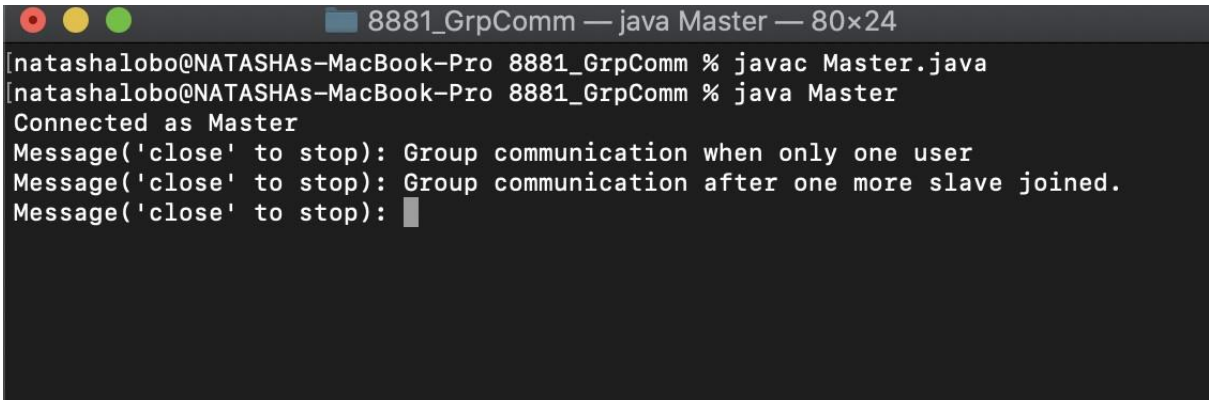
Slave.java

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

public class Slave {  public static void main(String args[])
  throws Exception {
      String r = "";
      Socket MyClient = new Socket("127.0.0.1", 8881);
      System.out.println("Connected as Slave");
      DataInputStream din = new
     DataInputStream(MyClient.getInputStream());  do {  r = din.readUTF();
         System.out.println("Master says: " + r);
      } while (!r.equals("stop"));
      din.close();
      MyClient.close();
   }
}
```

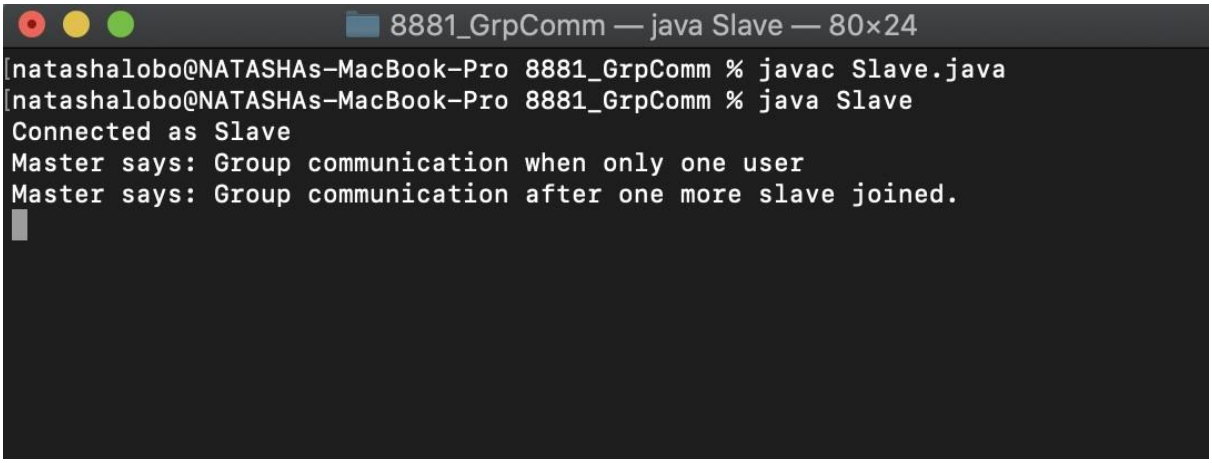
Multi Client Server Communication Output Server



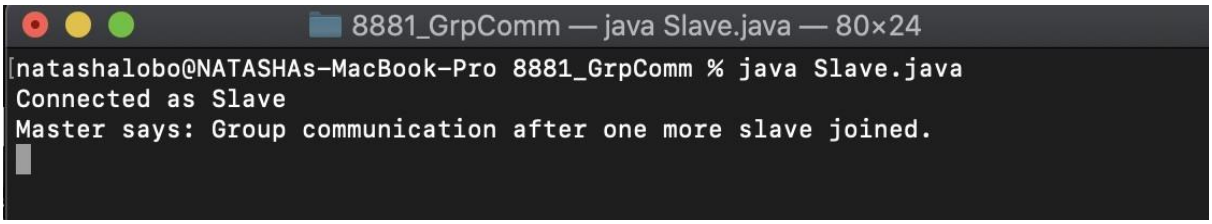
Master



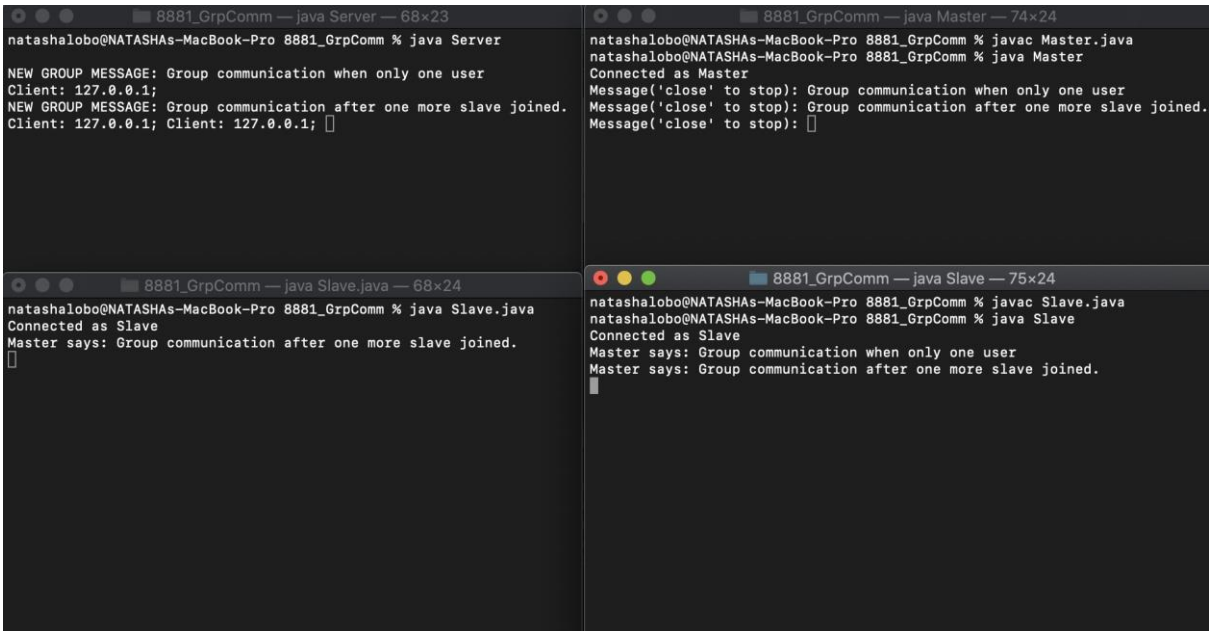
Slave



Another Slave



Group Communication Demonstration



Conclusions :

1. Implemented group communication using Java.
2. Understood and learnt the three types of group communication.

Postlab Questions:

1.Explain group communication.

2.Explain types of group communication.

LAB 6

Aim: To implement Lamport algorithm for logical clock synchronization

Lab Outcome:
Implement techniques for clock synchronization.

Theory:
The Lamport algorithm is a logical clock synchronization algorithm used in distributed systems to establish a partial ordering of events. The algorithm is based on the concept of logical clocks, which assign a timestamp to each event in a distributed system.

In Lamport's algorithm, each process maintains a logical clock that ticks whenever an event occurs at that process. The logical clock value assigned to an event is the maximum of the current logical clock value of the process and the timestamp of the incoming message that triggered the event.

The logical clocks in Lamport's algorithm satisfy the following two properties:

- If event A happens before event B, then the logical clock value of A is less than the logical clock value of B.
- If events A and B are concurrent, then the logical clock value of A is not equal to the logical clock value of B.

Using Lamport's algorithm, processes can synchronize their logical clocks so that they agree on the ordering of events. This enables distributed systems to reason about causality, and to detect and resolve conflicts that may arise due to concurrent events.

Lamport Algorithm
Lamport developed a "happens-before" notation to express this: If a and b are events in the same process, and a occurs before b, then a → b is true. If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a → b. This relationship is transitive i.e. a→b and b→c then a→c. Satisfying conditions for implementing clock: If a→b then c(a)<c(b). Implementation of logical clock:
Condition 1: If a and b are two events within the same process Pi and a occur before b then Ci(a) < Ci (b).
Condition 2: if a is the sending of a message by process Pi and b is the receipt of that message by process Pj then Ci(a) < Cj(b).
Condition 3: A clock Ci associated with a process Pi must always go forward never backward that is correction to the time of clock is done by +ive adding.

Total ordering and Partial ordering
Total ordering is an ordering that defines the exact order of every element in the series.

Partial ordering of elements in a series is an ordering that doesn't specify the exact order of every item, but only defines the order between certain key items that depend on each other.

The meaning of these words is the same in the context of distributed computing. The only significance of distributed computing to these terms is the fact that partial ordering of events is much commoner than total ordering. In a local, single-threaded application, the order in which events happen is ordered, implicitly, since the CPU can only do one thing at a time. In a distributed system, you generally only coordinate a partial ordering of those events that have a dependency on one another and let other events happen in whatever order they happen.

Example, taken from the comments: If you have three events {A, B, C}, then they are ordered if they always have to happen in the order A > B > C. However, if A must happen before C, but you don't care when B happens, then they are partially ordered. In this case we would say that the sequences A > B > C, A > C > B, and B > A > C all satisfy the partial ordering.

(Code & Output:)

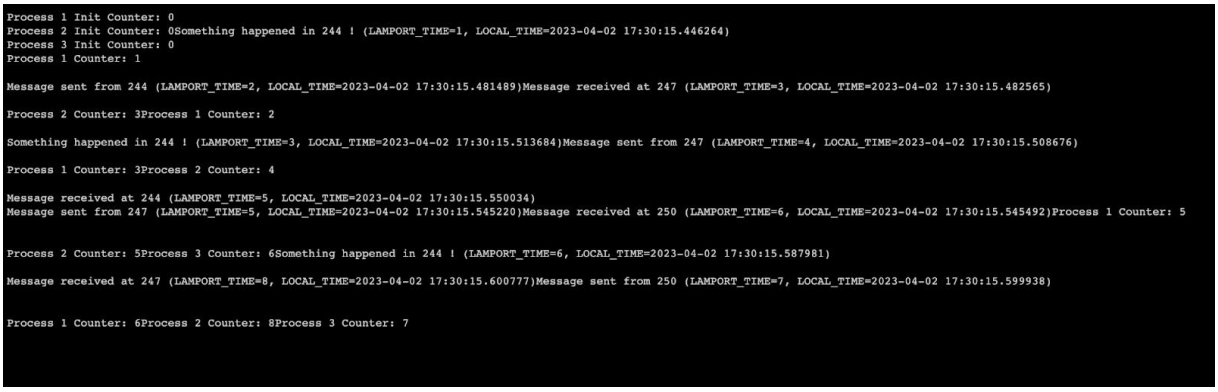
COLAB LINK:

https://colab.research.google.com/drive/1BW52qeCM_gam-23inrOc-xFdcizPJjE2?usp= sharing

```
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('Something 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
```

```
    print("Process 1 Init Counter: "+str(counter))
counter = event(pid, counter)     print("Process
1 Counter: "+str(counter))     counter =
send_message(pipe12, pid, counter)
print("Process 1 Counter: "+str(counter))
counter  = event(pid, counter)     print("Process
1 Counter: "+str(counter))     counter =
recv_message(pipe12, pid, counter)
print("Process 1 Counter: "+str(counter))
counter  = event(pid, counter)     print("Process
1 Counter: "+str(counter))

 def process_two(pipe21, pipe23):
    pid = getpid()
counter = 0
    print("Process 2 Init Counter: "+str(counter))
counter = recv_message(pipe21, pid, counter)
print("Process 2 Counter: "+str(counter))
counter = send_message(pipe21, pid, counter)
print("Process 2 Counter: "+str(counter))
counter = send_message(pipe23, pid, counter)
print("Process 2 Counter: "+str(counter))
counter = recv_message(pipe23, pid, counter)
print("Process 2 Counter: "+str(counter))


 def process_three(pipe32):
    pid = getpid()
counter = 0
    print("Process 3 Init Counter: "+str(counter))
counter = recv_message(pipe32, pid, counter)
print("Process 3 Counter: "+str(counter))
counter = send_message(pipe32, pid, counter)
print("Process 3 Counter: "+str(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()



Conclusions :

1. Learnt the functioning of Lamport's Algorithm for logical clock synchronization.
2. Understood the terminologies such as Partial ordering and Total ordering
3. In conclusion, a Lamport logical clock is an incrementing counter maintained in each process. Conceptually, this logical clock can be thought of as a clock that only has meaning concerning messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender (causality).

Postlab Questions:

1. Distinguish between physical clock and logical clock synchronization
2. Show the calculation of the time interval between two synchronizations of a physical clock
3. How will you implement Logical clocks by using counters?
4. Give an example of partial and total ordering of events

LAB 7

Aim: To implement Election Algorithms

Lab Outcome:
Implement techniques for Election Algorithms.

Theory:
In distributed systems, election algorithms are used to elect a leader or coordinator node among a group of nodes. The leader node is responsible for coordinating the activities of other nodes, managing resources, and making decisions on behalf of the group.

There are various election algorithms used in distributed systems, and the choice of algorithm depends on the requirements of the system, such as fault-tolerance, scalability, and availability.

One commonly used election algorithm is the Bully algorithm, which is a centralized algorithm. In this algorithm, the node with the highest priority becomes the leader, and if the leader fails, the next highest priority node takes over.

Another algorithm is the Ring algorithm, which is a decentralized algorithm. In this algorithm, nodes are arranged in a ring, and each node sends a message to its neighbor indicating its willingness to become the leader. The node with the highest priority becomes the leader.

A third algorithm is the Paxos algorithm, which is fault-tolerant and can handle network failures. In this algorithm, nodes propose a value to be chosen as the leader, and if the majority of nodes accept the proposal, the value is chosen as the leader.

These election algorithms ensure that a leader is elected in a fair and efficient manner, and they provide fault-tolerance and availability in distributed systems.

Bully Algorithm
The bully Algorithm proposed by Garcia-Molina follows the following algorithm

- When a process notices that the coordinator is no longer responding to requests, it initiates an election.
- A process, P, holds an election as follows:
  1. P sends an ELECTION message to all processes with higher numbers
  2. If no one responds, P wins the election and becomes the coordinator
  3. If one of the higher-up's answers, it takes over. P's job is done
- If a process can get an ELECTION message from one of its lower-numbered colleagues.

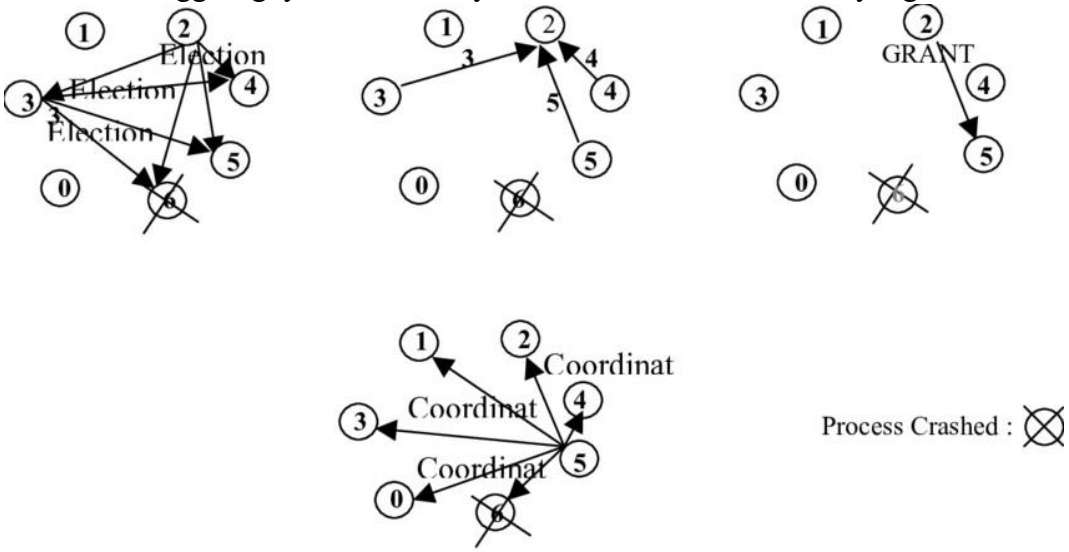
  The message arrives => the receiver sends an OK message back to the sender to indicate that he is alive and will take over.

  The receiver then holds an election, unless it is already holding one.
- Eventually, all processes give up but one does not give up and that one is the new coordinator.

It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

- If a process that previously went down came back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job.

· Thus, the biggest guy in town always wins, hence the name "bully algorithm."



(Code & Output:)

Java Program

```
import java.io.*;
import
java.util.Scanner;

class Main{
    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.println("Status:");
           sta[i]=in.nextInt();
            System.out.println("Priority");
            pro[i] = in.nextInt();
        }

        System.out.println("Which process will initiate the 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:

```
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 the 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

...Program finished with exit code 0
Press ENTER to exit console.

Conclusions :

1. Understood and learnt the concept of Election Algorithms and significance of a coordinator in a system.
2. Implemented the Bully Election Algorithm using Java.
3. The coordinator is elected using the Bully algorithm by considering the highest process id in the network.

Postlab Questions:

1. What is the role of a coordinator in Distributed systems?
2. Compare Bully and Ring algorithms.

LAB 8
Aim: To implement Mutual Exclusion / Deadlock Detection

Lab Outcome:
Demonstrate Mutual Exclusion algorithms and deadlock handling

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 cannot 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.

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

Requirements of Mutual exclusion Algorithm:

- **No Deadlock:**
  Two or more site should not endlessly wait for any message that will never arrive.

- **No Starvation:**
  Any site should not wait indefinitely to execute critical section while other sites 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.

- **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 cannot 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

Non-token based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme
- Example: Lamport's algorithm, Ricart–Agrawala algorithm

Quorum based approach:

- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a quorum.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion
- Example: Maekawa's Algorithm

Deadlock Handling:

The following are the strategies used for Deadlock Handling in Distributed System:

1. **Deadlock Prevention:** As the name implies, this strategy ensures that deadlock can never happen because system designing is carried out in such a way. If any one of the deadlock-causing conditions is not met then deadlock can be prevented. Following are the three

methods used for preventing deadlocks by making one of the deadlock conditions to be unsatisfied:

Collective Requests: In this strategy, all the processes will declare the required resources for their execution beforehand and will be allowed to execute only if there is the availability of all the required resources. When the process ends up with processing then only resources will be released. Hence, the hold and wait condition of deadlock will be prevented.
But the issue is initial resource requirements of a process before it starts are based on an assumption and not because they will be required. So, resources will be unnecessarily occupied by a process and prior allocation of resources also affects potential concurrency.

Ordered Requests: In this strategy, ordering is imposed on the resources and thus, process requests for resources in increasing order. Hence, the circular wait condition of deadlock can be prevented.
An ordering strictly indicates that a process never asks for a low resource while holding a high one.
There are two more ways of dealing with global timing and transactions in distributed systems, both of which are based on the principle of assigning a global timestamp to each transaction as soon as it begins.
It is better to give priority to the old processes because of their long existence and might be holding more resources.
It also eliminates starvation issues as the younger transaction will eventually be out of the system.

Pre-emption: Resource allocation strategies that reject no-pre-emption conditions can be used to avoid deadlocks.

Wait-die: If an older process requires a resource held by a younger process, the latter will have to wait. A young process will be destroyed if it requests a resource controlled by an older process.

Wound-wait: If an old process seeks a resource held by a young process, the young process will be pre-empted, wounded, and killed, and the old process will resume and wait. If a young process needs a resource held by an older process, it will have to wait.

2. **Deadlock Avoidance:** In this strategy, deadlock can be avoided by examining the state of the system at every step. The distributed system reviews the allocation of resources and wherever it finds an unsafe state, the system backtracks one step and again comes to the

safe state. For this, resource allocation takes time whenever requested by a process. Firstly, the system analysis occurs whether the granting of resources will make the system in a safe state or unsafe state then only allocation will be made.

A safe state refers to the state when the system is not in deadlocked state and order is there for the process regarding the granting of requests.
An unsafe state refers to the state when no safe sequence exists for the system. Safe sequence implies the ordering of a process in such a way that all the processes run to completion in a safe state.

3. **Deadlock Detection and Recovery:** In this strategy, deadlock is detected and an attempt is made to resolve the deadlock state of the system. These approaches rely on a Wait-For-Graph (WFG), which is generated and evaluated for cycles in some methods.

The following two requirements must be met by a deadlock detection algorithm:

> Progress: In a given period, the algorithm must find all existing deadlocks. There should be no deadlock existing in the system which is undetected under this condition. To put it another way, after all, wait-for dependencies for a deadlock have arisen, the algorithm should not wait for any additional events to detect the deadlock.
> No False Deadlocks: Deadlocks that do not exist should not be reported by the algorithm which is called phantom or false deadlocks.

There are different types of deadlock detection techniques:

> Centralized Deadlock Detector: The resource graph for the entire system is managed by a central coordinator. When the coordinator detects a cycle, it terminates one of the processes involved in the cycle to break the deadlock. Messages must be passed when updating the coordinator's graph. Following are the methods:
> A message must be provided to the coordinator whenever an arc is created or removed from the resource graph.
>
> Hierarchical Deadlock Detector: In this approach, deadlock detectors are arranged in a hierarchy. Here, only those deadlocks can be detected that fall within their range.
>
> Distributed Deadlock Detector: In this approach, detectors are distributed so that all the sites can fully participate to resolve the deadlock state. In one of the following below four classes for the Distributed Detection Algorithm- The probe-based scheme can be used for this purpose. It follows local WFGs to detect local deadlocks and probe messages to detect global deadlocks.

There are four classes for the Distributed Detection Algorithm:

- *Path-pushing*: In path-pushing algorithms, the detection of distributed deadlocks is carried out by maintaining an explicit global WFG.
- *Edge-chasing*: In an edge-chasing algorithm, probe messages are used to detect the presence of a cycle in a distributed graph structure along the edges of the graph.
- *Diffusion computation*: Here, the computation for deadlock detection is dispersed throughout the system's WFG.
- *Global state detection*: The detection of Distributed deadlocks can be made by taking a snapshot of the system and then inspecting it for signs of a deadlock.

Implementation:

Suzuki-Kasami Algorithm:

1. **To enter Critical section:**
   - When a site S_i wants to enter the critical section and it does not have the token then it increments its sequence number $\mathbf{RN_i[i]}$ and sends a request message **REQUEST(i, sn)** to all other sites in order to request the token. Here **sn** is update value of $\mathbf{RN_i[i]}$
   - When a site S_j receives the request message **REQUEST(i, sn)** from site S_i, it sets $\mathbf{RN_j[i]}$ to maximum of $\mathbf{RN_j[i]}$ and **sn** i.e $\mathbf{RN_j[i]}$ = max($\mathbf{RN_j[i]}$, **sn**).
   - After updating $\mathbf{RN_j[i]}$, Site S_j sends the token to site S_i if it has token and $\mathbf{RN_j[i]}$ = **LN[i]** + 1
2. **To execute the critical section:**
   - Site S_i executes the critical section if it has acquired the token.
3. **To release the critical section:**
   After finishing the execution Site S_i exits the critical section and does following:
   - sets **LN[i]** = $\mathbf{RN_i[i]}$ to indicate that its critical section request $\mathbf{RN_i[i]}$ has been executed
   - For every site S_j, whose ID is not present in the token queue **Q**, it appends its ID to **Q** if $\mathbf{RN_i[j]}$ = **LN[j]** + 1 to indicate that site S_j has an outstanding request.
   - After above updation, if the Queue **Q** is non-empty, it pops a site ID from the **Q** and sends the token to site indicated by popped ID.
   - If the queue **Q** is empty, it keeps the token

Message Complexity:
The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution.

This N messages involves

- (N – 1) request messages
- 1 reply message

Code:

```
import keyboard
import time
import threading

runningP = -1

# RN arrays of processes
RN = {
	0: [0, 0, 0, 0, 0],
	1: [0, 0, 0, 0, 0],
	2: [0, 0, 0, 0, 0],
	3: [0, 0, 0, 0, 0],
	4: [0, 0, 0, 0, 0]
}

token = {
	"token_owner": 2,
	"Q": [],
	"LN": [0, 0, 0, 0, 0],
	"isRunning": False
}


def dispCurrentRNState():
	for key, value in RN.items():
		print(key, ": ", value)


def updateRN(processNo, sequenceNumber):
	for key, value in RN.items():
		value[processNo] = max(value[processNo], sequenceNumber)

# Execute cs and remaining tasks


def executeCS(processForCS):
    print("\n*****************************\n")
    print(f"Process {processForCS} executing CS...")
    print('Token owner is: {}'.format(token["token_owner"]))

    time.sleep(10)
    print(f'\nProcess {processForCS} has completed running CS')

    # Process completed CS
    token["isRunning"] = False

    #update LN
    token["LN"][processForCS] = RN[processForCS][processForCS]
    # print(f"Process Completed CS")


    #Check For Outstanding Requests
```

```
    # For every site Sj, whose ID is not present in the token queue Q, it appends its ID to Q if RNi[j] = LN[j] + 1 to indicate that site Sj has an outstanding request.
    for index, val in enumerate(RN[token["token_owner"]]):
        # print("Running P: ", runningP)
        if(val == token["LN"][index] + 1 and index != runningP and index not in token["Q"]):
            # outstanding Requests
            print(f'Process {index}\'s request is outstanding, it will be added to Token\'s Queue')
            token["Q"].append(index)
            print(f'Queue: {token["Q"]}')


    #Handing out the token
    if(len(token["Q"]) != 0):
        # pop a process from the queue and give it the token
        poppedPs = token["Q"].pop(0)
        token["token_owner"] = poppedPs
        token["isRunning"] = True
        executeCS(poppedPs)


if __name__ == "__main__":
       # print("Press Key E to exit")
       print("Running Main Again")

    # Display Current State of RN Arrays
       print("Current RN Arrays: ")
       dispCurrentRNState()
       print(" ")
       print('Token owner is: {}'.format(token["token_owner"]))

       while True:

              if(token["isRunning"]):

                     processes = input(
                            "Enter Process Numbers which want to access C.S separated by space (Click N for None): ")

                     if(processes != 'N'):
                            psList = processes.strip().split(" ")
                            print(" ")

                            for ps in psList:
                                   processForCS = int(ps)
                                   print(f"***** Process {processForCS} *****")
                                   seqNo = RN[processForCS][processForCS]+1
                                   # Broadcasting Request

                                   print(f"Process No.: {processForCS}")
                                   print(f"Sequence No.: {seqNo}")
```

```
                                print(f"Broadcasting  Request  ({processForCS}  ,
{seqNo}) .......")
                                time.sleep(2)
                                print("Broadcast complete")
                                print(" ")

                                # Updating RN Arrays
                                print("Updating RN Arrays at all process sites")
                                updateRN(processForCS, seqNo)
                                print("Current RN Arrays: ")
                                dispCurrentRNState()
                                print(" ")



             else:

                    processForCS  =  int(input("Enter  Process  No.  which  wants  to
access C.S: "))
                    seqNo = RN[processForCS][processForCS]+1
            # Broadcasting Request

                    print(f"Process No.: {processForCS}")
                    print(f"Sequence No.: {seqNo}")
                    print(f"Broadcasting   Request   ({processForCS}   ,   {seqNo})
.......")
                    time.sleep(2)
                    print("Broadcast complete")
                    print(" ")

             # Updating RN Arrays
                    print("Updating RN Arrays at all process sites")
                    updateRN(processForCS, seqNo)
                    print("Current RN Arrays: ")
                    dispCurrentRNState()
                    print(" ")

                    # Check condition of sending token: RNj[i] = LN[i] + 1
                    if(RN[token["token_owner"]][processForCS]                    ==
token["LN"][processForCS] + 1):
                           # give the token
                           print(f"Conditions     met,     giving     token     to
{processForCS}...")
                           token["token_owner"] = processForCS

                           # print(f"New Token Owner: {token["token_owner"]}")
                           print('Token owner is: {}'.format(token["token_owner"]))

                           token["isRunning"] = True
                           runningP = processForCS

                           thread       =       threading.Thread(target=executeCS,
args=(processForCS, ))
                           thread.start()
```

```
                        print("Main Continuing Running")

            if keyboard.is_pressed('E'):
                  break
```

Output:

```
Current RN Arrays:
0 :  [0, 0, 0, 0, 0]
1 :  [0, 0, 0, 0, 0]
2 :  [0, 0, 0, 0, 0]
3 :  [0, 0, 0, 0, 0]
4 :  [0, 0, 0, 0, 0]

Token owner is: 2
Enter Process No. which wants to access C.S: 1
Process No.: 1
Sequence No.: 1
Broadcasting Request (1 , 1) .......
Broadcast complete

Updating RN Arrays at all process sites
Current RN Arrays:
0 :  [0, 1, 0, 0, 0]
1 :  [0, 1, 0, 0, 0]
2 :  [0, 1, 0, 0, 0]
3 :  [0, 1, 0, 0, 0]
4 :  [0, 1, 0, 0, 0]

Conditions met, giving token to 1...
Token owner is: 1

*****************************
Main Continuing Running

Process 1 executing CS...
Token owner is: 1
Enter Process Numbers which want to access C.S separated by space (Click N for None): 2 3

***** Process 2 *****
Process No.: 2
Sequence No.: 1
Broadcasting Request (2 , 1) .......
Broadcast complete
```

```
Updating RN Arrays at all process sites
Current RN Arrays:
0 :  [0, 1, 1, 0, 0]
1 :  [0, 1, 1, 0, 0]
2 :  [0, 1, 1, 0, 0]
3 :  [0, 1, 1, 0, 0]
4 :  [0, 1, 1, 0, 0]

***** Process 3 *****
Process No.: 3
Sequence No.: 1
Broadcasting Request (3 , 1) .......
Broadcast complete

Updating RN Arrays at all process sites
Current RN Arrays:
0 :  [0, 1, 1, 1, 0]
1 :  [0, 1, 1, 1, 0]
2 :  [0, 1, 1, 1, 0]
3 :  [0, 1, 1, 1, 0]
4 :  [0, 1, 1, 1, 0]

Enter Process Numbers which want to access C.S separated by space (Click N for None):
Process 1 has completed running CS
Process 2's request is outstanding, it will be added to Token's Queue
Queue: [2]
Process 3's request is outstanding, it will be added to Token's Queue
Queue: [2, 3]

******************************

Process 2 executing CS...
Token owner is: 2

Process 2 has completed running CS

******************************

Process 3 executing CS...
Token owner is: 3

Process 3 has completed running CS
```

Link to Code File:
https://drive.google.com/file/d/1HIm9UgHaVARTs_4rNj8FC6nQgKMKpgnJ/view?usp=share_link

Conclusion:
In conclusion, our experiment focused on studying mutual exclusion and deadlock handling in a distributed system. We implemented a simulation of the Suzuki-Kasami algorithm as a solution to the challenges we identified. The algorithm demonstrated how processes in a distributed system can coordinate with each other to prevent race conditions and ensure mutual exclusion, without causing deadlocks.

Postlab Questions:
1. Explain the different ways of recovery from deadlock.
2. What are the features of CMH algorithm

LAB 9

Aim: To implement Stateful and Stateless File Server.

Lab Outcome:
Describe the concepts of distributed File Systems with some case studies.

Theory:

Stateful and Stateless are two types of servers used in computer networks.

A stateful server is one that maintains the state of the client/server relationship across multiple requests. It keeps track of information about the client's previous interactions with the server, such as the client's session data, authentication credentials, and other contextspecific information.

For example :

a stateful server is a web server that maintains user session information across multiple requests. When a user logs in to a web application, the server stores the user's session data (such as their login credentials) and uses that information to authenticate the user on subsequent requests.

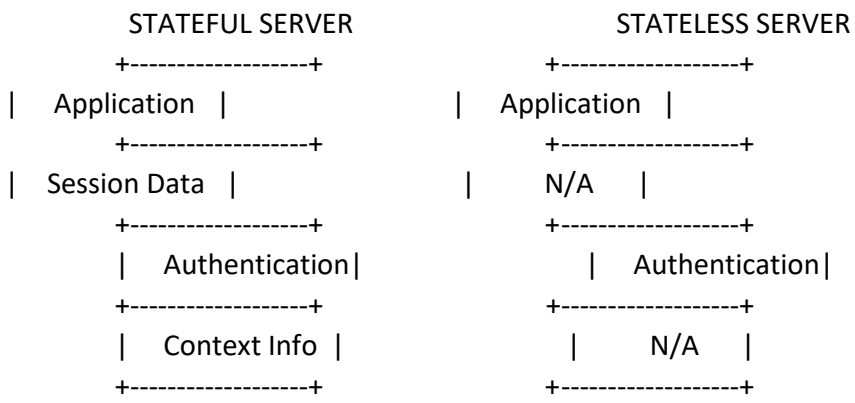
A stateless server does not keep track of any information about the client's previous interactions. Each request is treated as an independent event, and the server does not maintain any session data or other context-specific information.

For example :

a stateless server is a Domain Name System (DNS) server that provides IP addresses for domain names. Each request to the DNS server is independent of any previous requests, and the server does not maintain any session data or other context-specific information.

Overall, the choice between stateful and stateless servers depends on the specific requirements of the application and the trade-offs between scalability, reliability, and ease of implementation.

Here's a diagram to illustrate the difference between the two:



Code link:
https://drive.google.com/drive/folders/1zM9mFipx_uA5GMwkJQwjwpXqmgSk3gcB?usp=sharing

(Code & Output:)

STATEFUL SERVER

StateFulServer.PY

```
import socket
import threading
import os

class FileSystem:
    def __init__(self, root_path):
        self.root_path = root_path

    def create_file(self, path):
        full_path = os.path.join(self.root_path, path)
        with open(full_path, 'w') as f:
            f.write('')
        return "File created successfully"

    def read_file(self, path):
        full_path = os.path.join(self.root_path, path)
        with open(full_path, 'r') as f:
            content = f.read()
        return content

    def write_file(self, path, content):
        full_path = os.path.join(self.root_path, path)
        with open(full_path, 'w') as f:
            f.write(content)
        return "File written successfully"

    def delete_file(self, path):
        full_path = os.path.join(self.root_path, path)
        os.remove(full_path)
        return "File deleted successfully"
```

```
class StatefulFileServer:
    def __init__(self, host, port, file_system):
        self.host = host
        self.port = port
        self.file_system = file_system
        self.sessions = {}

    def run(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
sock:
            sock.bind((self.host, self.port))
            sock.listen()

            while True:
                conn, addr = sock.accept()
                client_thread =
threading.Thread(target=self.handle_client, args=(conn, addr))
                client_thread.start()

    def handle_client(self, conn, addr):
        addr_1 = input("Enter portNumber: ")
        client_id = addr[0] + ":" + addr_1
        print("Client: %s" % client_id)
        session_data = self.sessions.get(client_id, {})
        print(f"Stored data for client {client_id}: {session_data}")
        # Parse client request and perform file system operation
        request = conn.recv(1024).decode()
        response = self.handle_request(request, session_data)

        # Send response to client
        conn.sendall(response.encode())

        # Update session data
        self.sessions[client_id] = session_data

        conn.close()
```

```
    def handle_request(self, request, session_data):
        tokens = request.split()
        command = tokens[0]
        session_data[command] = tokens[1]

        if command == "CREATE":
            path = tokens[1]
            return self.file_system.create_file(path)

        elif command == "READ":
            path = tokens[1]
            return self.file_system.read_file(path)

        elif command == "WRITE":
            path = tokens[1]
            content = ' '.join(tokens[2:])
            return self.file_system.write_file(path, content)

        elif command == "DELETE":
            path = tokens[1]
            return self.file_system.delete_file(path)

        else:
            return "Unknown command"

if __name__ == "__main__":
    file_system = FileSystem('./filesystem')
    server = StatefulFileServer('localhost', 12345, file_system)
    server.run()
```

StateFulClient.PY

```
import socket

class StatefulFileClient:
    def __init__(self, host, port):
        self.host = host
```

```
        self.port = port

    def run(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
sock:
            sock.connect((self.host, self.port))

            while True:
                # Get user input
                user_input = input("> ")

                # Send user input to server
                sock.sendall(user_input.encode())

                # Receive and print response from server
                response = sock.recv(1024).decode()
                print(response)

if __name__ == "__main__":
    client = StatefulFileClient('localhost', 12345)
    client.run()
```

Output :

Server:

```
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateFulServer.py
Enter portNumber: 8889
Client: 127.0.0.1:8889
Stored data for client 127.0.0.1:8889: {}
Enter portNumber: 8890
Client: 127.0.0.1:8890
Stored data for client 127.0.0.1:8890: {}
Enter portNumber: 8889
Client: 127.0.0.1:8889
Stored data for client 127.0.0.1:8889: {'CREATE': 'first.txt'}
Enter portNumber: 8890
Client: 127.0.0.1:8890
Stored data for client 127.0.0.1:8890: {'CREATE': 'demo.txt'}
```

Client1:

```
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateFulClient.py
> CREATE first.txt
File created successfully
> Traceback (most recent call last):
  File "C:\Users\Raj\Documents\SEM 8\DC\prac8\StateFulClient.py", line 25, in <module>
    client.run()
  File "C:\Users\Raj\Documents\SEM 8\DC\prac8\StateFulClient.py", line 14, in run
    user_input = input("> ")
KeyboardInterrupt
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateFulClient.py
> DELETE first.txt
File deleted successfully
>
```

Client2:

```
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateFulClient.py
> CREATE demo.txt
File created successfully
> Traceback (most recent call last):
  File "C:\Users\Raj\Documents\SEM 8\DC\prac8\StateFulClient.py", line 25, in <module>
    client.run()
  File "C:\Users\Raj\Documents\SEM 8\DC\prac8\StateFulClient.py", line 14, in run
    user_input = input("> ")
KeyboardInterrupt
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateFulClient.py
> DELETE demo.txt
File deleted successfully
>
```

STATELESS SERVER

SERVER.PY

```python
import socket
import threading
import os

class FileSystem:
    def __init__(self, root_path):
        self.root_path = root_path

    def create_file(self, path):
        full_path = os.path.join(self.root_path, path)
```

```
        with open(full_path, 'w') as f:
            f.write('')
        return "File created successfully"

    def read_file(self, path):
        full_path = os.path.join(self.root_path, path)
        with open(full_path, 'r') as f:
            content = f.read()
        return content

    def write_file(self, path, content):
        full_path = os.path.join(self.root_path, path)
        with open(full_path, 'w') as f:
            f.write(content)
        return "File written successfully"

    def delete_file(self, path):
        full_path = os.path.join(self.root_path, path)
        os.remove(full_path)
        return "File deleted successfully"

class StatefulFileServer:
    def __init__(self, host, port, file_system):
        self.host = host
        self.port = port
        self.file_system = file_system
        self.sessions = {}

    def run(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
sock:
            sock.bind((self.host, self.port))
            sock.listen()

            while True:
                conn, addr = sock.accept()
                client_thread =
threading.Thread(target=self.handle_client, args=(conn, addr))
```

```
            client_thread.start()

    def handle_client(self, conn, addr):
        # addr_1 = input("Enter portNumber: ")
        client_id = addr[0] + ":" + str(addr[1])
        print("Client: %s" % client_id)
        # Parse client request and perform file system operation
        request = conn.recv(1024).decode()
        response = self.handle_request(request)

        # Send response to client
        conn.sendall(response.encode())

        # Update session data


        conn.close()

    def handle_request(self, request):
        tokens = request.split()
        command = tokens[0]
        # session_data[command] = tokens[1]

        if command == "CREATE":
            path = tokens[1]
            return self.file_system.create_file(path)

        elif command == "READ":
            path = tokens[1]
            return self.file_system.read_file(path)

        elif command == "WRITE":
            path = tokens[1]
            content = ' '.join(tokens[2:])
            return self.file_system.write_file(path, content)

        elif command == "DELETE":
            path = tokens[1]
```

```
            return self.file_system.delete_file(path)

        else:
            return "Unknown command"

if __name__ == "__main__":
    file_system = FileSystem('./filesystem')
    server = StatefulFileServer('localhost', 12345, file_system)
    server.run()
```

StateLessClient.py:

```
import socket

class StatefulFileClient:
    def __init__(self, host, port):
        self.host = host
        self.port = port

    def run(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
sock:
            sock.connect((self.host, self.port))

            while True:
                # Get user input
                user_input = input("> ")

                # Send user input to server
                sock.sendall(user_input.encode())

                # Receive and print response from server
                response = sock.recv(1024).decode()
                print(response)

if __name__ == "__main__":
    client = StatefulFileClient('localhost', 12345)
    client.run()
```

Output :

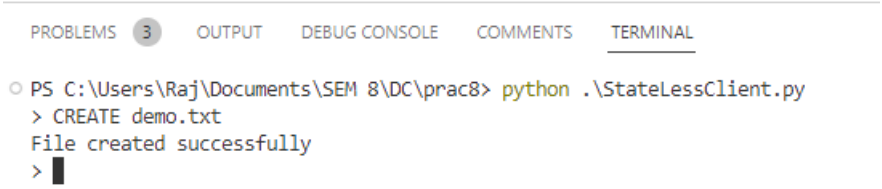
Server

```
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateLessServer.py
Client: 127.0.0.1:65508
Client: 127.0.0.1:65524
Client: 127.0.0.1:49159
```

Client 1

```
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateLessClient.py
> CREATE req.txt
File created successfully
> Traceback (most recent call last):
  File "C:\Users\Raj\Documents\SEM 8\DC\prac8\StateLessClient.py", line 25, in <module>
    client.run()
  File "C:\Users\Raj\Documents\SEM 8\DC\prac8\StateLessClient.py", line 14, in run
    user_input = input("> ")
KeyboardInterrupt
PS C:\Users\Raj\Documents\SEM 8\DC\prac8> python .\StateLessClient.py
>
```

Client2



Conclusions :

1. Implemented Stateful and Stateless Server
2. Compared the performance of stateful and stateless servers in handling client requests.
3. The response times of the stateless server were consistently faster and more stable compared to the stateful server.

4. Since the stateful server needs to keep track of client sessions, it requires more resources and processing power, which can result in slower response times and higher variability.
5. It is important to consider the requirements of the application and choose the appropriate server architecture based on those requirements.

Postlab Questions:

1. Compare Stateful and stateless servers
2. Explain: 'Exactly Once' call semantics

LAB 10

Aim: To Study HDFS and MapReduce

Lab Outcome:

Describe the concepts of distributed File Systems with some case studies

Theory:

Hadoop:

With growing data velocity, the data size easily outgrows the storage limit of a machine. A solution would be to store the data across a network of machines. Such filesystems are called *distributed filesystems*. Since data is stored across a network all the complications of a network come in.
This is where Hadoop comes in. It provides one of the most reliable filesystems. HDFS (Hadoop Distributed File System) is a unique design that provides storage for *extremely large files* with streaming data access pattern and it runs on *commodity hardware*.

Let's elaborate the terms:

- ***Extremely large files***: Here we are talking about the data in range of petabytes (1000 TB).
- ***Streaming Data Access Pattern***: HDFS is designed on principle of *write-once and read-many-times*. Once data is written large portions of dataset can be processed any number times.
- ***Commodity hardware:*** Hardware that is inexpensive and easily available in the market. This is one of feature which specially distinguishes HDFS from other file system.

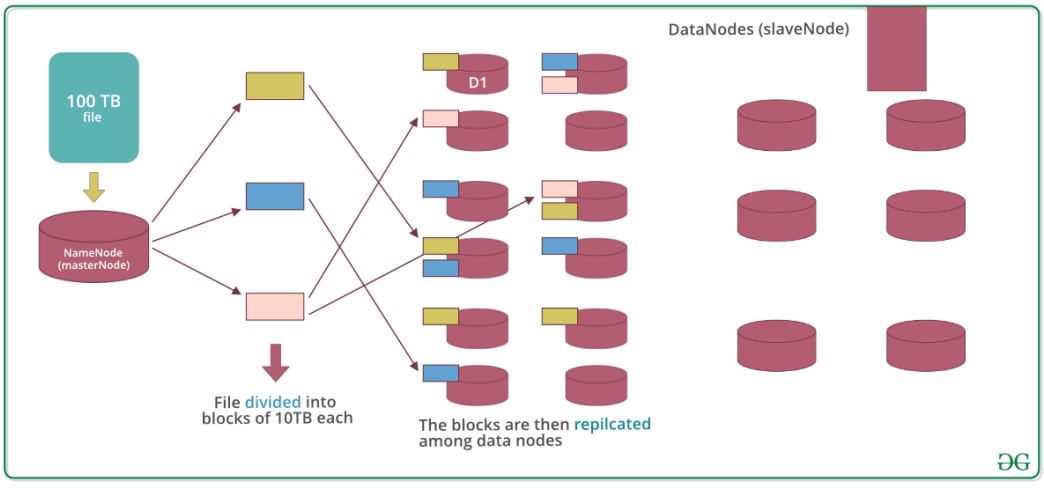
Nodes: Master-slave nodes typically forms the HDFS cluster.

1. **NameNode(MasterNode):**
    - Manages all the slave nodes and assign work to them.
    - It executes filesystem namespace operations like opening, closing, renaming files and directories.
    - It should be deployed on reliable hardware which has the high config. not on commodity hardware.
2. **DataNode(SlaveNode):**
    - Actual worker nodes, who do the actual work like reading, writing, processing etc.
    - They also perform creation, deletion, and replication upon instruction from the master.
    - They can be deployed on commodity hardware.

HDFS daemons: Daemons are the processes running in background.

- **Namenodes:**
    - Run on the master node.
    - Store metadata (data about data) like file path, the number of blocks, block Ids. etc.
    - Require high amount of RAM.
    - Store meta-data in RAM for fast retrieval i.e to reduce seek time. Though a persistent copy of it is kept on disk.
- **DataNodes:**
    - Run on slave nodes.
    - Require high memory as data is actually stored here.

Data storage in HDFS: Now let us see how the data is stored in a distributed manner.



Assuming that 100TB file is inserted, then masternode(namenode) will first divide the file into blocks of 10TB (default size is 128 MB in Hadoop 2.x and above). Then these blocks are stored across different datanodes(slavenode). Datanodes(slavenode)replicate the blocks among themselves and the information of what blocks they contain is sent to the master. Default replication factor is 3 means for each block 3 replicas are created (including itself). In hdfs.site.xml we can increase or decrease the replication factor i.e we can edit its configuration here.

Terms related to HDFS:

- ***HeartBeat***: It is the signal that datanode continuously sends to namenode. If namenode doesn't receive heartbeat from a datanode then it will consider it dead.
- ***Balancing***: If a datanode is crashed the blocks present on it will be gone too and the blocks will be *under-replicated* compared to the remaining blocks. Here master node(namenode) will give a signal to datanodes containing replicas of those lost blocks to replicate so that overall distribution of blocks is balanced.
- ***Replication:*** It is done by datanode.

Features:

- Distributed data storage.
- Blocks reduce seek time.
- The data is highly available as the same block is present at multiple datanodes.
- Even if multiple datanodes are down we can still do our work, thus making it highly reliable.
- High fault tolerance.

MapReduce:

MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

MapReduce Architecture explained in detail:

- One map task is created for each split which then executes map function for each record in the split.

- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.
- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

- Map tasks (Splits & Mapping)
- Reduce tasks (Shuffling, Reducing)

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

- Jobtracker: Acts like a master (responsible for complete execution of submitted job)
- Multiple Task Trackers: Acts like slaves, each of them performing the job

For every job submitted for execution in the system, there is one Jobtracker that resides on Namenode and there are multiple tasktrackers which reside on Datanode.

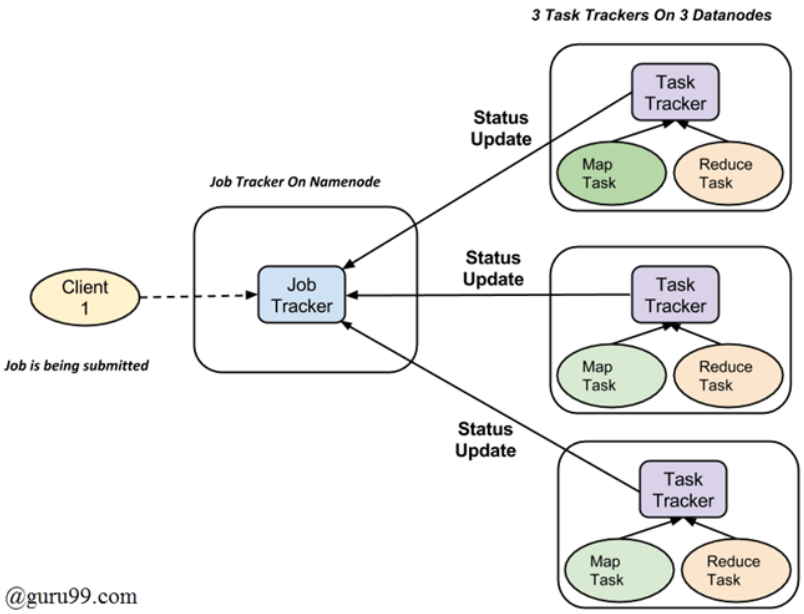


Fig. How Hadoop Mapreduce Works

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends 'heartbeat' signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

Implementation of Hadoop Cluster and HDFS:

To demonstrate the working and configuration of HDFS, a cluster consisting of 1 Master and 2 Slaves is configured. The machine nodes are independent AWS ec2 instances. For ease of operation a tool called MobaXterm is used for remote login (SSH) into the instances.

After successful configuration, the daemons are started in both master and slave nodes. Subsequently, our HDFS can be tested by storing files.

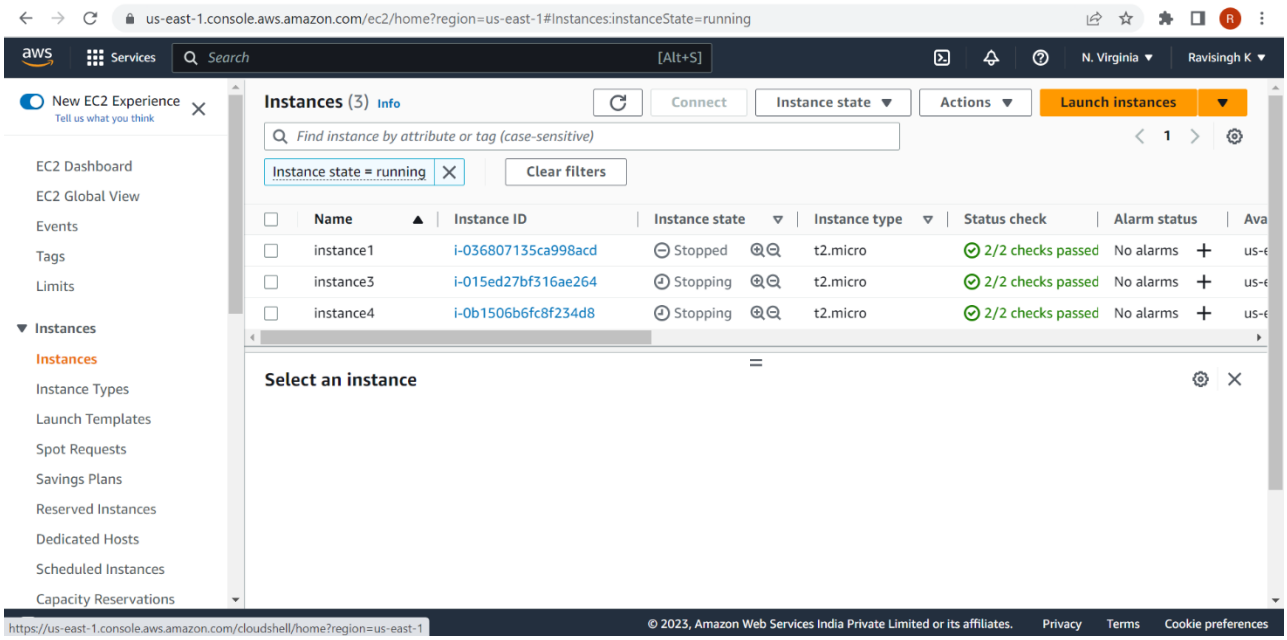
Step 1: Setting up Remote Machines

Three ec2 instances having Ubuntu as local OS were created using AWS Free Tier account.

Instance1 -> Master Private IP: 172.31.87.194

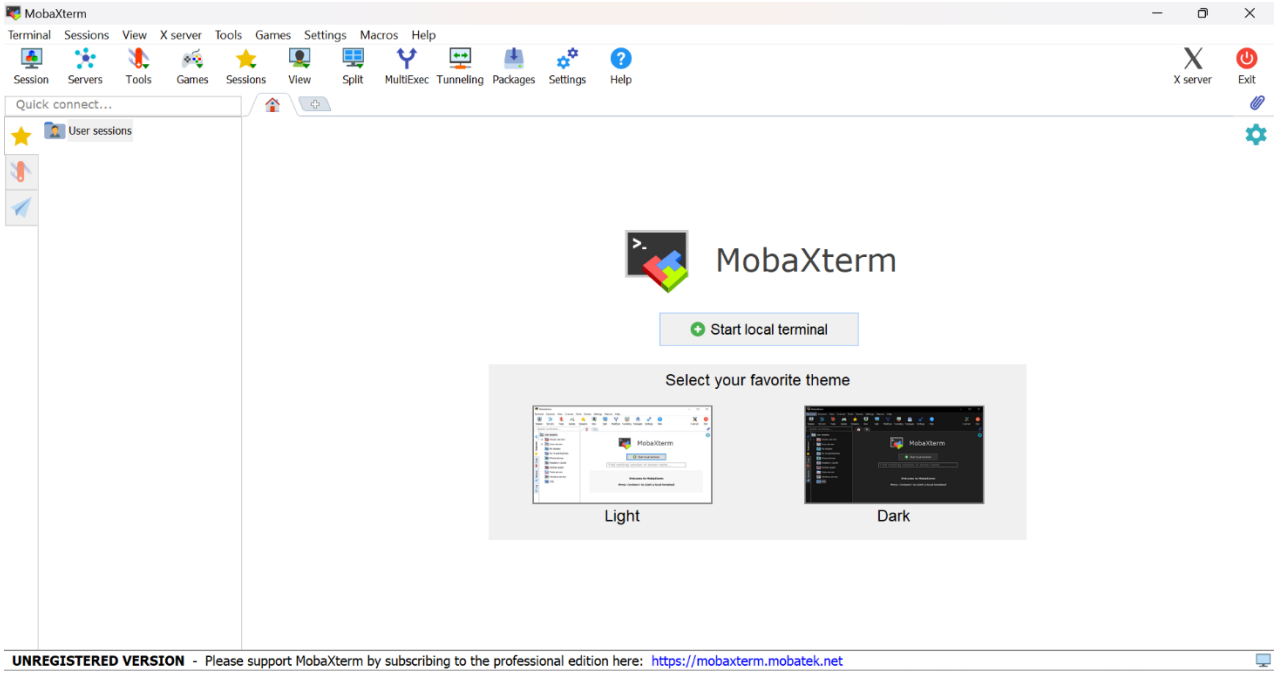
Instance3 -> Slave1 Private IP: 172.31.92.80

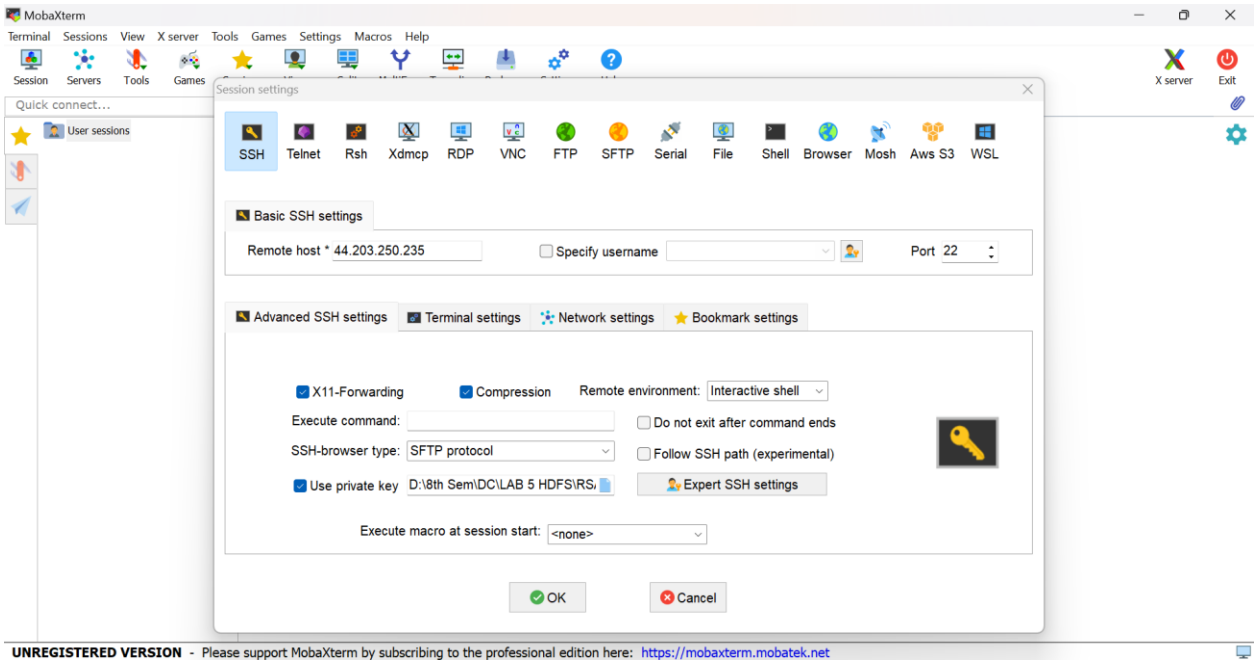
Instance4 -> Slave2 Private IP: 172.31.28.27



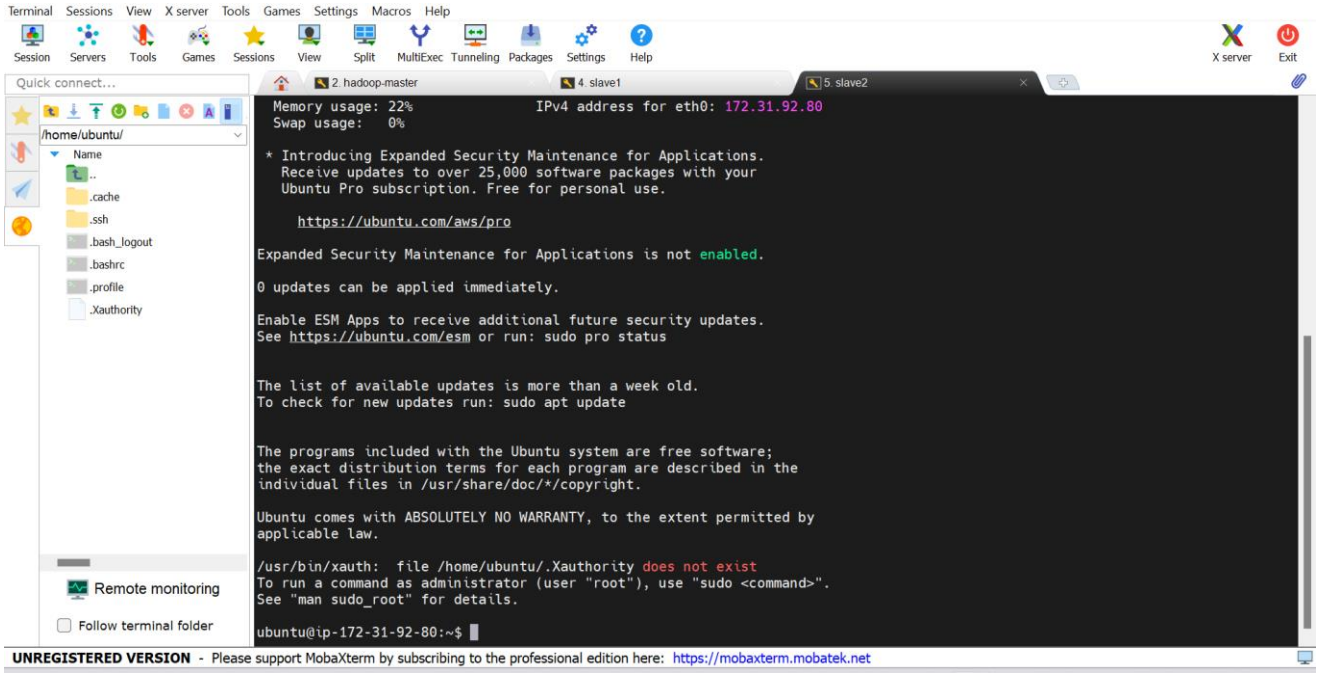
Step 2: Setting up SSH Client

MobaXterm is used an SSH Client to login to remote machines





After connecting to all our nodes, the environment looks like this. SSH connections to our remote machines are lined up as tabs

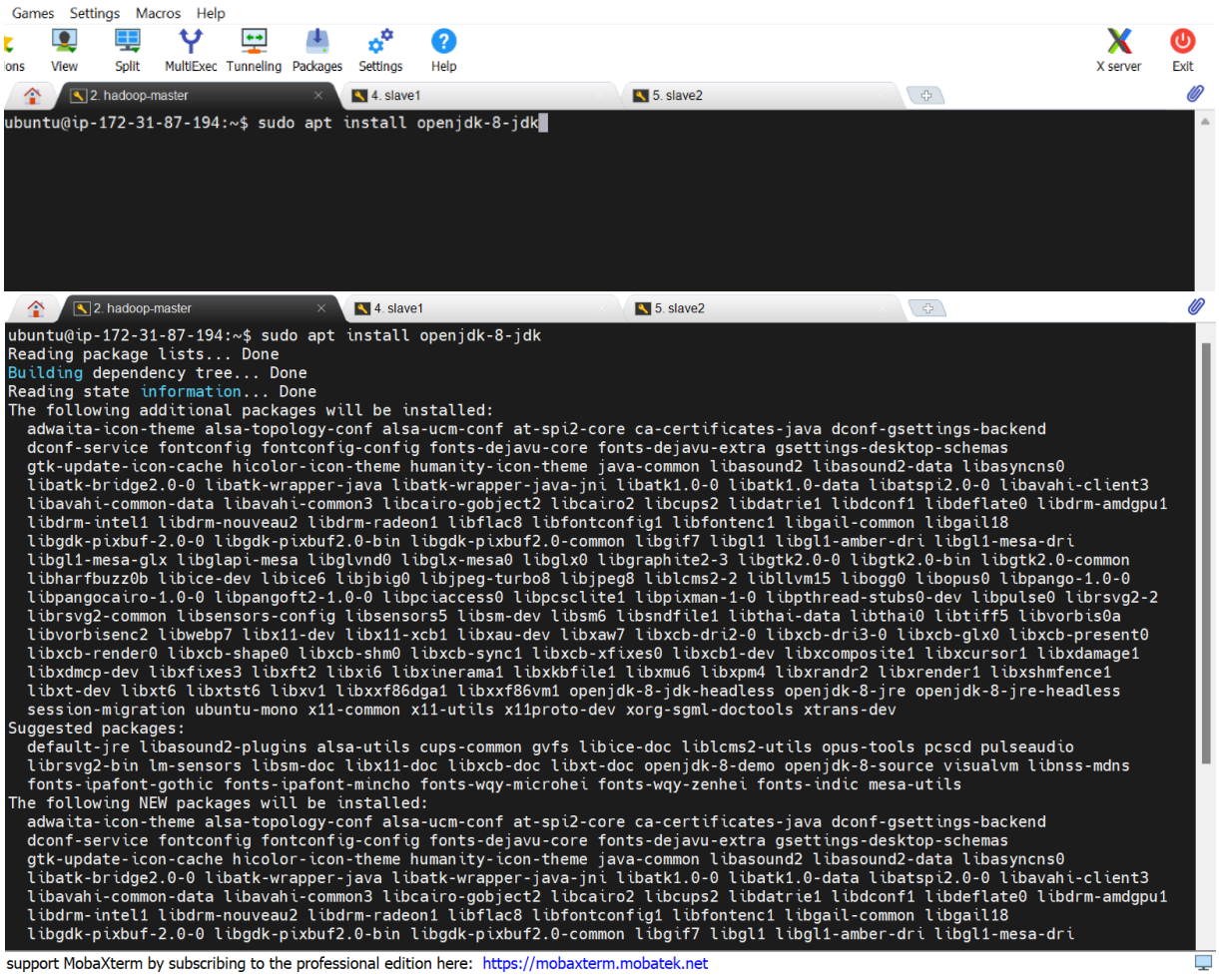


Step 3: Configuration of Hadoop Cluster

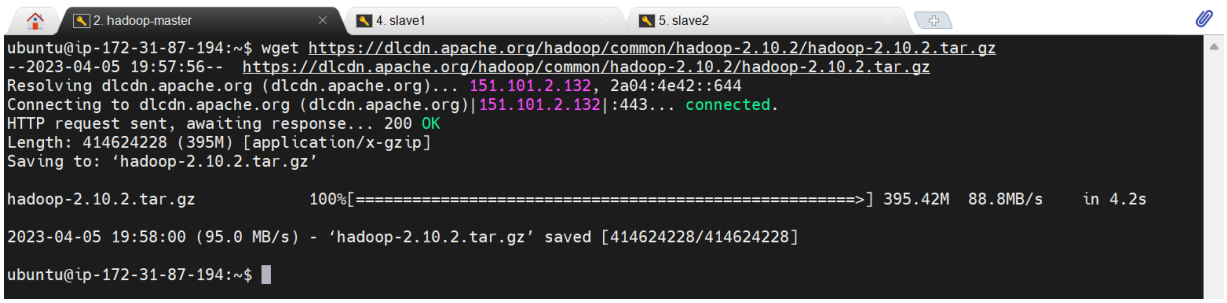
Step 3.1: Download Hadoop and JDK in all three machines

The step is repeated for all the 3 nodes

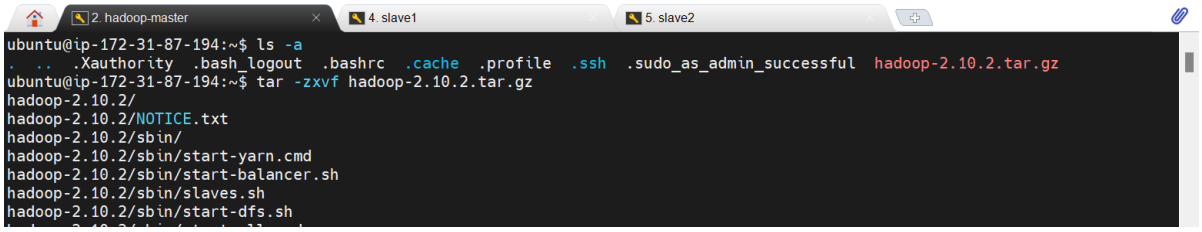
Download jdk-8



Download Hadoop



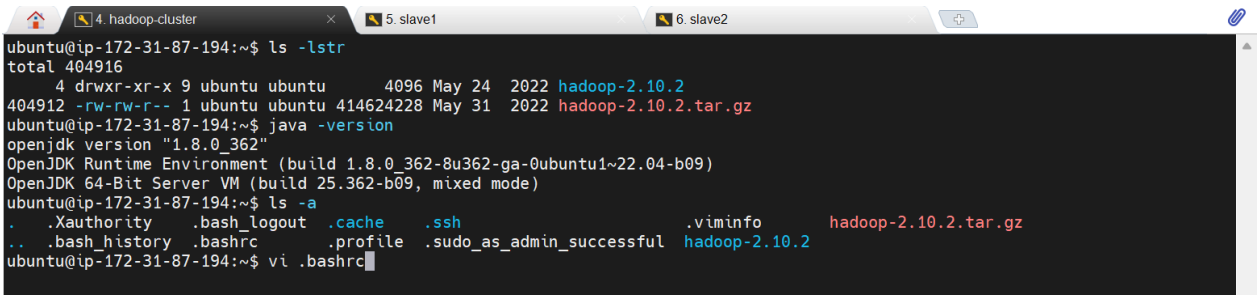
Extracting Hadoop

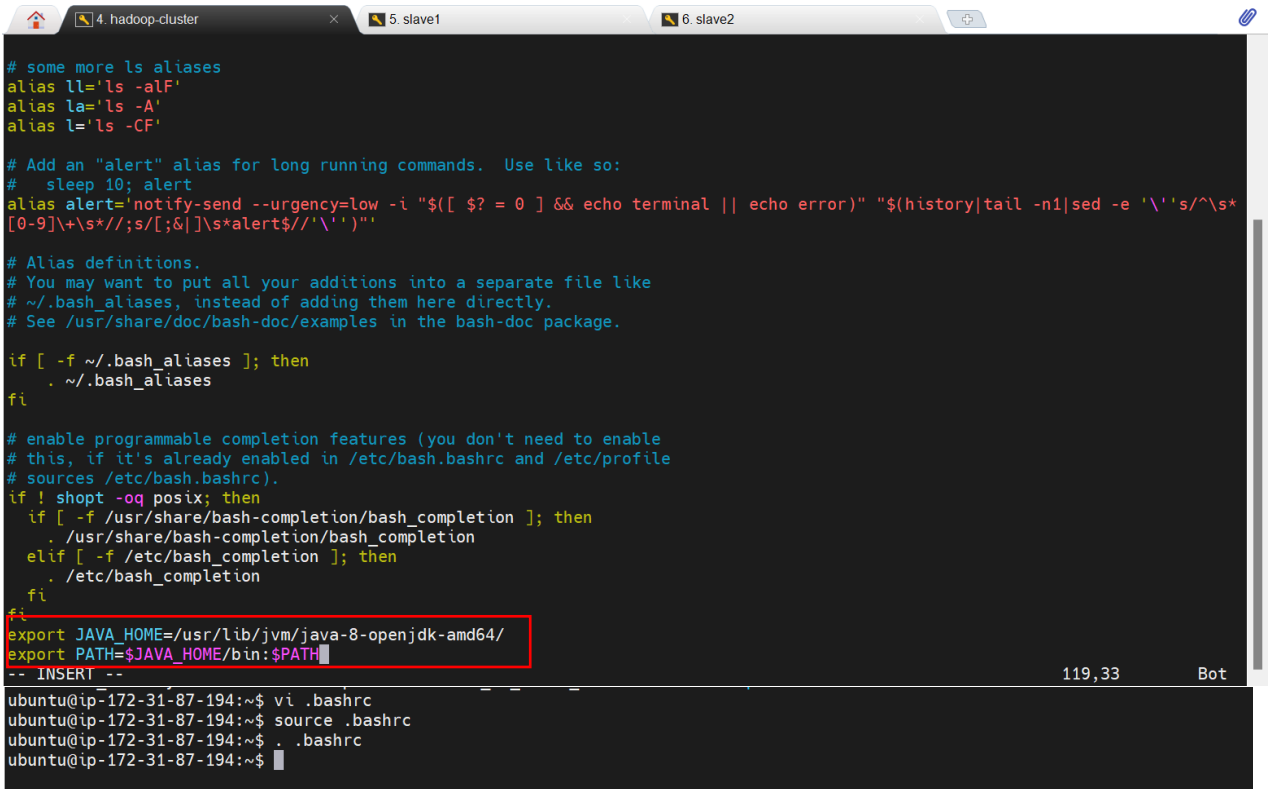


Step 3.2: Environment Setup for java

This step is repeated for all three nodes

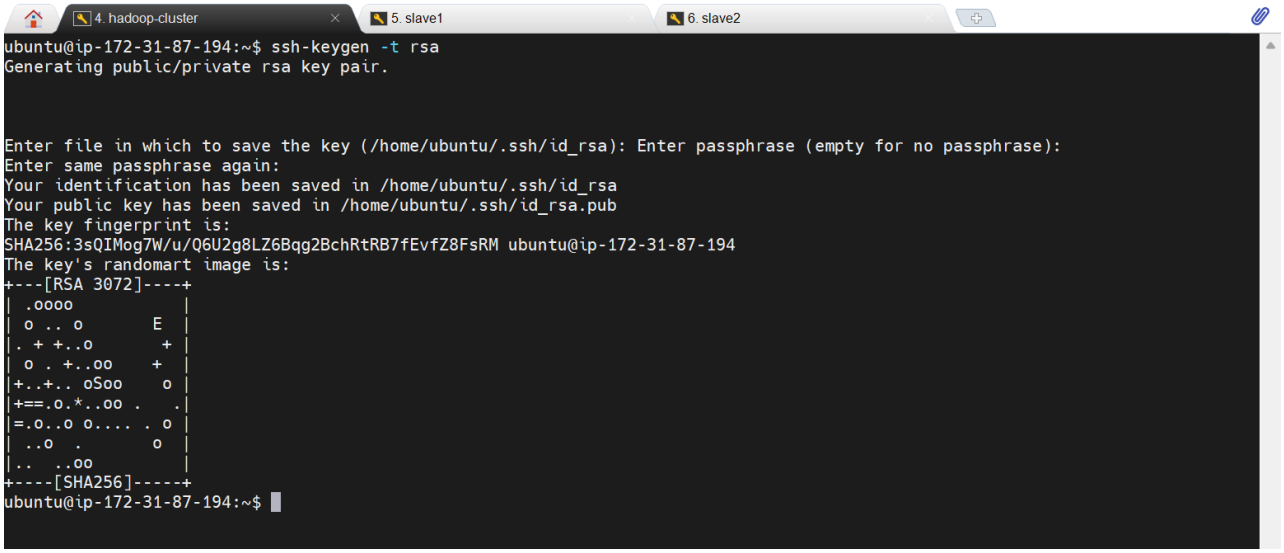
Adding JDK Path in .bashrc file and executing it

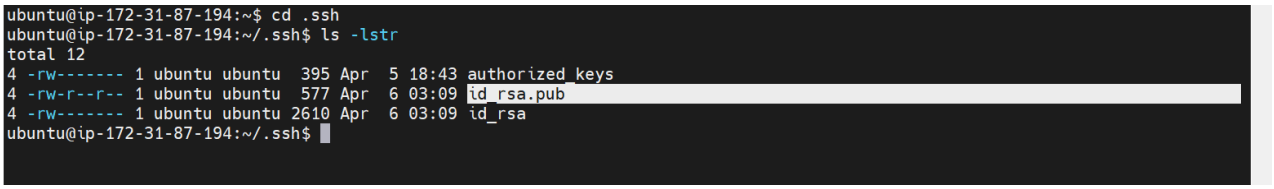


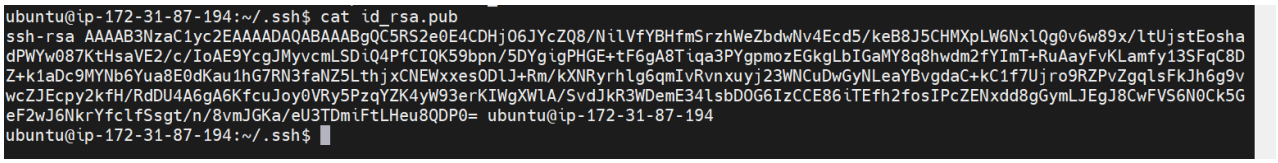


Step 3.3: SSH Setup Between Master and Slaves

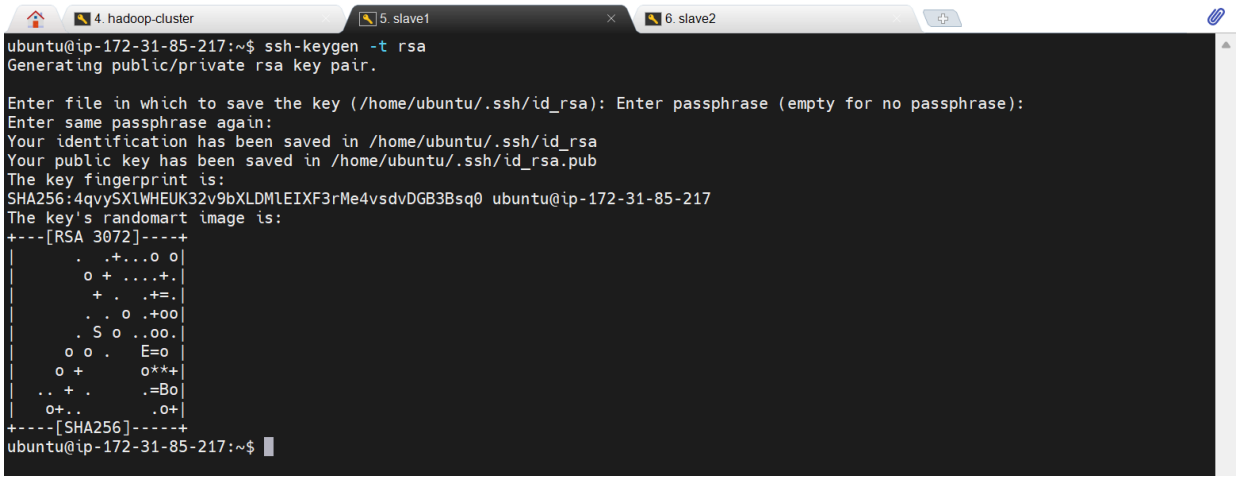
- Generate Key Pairs on each node using: ssh-keygen -t rsa. The generated pairs are stored in id_rsa.pub in .ssh folder
- Key Pairs from all the nodes are copied together and pasted in authorized_keys file present in .ssh folder at every node.

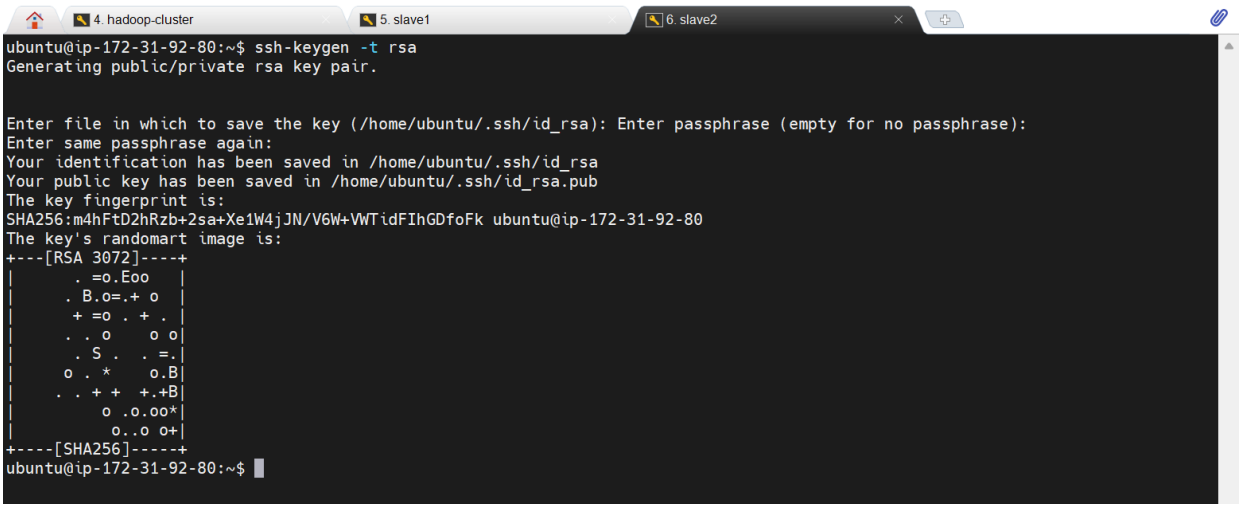




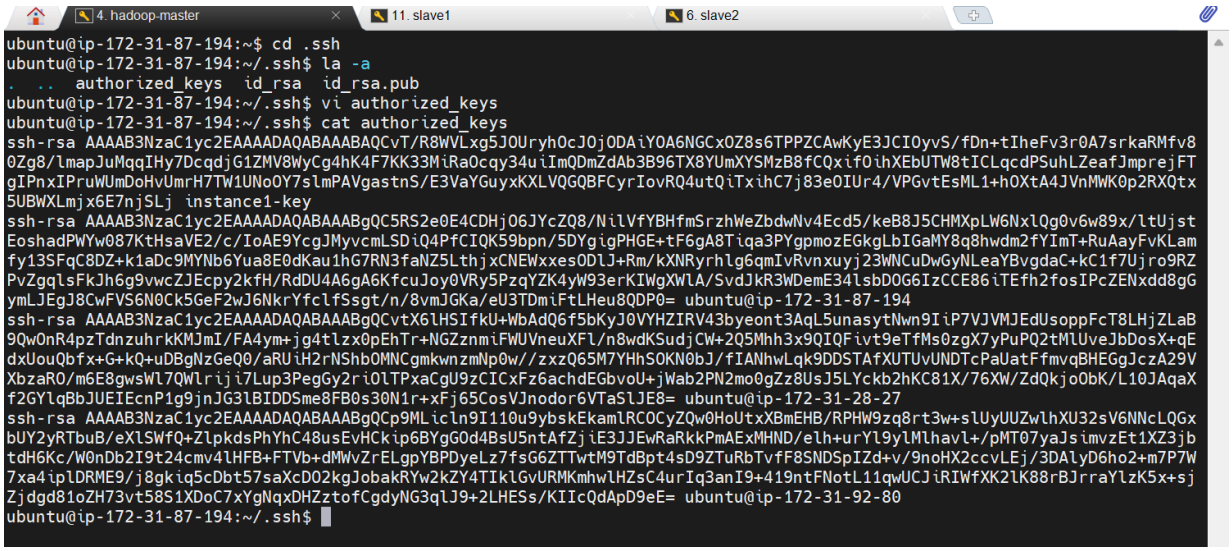


Key Pair generation at slave nodes



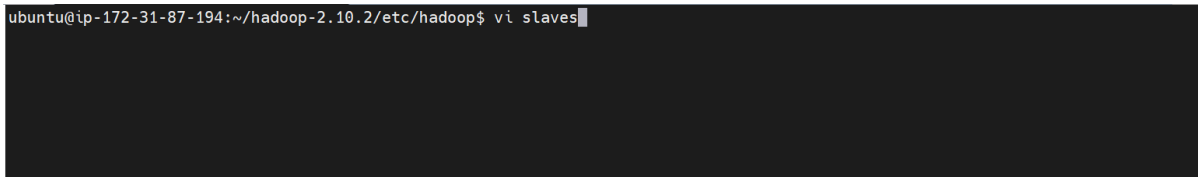


Final authorized_keys file: This file needs to be maintained at every node



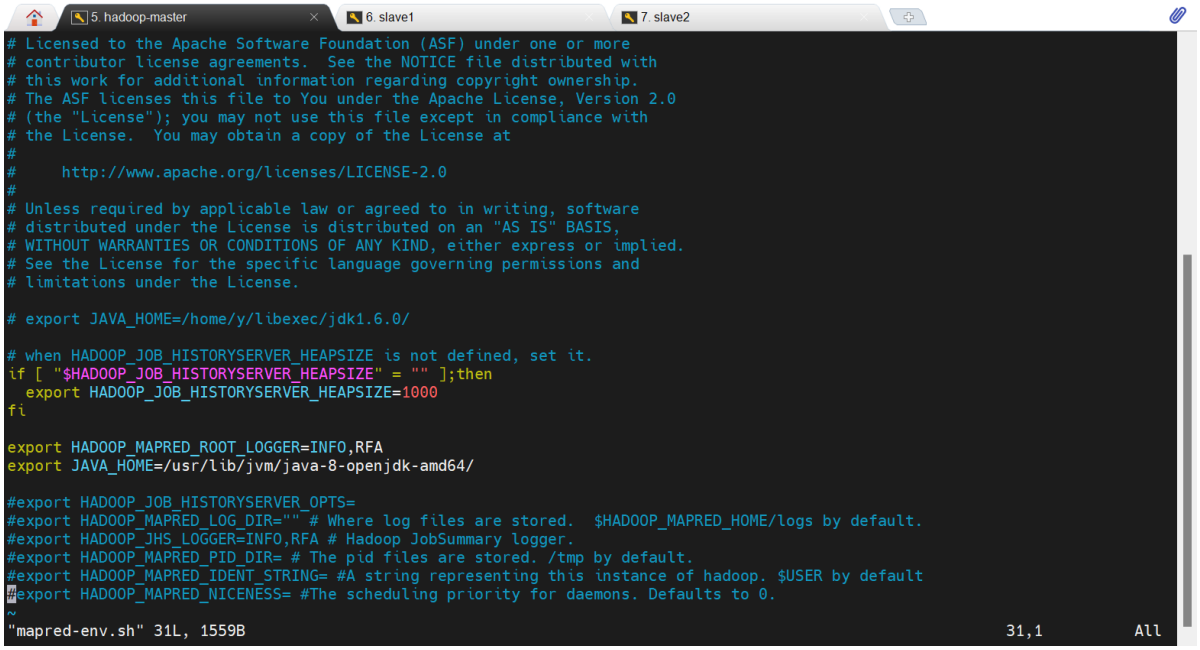
Step 3.4: Configuration of Hadoop Files. Files present in path: hadoop-2.10.2/etc/Hadoop

Configuration of slaves files (Only at Master Node): Add Private IPs of Slaves in slaves file present in master node

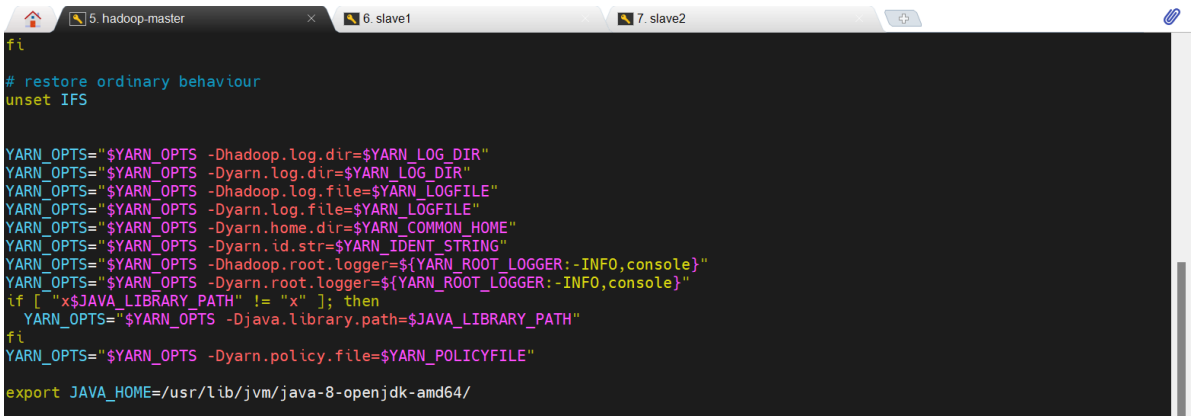


Configuration of env files (Needs to be repeated for every node): Adding JDK path to each file

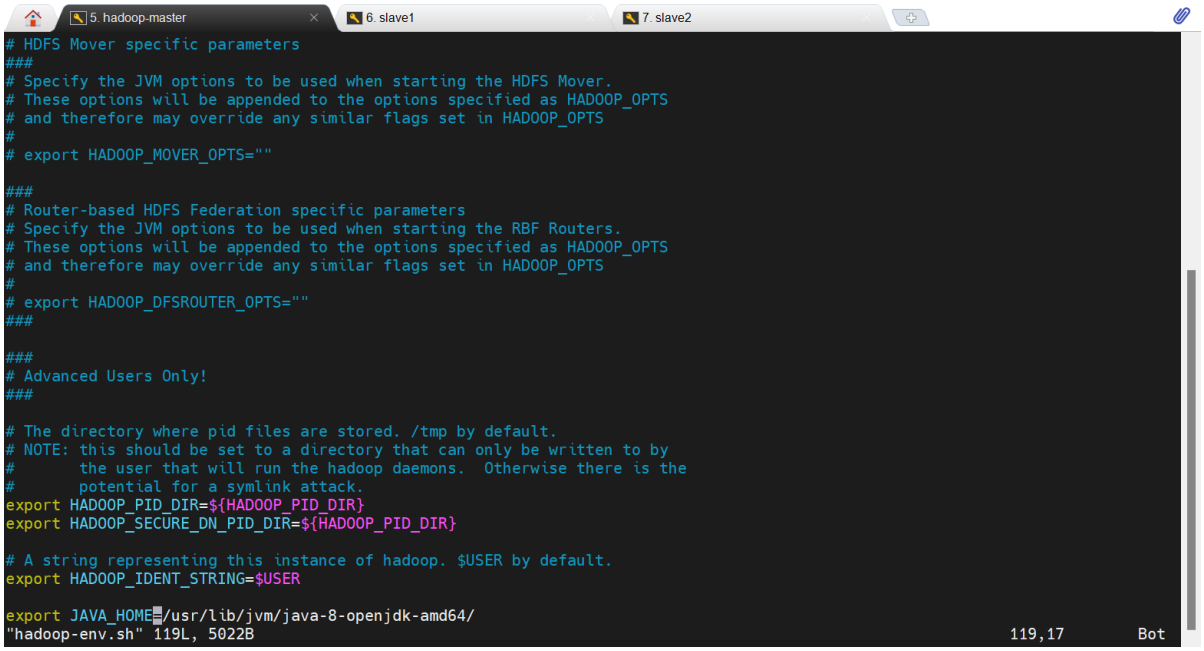
mapred-env.sh



yarn-env.sh



hadoop-env.sh



Configuration of XML Files:

core-site.xml: Same for master as slaves

The IP address provided is the private IP address of the Master Node

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://172.31.87.194:50000</value>
</property>
</configuration>
~
~
```

yarn-site.xml: Same for master and slaves

The IP address provided is the private IP address of the Master Node

```
<property>
<name>yarn.nodemanager.aux-services</name> <value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
<description>The hostname of the RM.</description>
<name>yarn.resourcemanager.hostname</name>
<value>172.31.87.194</value>
</property>
<property>
<description>The address of the applications manager interface in the RM.</description>
<name>yarn.resourcemanager.address</name>
<value>172.31.87.194:8032</value>
</property>
```

hdfs-site.xml:

Mater Node:

```
<configuration>
<property>
<name>dfs.namenode.name.dir</name>
<value>/home/ubuntu/hadoop2-dir/namenode-dir</value>
</property>
</configuration>
~
```

Slave Nodes:

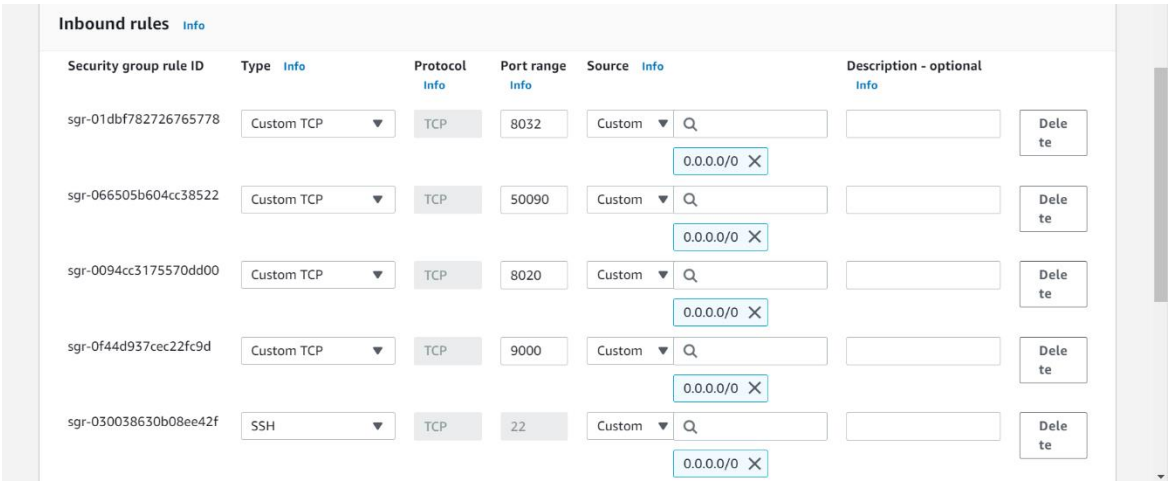
```
<configuration>

<property>
<name>dfs.datanode.data.dir</name>
<value>/home/ubuntu/hadoop2-dir/datanode-dir</value>
</property>

</configuration>
```

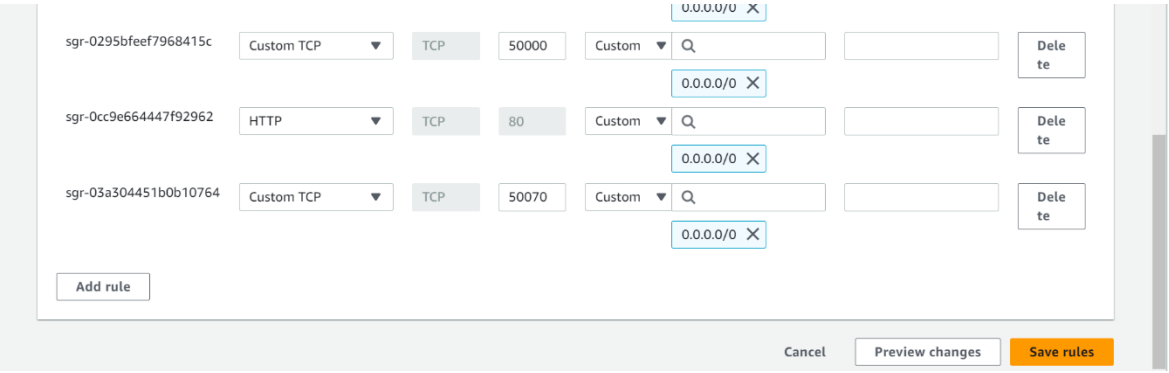
mapred-site.xml: Same for master and slaves

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

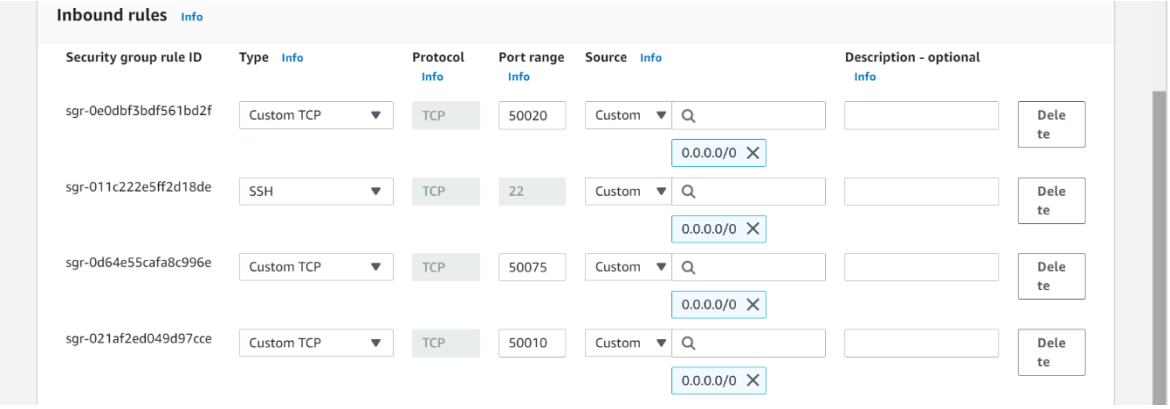
Step 4: Adding inbound rules in security groups of instances

Master Instance:



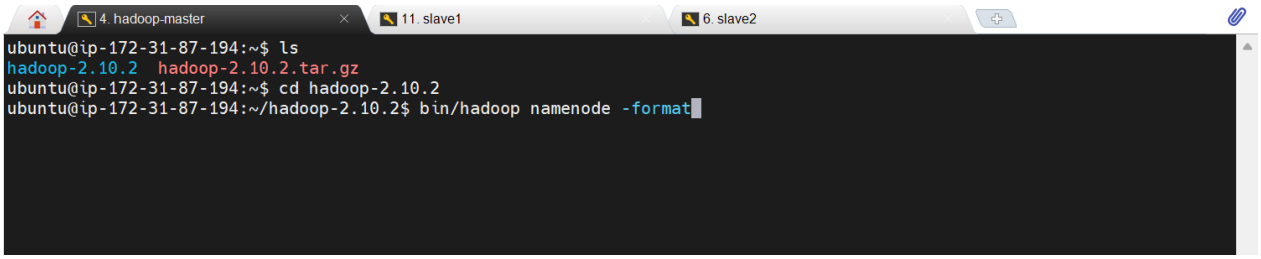


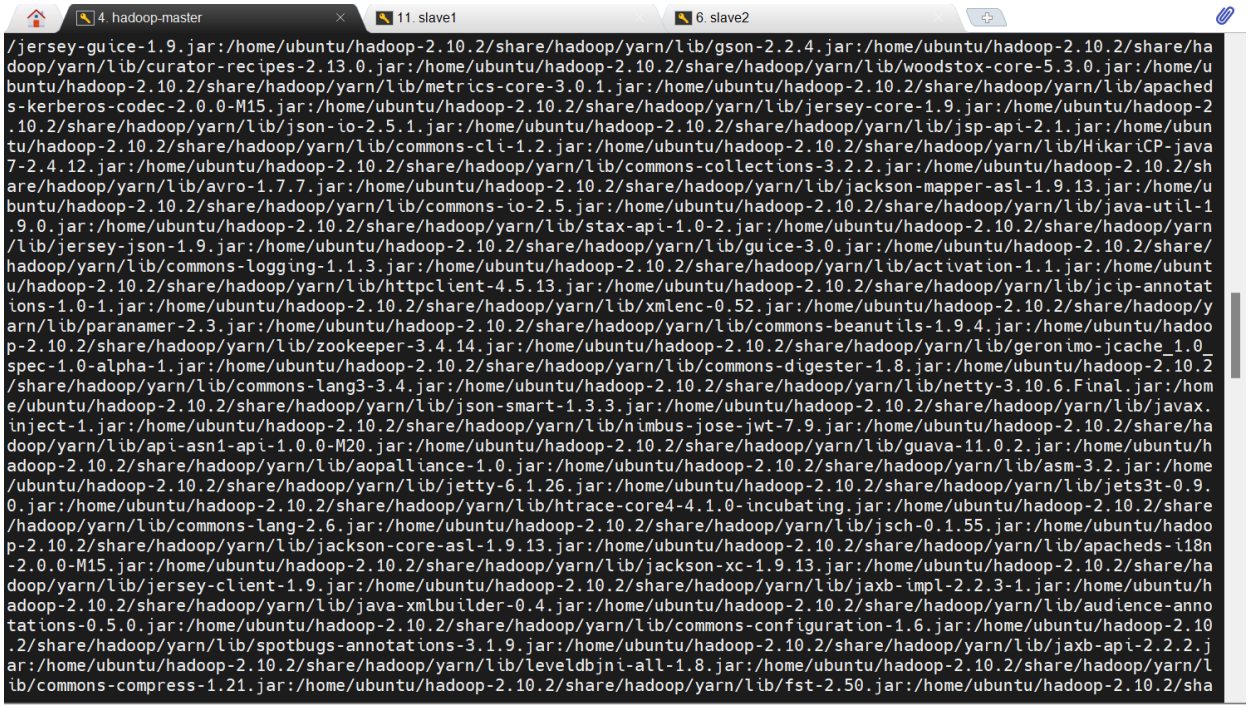
Slave Instances:

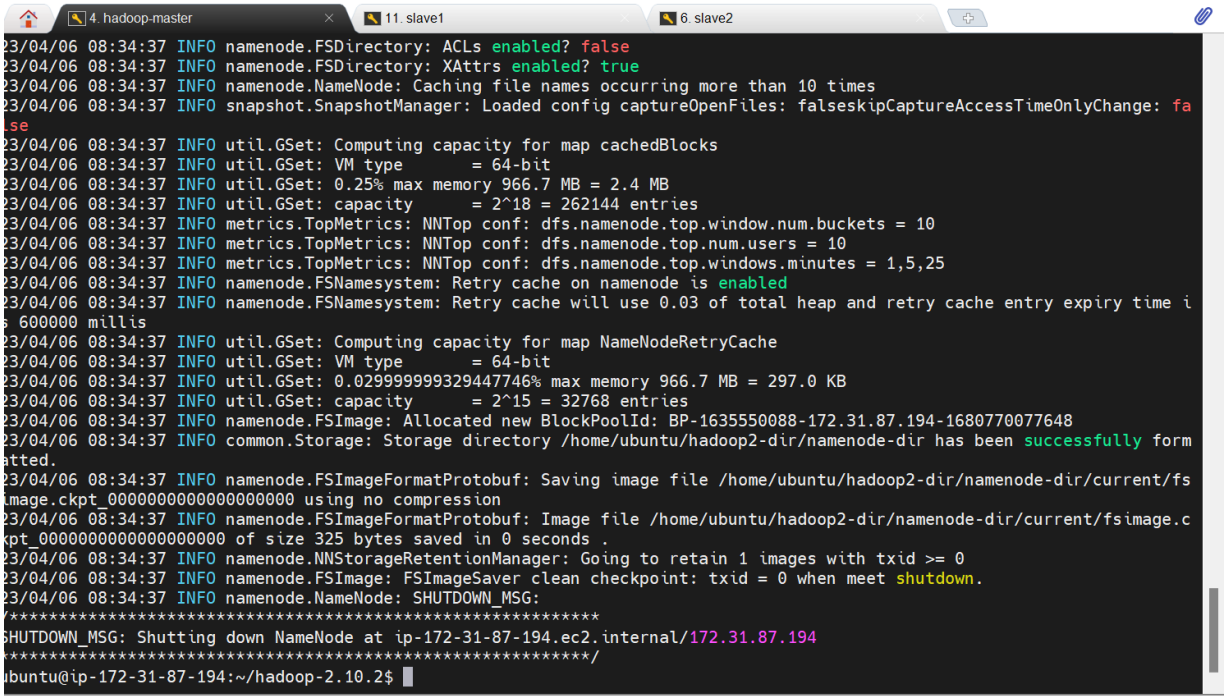


Step 5: Formatting of Hadoop Cluster and Starting of the daemons

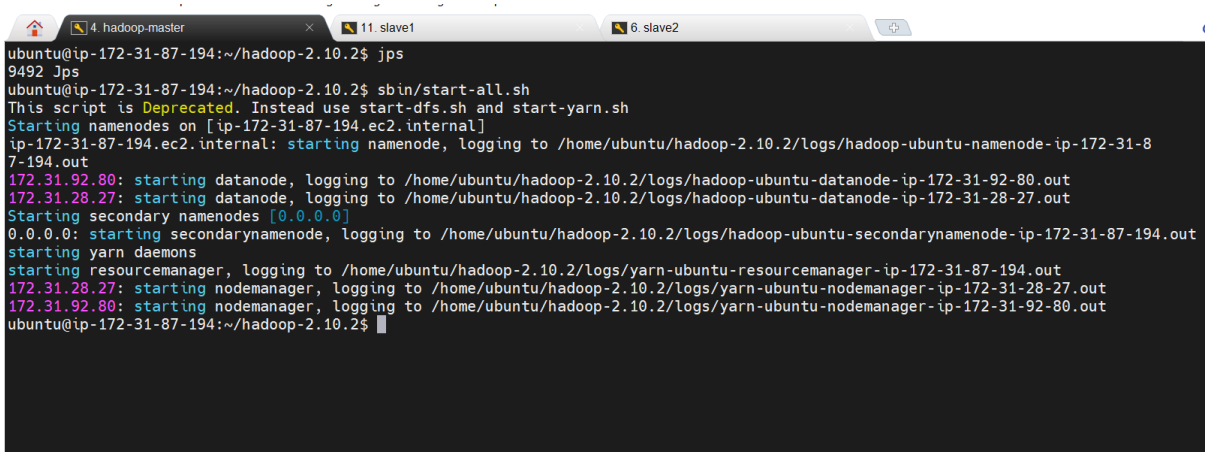
Format Hadoop Cluster: Command is executed only at the Master Node




4. hadoop-master
11. slave1
6. slave2
/jersey-guice-1.9.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/gson-2.2.4.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/curator-recipes-2.13.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/woodstox-core-5.3.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/metrics-core-3.0.1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/apacheds-kerberos-codec-2.0.0-M15.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jersey-core-1.9.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/json-io-2.5.1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jsp-api-2.1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-cli-1.2.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/HikariCP-java7-2.4.12.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-collections-3.2.2.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/avro-1.7.7.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jackson-mapper-asl-1.9.13.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-io-2.5.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/java-util-1.9.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/stax-api-1.0-2.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jersey-json-1.9.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/guice-3.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-logging-1.1.3.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/activation-1.1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/httpclient-4.5.13.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jcip-annotations-1.0-1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/xmlenc-0.52.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/paranamer-2.3.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-beanutils-1.9.4.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/zookeeper-3.4.14.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/geronimo-jcache_1.0_spec-1.0-alpha-1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-digester-1.8.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-lang3-3.4.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/netty-3.10.6.Final.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/json-smart-1.3.3.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/javax.inject-1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/nimbus-jose-jwt-7.9.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/api-asn1-api-1.0.0-M20.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/guava-11.0.2.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/aopalliance-1.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/asm-3.2.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jetty-6.1.26.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jets3t-0.9.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/htrace-core4-4.1.0-incubating.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-lang-2.6.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jsch-0.1.55.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jackson-core-asl-1.9.13.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/apacheds-i18n-2.0.0-M15.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jackson-xc-1.9.13.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jersey-client-1.9.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jaxb-impl-2.2.3-1.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/java-xmlbuilder-0.4.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/audience-annotations-0.5.0.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-configuration-1.6.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/spotbugs-annotations-3.1.9.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/jaxb-api-2.2.2.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/leveldbjni-all-1.8.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/commons-compress-1.21.jar:/home/ubuntu/hadoop-2.10.2/share/hadoop/yarn/lib/fst-2.50.jar:/home/ubuntu/hadoop-2.10.2/sha


4. hadoop-master
11. slave1
6. slave2
23/04/06 08:34:37 INFO namenode.FSDirectory: ACLs enabled? false
23/04/06 08:34:37 INFO namenode.FSDirectory: XAttrs enabled? true
23/04/06 08:34:37 INFO namenode.NameNode: Caching file names occurring more than 10 times
23/04/06 08:34:37 INFO snapshot.SnapshotManager: Loaded config captureOpenFiles: falseskipCaptureAccessTimeOnlyChange: false
23/04/06 08:34:37 INFO util.GSet: Computing capacity for map cachedBlocks
23/04/06 08:34:37 INFO util.GSet: VM type = 64-bit
23/04/06 08:34:37 INFO util.GSet: 0.25% max memory 966.7 MB = 2.4 MB
23/04/06 08:34:37 INFO util.GSet: capacity = 2^18 = 262144 entries
23/04/06 08:34:37 INFO metrics.TopMetrics: NNTop conf: dfs.namenode.top.window.num.buckets = 10
23/04/06 08:34:37 INFO metrics.TopMetrics: NNTop conf: dfs.namenode.top.num.users = 10
23/04/06 08:34:37 INFO metrics.TopMetrics: NNTop conf: dfs.namenode.top.windows.minutes = 1,5,25
23/04/06 08:34:37 INFO namenode.FSNamesystem: Retry cache on namenode is enabled
23/04/06 08:34:37 INFO namenode.FSNamesystem: Retry cache will use 0.03 of total heap and retry cache entry expiry time is 600000 millis
23/04/06 08:34:37 INFO util.GSet: Computing capacity for map NameNodeRetryCache
23/04/06 08:34:37 INFO util.GSet: VM type = 64-bit
23/04/06 08:34:37 INFO util.GSet: 0.029999999329447746% max memory 966.7 MB = 297.0 KB
23/04/06 08:34:37 INFO util.GSet: capacity = 2^15 = 32768 entries
23/04/06 08:34:37 INFO namenode.FSImage: Allocated new BlockPoolId: BP-1635550088-172.31.87.194-1680770077648
23/04/06 08:34:37 INFO common.Storage: Storage directory /home/ubuntu/hadoop2-dir/namenode-dir has been successfully formatted.
23/04/06 08:34:37 INFO namenode.FSImageFormatProtobuf: Saving image file /home/ubuntu/hadoop2-dir/namenode-dir/current/fsimage.ckpt_0000000000000000000 using no compression
23/04/06 08:34:37 INFO namenode.FSImageFormatProtobuf: Image file /home/ubuntu/hadoop2-dir/namenode-dir/current/fsimage.ckpt_0000000000000000000 of size 325 bytes saved in 0 seconds .
23/04/06 08:34:37 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with txid >= 0
23/04/06 08:34:37 INFO namenode.FSImage: FSImageSaver clean checkpoint: txid = 0 when meet shutdown.
23/04/06 08:34:37 INFO namenode.NameNode: SHUTDOWN_MSG:
/**
SHUTDOWN_MSG: Shutting down NameNode at ip-172-31-87-194.ec2.internal/172.31.87.194
**/
ubuntu@ip-172-31-87-194:~/hadoop-2.10.2$

Starting of daemons: Command Executed only at Master Node



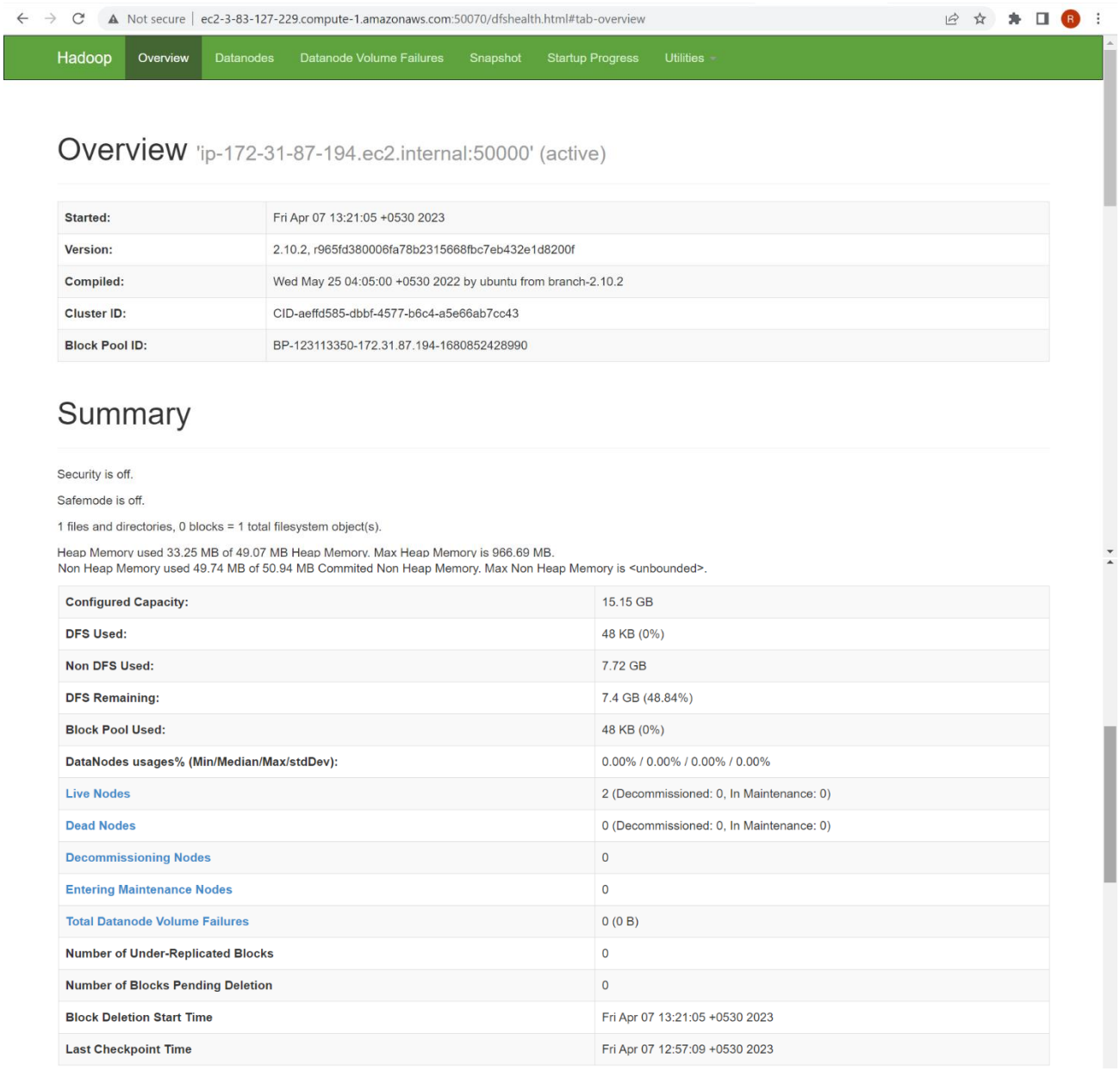
Daemons running at the Master:

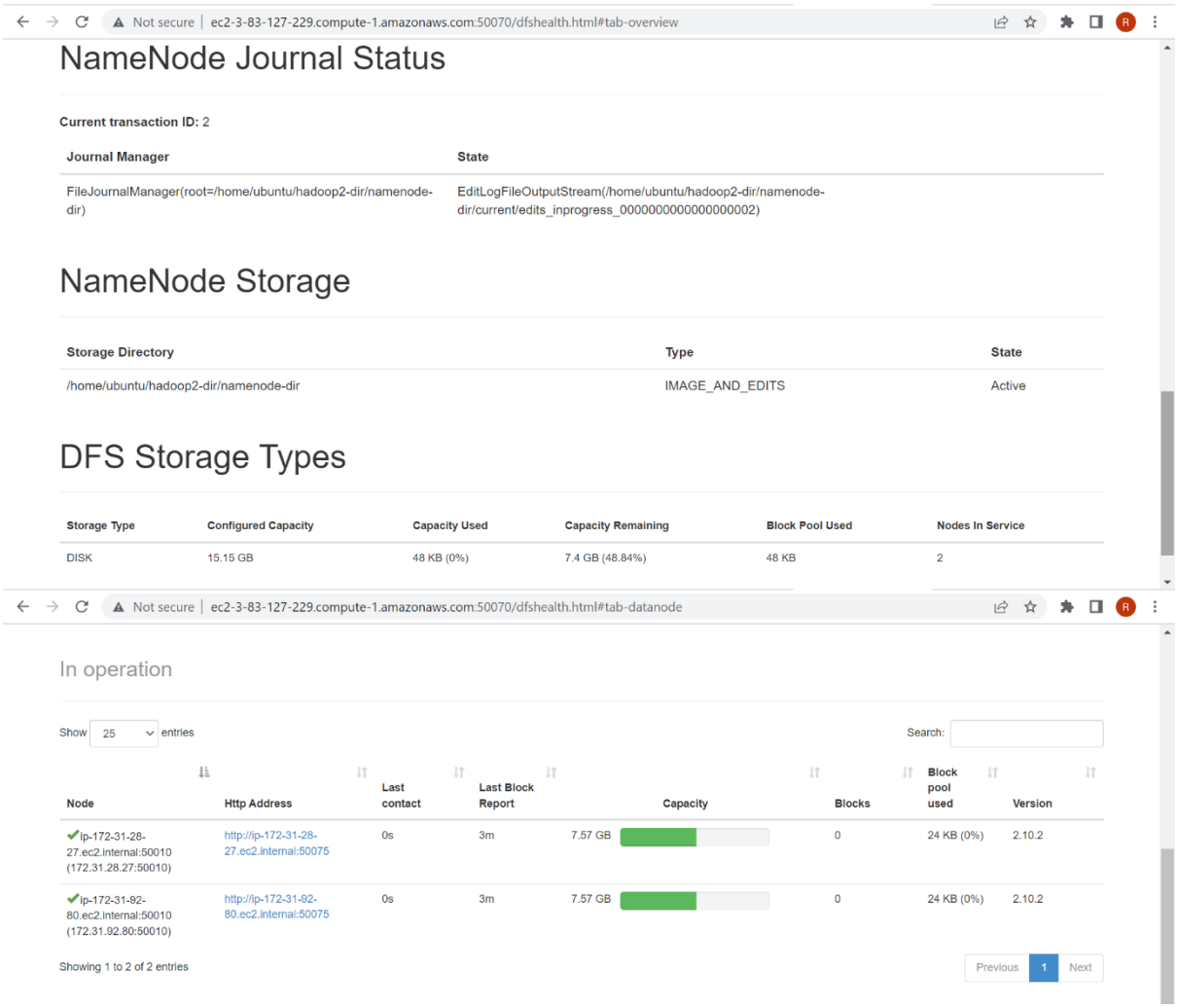
Daemons running at slave 1:

Daemons running at slave 2:

Step 6: Launching the Namenode Web Interface

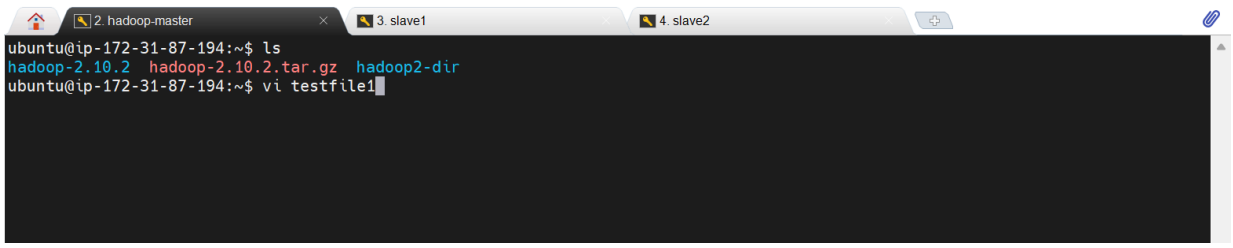
The Web UI Provided by Apache, provides the following information about the cluster:

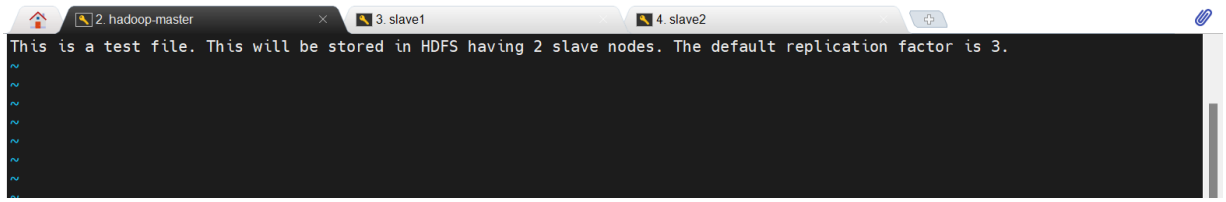




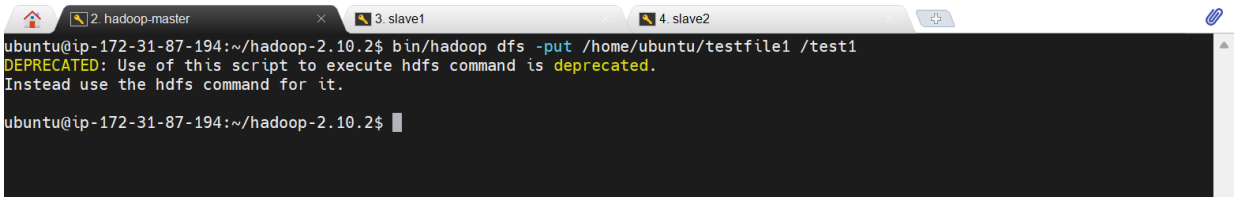
Step 7: Testing HDFS by storing files

<u>Creating a Test File</u>

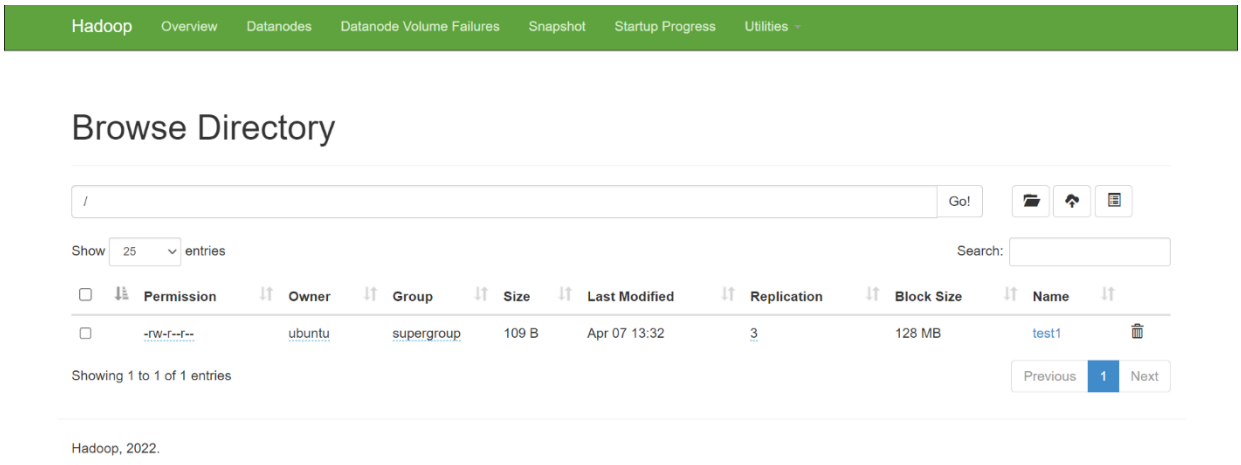


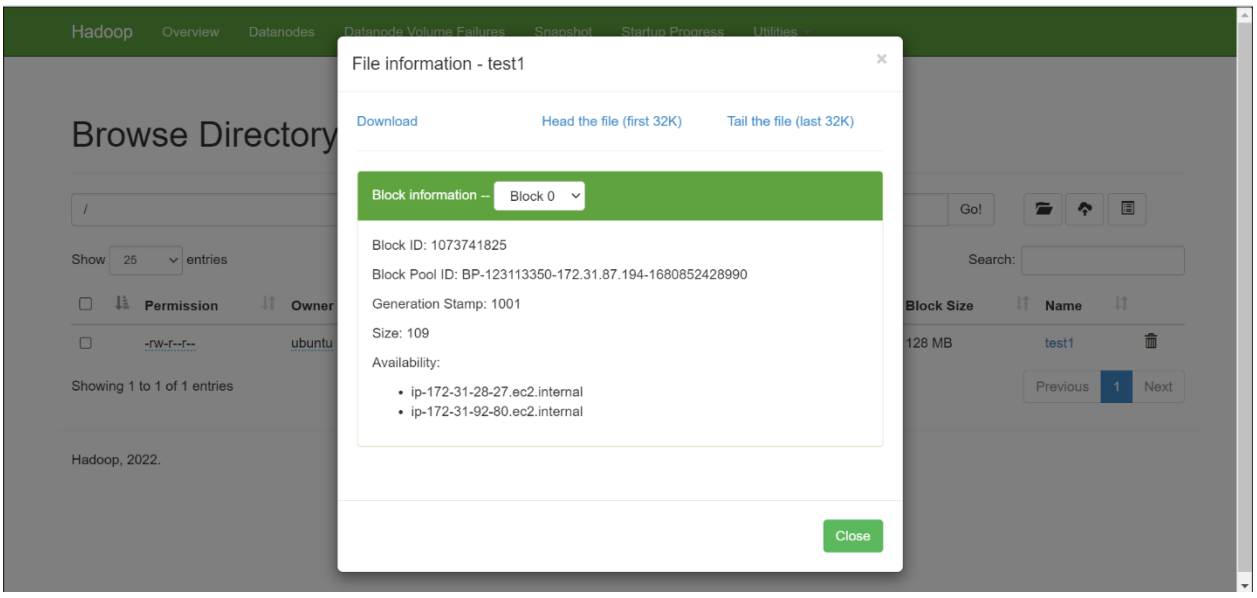


Storing it in the HDFS:

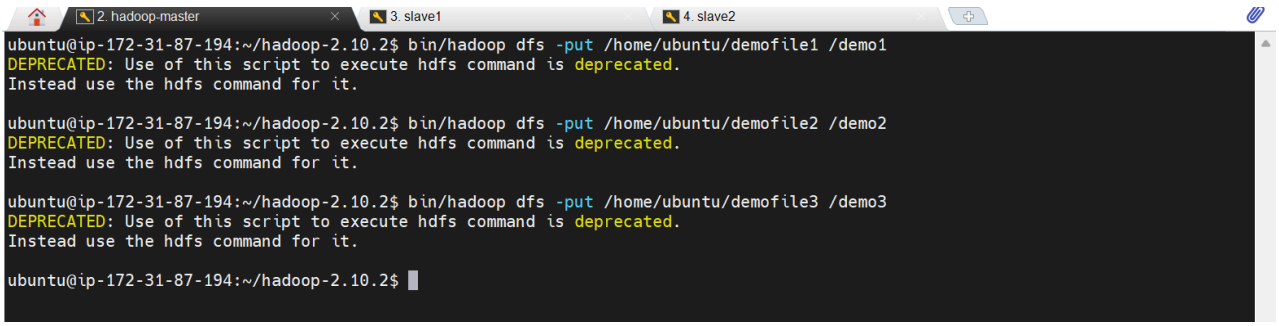


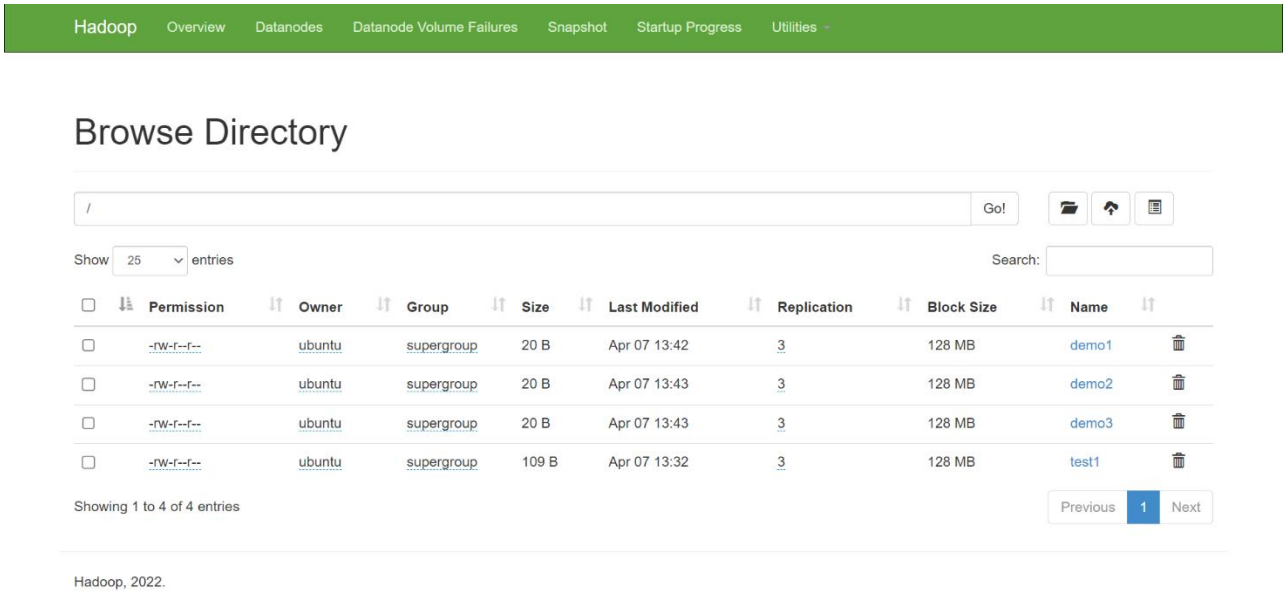
Successfully Stored in the DFS with replication factor of 3





Storing more files:





Conclusion:

In conclusion, our experiment studying MapReduce and Hadoop Distributed File System (HDFS) has shown that these technologies are powerful tools for managing and processing large-scale data sets in a distributed computing environment.

The implementation of Hadoop Distributed File System (HDFS) with one master and two slave nodes on Amazon Web Services (AWS) EC2 instances was successful. The experiment showed that HDFS can be deployed and managed on the cloud, allowing for the storage and processing of large amounts of data in a distributed manner.

References:

- Data Engineering. (2022, April 12). Hadoop Multi Node Cluster Setup [Video]. YouTube. https://www.youtube.com/watch?v=_iP2Em-5Abw
- Kumar, M. R. N. (2022, September 17). Hadoop-3.3.1 Installation guide for Ubuntu - Dev Genius [Video]. Medium. https://blog.devgenius.io/install-configure-and-setup-hadoop-in-ubuntu-a3cdd6305a0e
- Gaurav Sharma. (2022, October 20). AWS Tutorials - 10 - Create First EC2 Instance | EC2 Instance Creation in AWS [Video]. YouTube. https://www.youtube.com/watch?v=f-T4xWUZWSk

Postlab Questions:

1. What are the differences between traditional file systems and HDFS?
2. Enlist key features and components of HDFS.