# Distributed Systems LAB

(CO407)

7th Semester (2020-2021)

Batch: A1 G3


Submitted To:
Mr. Rahul
Assistant Professor
Department of Computer Engineering,
DTU

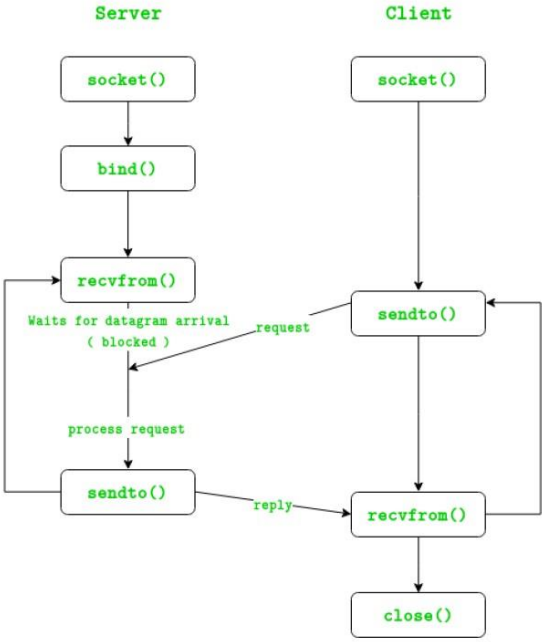
Submitted by:
Anuj Soni
2K17/CO/070
CO-A1

| S. No. | Experiment | Date | Remarks |
|---|---|---|---|
| 1. | To implement concurrent day-time client-server application. | 25 August 2020 | |
| 2. | To implement Berkeley clock synchronization algorithm. | 1 September 2020 | |
| 3. | To implement Lamport Clock synchronization between processes with different clocks and update intervals by exchanging messages between them. | 8 September 2020 | |
| 4. | To implement Mutual Exclusion using centralized algorithm. | 15 September 2020 | |
| 5. | To implement Mutual Exclusion using Token Ring Algorithm. | 22 September 2020 | |
| 6. | To implement Bully Election Algorithm. | 29 September 2020 | |
| 7. | To implement Ring Election Algorithm. | 6 October 2020 | |
| 8. | To implement Election Algorithm for Wireless Network. | 13 October 2020 | |
| 9. | To implement entry eventual consistency between processes with mutual exclusive update replicated datastore. | 20 October 2020 | |
| 10. | To implement 2-Phase Commit client-server. | 27 October 2020 | |

# PROGRAM-1

**Aim:** Implement concurrent day-time client-server application.

**Introduction:** There are two major transport layer protocols to communicate between hosts: TCP and UDP. In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.




**Socket**

A socket is a combination of IP address and port on one system. On each system a socket exists for a process interacting with the socket on other system over the network. A combination of local socket and the socket at the remote system is also known a 'Four tuple' or '4-tuple'. Each connection between two processes running at different systems can be uniquely identified through their 4-tuple. **Function Descriptions socket()**

Creates an UN-named socket inside the kernel and returns an integer known as socket descriptor. This function takes domain/family as its first argument. For Internet family of ipv4 addresses we use AF_INET. The second argument 'SOCK_STREAM' specifies that the transport layer protocol that we want should be reliable i.e. It should have acknowledgement techniques. The third argument is generally left zero to let the kernel decide the default protocol to use for this connection. For connection oriented reliable connections, the default protocol used is TCP.

**bind()**

Assigns the details specified in the structure 'serv_addr' to the socket created in the step above. The details include, the family/domain, the interface to listen on(in case the system has multiple interfaces to network) and the port on which the server will wait for the client requests to come. **listen()**

With second argument as '10' specifies maximum number of client connections that server will queue for this listening socket. After the call to listen(), this socket becomes a fully functional listening socket.

**accept()**

The server is put to sleep and when for an incoming client request, the three-way TCP handshake is complete, the function accept () wakes up and returns the socket descriptor representing the client socket. Accept() is run in an infinite loop so that the server is always running and the delay or sleep of 1 sec ensures that this server does not eat up all your CPU processing. As soon as server gets a request from client, it prepares the date and time and writes on the client socket through the descriptor returned by accept().

## Algorithm / Explanation:

Server:

1. create UDP socket
2. Bind socket to address
3. wait for datagram from client
4. process and reply to client request
5. repeat while server is active

Client:

1. create UDP socket
2. send request to server
3. wait for datagram from server
4. process and reply from server
5. close socket and exit

## Code:

Server:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
int main()
{
    struct sockaddr_in sa;                    // Socket address data structure
    int n, sockfd;                            // read and source
    char buff[1025];                          // buffer to store the readstream
    sockfd = socket(PF_INET, SOCK_STREAM, 0); // New socket created
    // Checking for valid socket
    if (sockfd < 0)
    {
        printf("Error in creation\n");
        exit(0);
    }
    else
    {
        printf("Socket created\n");
    }
    // Clearing and assigning type and address to the socket
    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(5600);
```

```
    // establishing and verifying the connection
    if (connect(sockfd, (struct sockaddr *)&sa, sizeof(sa)) < 0)
    {
        printf("Connection failed\n");
        exit(0);
    }

    else
        printf("Connection made\n"); // Reading and priting data from the server
after verification

    if (n = read(sockfd, buff, sizeof(buff)) < 0)
    {
        printf("Read Error\n");
        exit(0);
    }
    else
    {
        printf("Read message: %s\n", buff);
        printf("%s\n", buff);
        printf("Done with connection, exiting\n");
    }
    close(sockfd); // Closing the socket
    return 0;
}
```

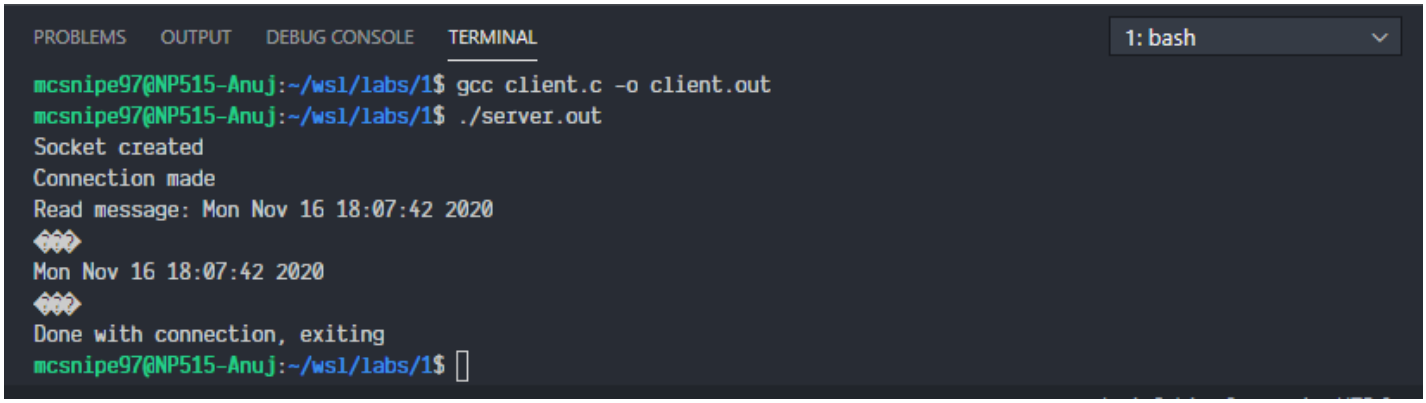
Client:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
int main()
{
    struct sockaddr_in sa;                    // Socket address data structure
    int sockfd, coontfd;                      // Source and destination
addresses
    char str[1025];                           // Buffer to hold the out-going
stream
    time_t tick;                              // System time data structure
    sockfd = socket(AF_INET, SOCK_STREAM, 0); // New socket created
                                              // Checking for valid socket
    if (sockfd < 0)
    {
        printf("Error in creating socket\n");
        exit(0);
    }
    else
```

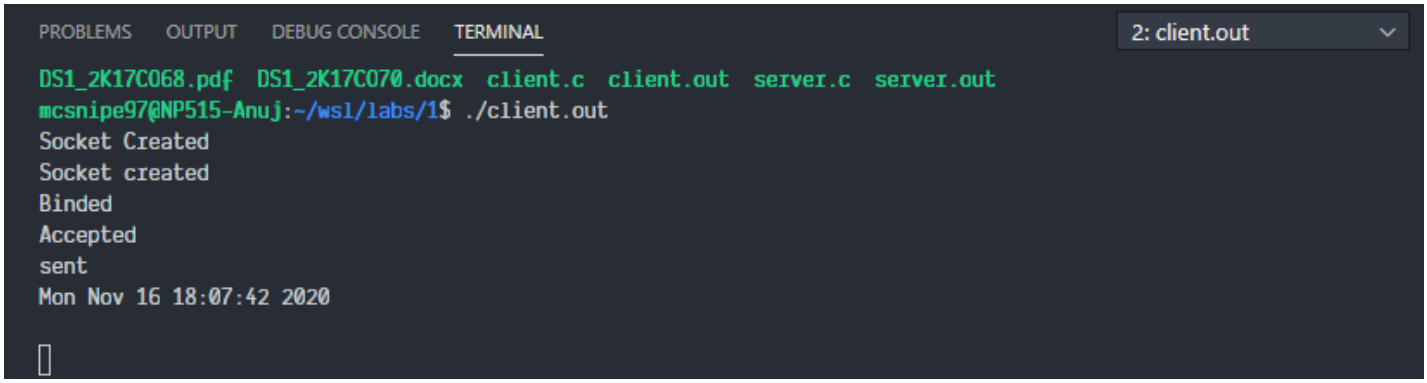
```
    {
        printf("Socket Created\n");
    }
    // Clearing and assigning type and address to the socket
    printf("Socket created\n");
    bzero(&sa, sizeof(sa));
    memset(str, '0', sizeof(str)); // clearing the buffer
    sa.sin_family = AF_INET;
    sa.sin_port = htons(5600);
    sa.sin_addr.s_addr = htonl(INADDR_ANY); // binding and verifying the socket
to address
    if (bind(sockfd, (struct sockaddr *)&sa, sizeof(sa)) < 0)
    {
        printf("Bind Error\n");
    }
    else
        printf("Binded\n");
    // starts the server with a max client queue size set as 10
    listen(sockfd, 10); // server run
    while (1)
    {
        coontfd = accept(sockfd, (struct sockaddr *)NULL, NULL); // Accept a
request from client
        printf("Accepted\n");
        tick = time(NULL);
        snprintf(str, sizeof(str), "%.24s\r\n", ctime(&tick)); //read sys time
and write to buffer
        printf("sent\n");
        printf("%s\n", str);
        write(coontfd, str, strlen(str)); // send buffer to client
    }
    close(sockfd); // close the socket return 0;
}
```

## Output:

Server:



Client:




## Conclusion:

1. We successfully implemented a date-time client-server.
2. UDP is a connectionless protocol where the server waits for a request from a client to become active. Each connection is treated as a new one.
3. On a local system i.e. within the same computer, the loop back address should be used as the argument to the client.
4. The connect procedure follows the Three-way handshake process to establish the connection.

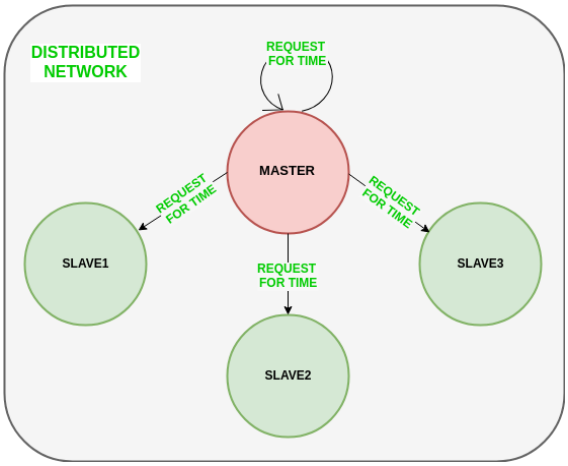
# PROGRAM - 2

**Aim**: To implement Berkeley clock synchronization algorithm.

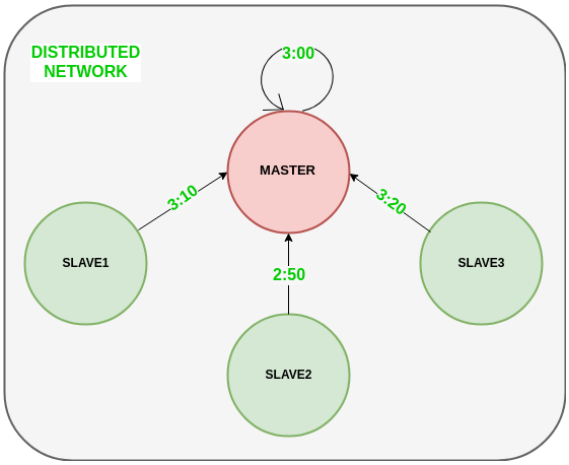
**Theory**: Berkeley's Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess an UTC server.

Steps involved in Berkeley's Algorithm:

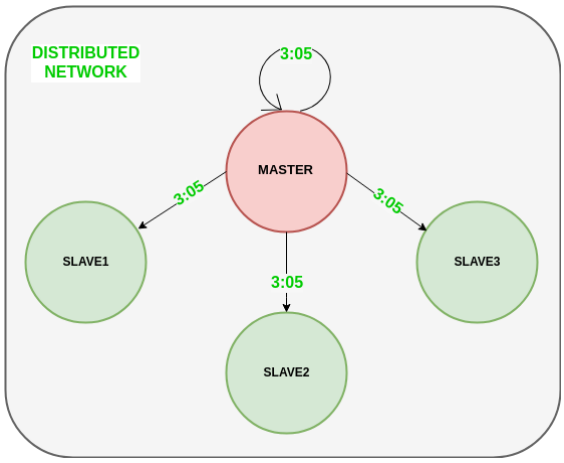
1.




2.




3.



**Code**:

Server:

```
from functools import reduce
from dateutil import parser
import threading
import datetime
import socket
import time


# datastructure used to store client address and clock data
client_data = {}


''' nested thread function used to receive
    clock time from a connected client '''


def startRecieveingClockTime(connector, address):

    while True:
        # recieve clock time
        clock_time_string = connector.recv(1024).decode()
        clock_time = parser.parse(clock_time_string)
        clock_time_diff = datetime.datetime.now() - \
            clock_time

        client_data[address] = {
            "clock_time": clock_time,
            "time_difference": clock_time_diff,
            "connector": connector
        }

        print("Client Data updated with: " + str(address),
              end="\n\n")
        time.sleep(5)


''' master thread function used to open portal for
    accepting clients over given port '''


def startConnecting(master_server):

    # fetch clock time at slaves / clients
    while True:
        # accepting a client / slave clock client
        master_slave_connector, addr = master_server.accept()
```

```
        slave_address = str(addr[0]) + ":" + str(addr[1])

        print(slave_address + " got connected successfully")

        current_thread = threading.Thread(
            target=startRecieveingClockTime,
            args=(master_slave_connector,
                  slave_address, ))
        current_thread.start()


# subroutine function used to fetch average clock difference
def getAverageClockDiff():

    current_client_data = client_data.copy()

    time_difference_list = list(client['time_difference']
                                for client_addr, client
                                in client_data.items())

    sum_of_clock_difference = sum(time_difference_list,
                                  datetime.timedelta(0, 0))

    average_clock_difference = sum_of_clock_difference \
        / len(client_data)

    return average_clock_difference


''' master sync thread function used to generate
    cycles of clock synchronization in the network '''


def synchronizeAllClocks():

    while True:

        print("New synchroniztion cycle started.")
        print("Number of clients to be synchronized: " +
              str(len(client_data)))

        if len(client_data) > 0:

            average_clock_difference = getAverageClockDiff()

            for client_addr, client in client_data.items():
                try:
                    synchronized_time = \
                        datetime.datetime.now() + \
                        average_clock_difference
```

```
                    client['connector'].send(str(
                        synchronized_time).encode())

                except Exception as e:
                    print("Something went wrong while " +
                          "sending synchronized time " +
                          "through " + str(client_addr))

        else:
            print("No client data." +
                  " Synchronization not applicable.")

        print("\n\n")

        time.sleep(5)


# function used to initiate the Clock Server / Master Node
def initiateClockServer(port=8080):

    master_server = socket.socket()
    master_server.setsockopt(socket.SOL_SOCKET,
                             socket.SO_REUSEADDR, 1)

    print("Socket at master node created successfully\n")

    master_server.bind(('', port))

    # Start listening to requests
    master_server.listen(10)
    print("Clock server started...\n")

    # start making connections
    print("Starting to make connections...\n")
    master_thread = threading.Thread(
        target=startConnecting,
        args=(master_server, ))
    master_thread.start()

    # start synchroniztion
    print("Starting synchronization parallely...\n")
    sync_thread = threading.Thread(
        target=synchronizeAllClocks,
        args=())
    sync_thread.start()


# Driver function
if __name__ == '__main__':
```

```
    # Trigger the Clock Server
    initiateClockServer(port=8080)
```

Client:

```
from timeit import default_timer as timer
from dateutil import parser
import threading
import datetime
import socket
import time


# client thread function used to send time at client side
def startSendingTime(slave_client):

    while True:
        # provide server with clock time at the client
        slave_client.send(str(
            datetime.datetime.now()).encode())

        print("Recent time sent successfully",
              end="\n\n")
        time.sleep(5)


# client thread function used to receive synchronized time
def startReceivingTime(slave_client):

    while True:
        # receive data from the server
        Synchronized_time = parser.parse(
            slave_client.recv(1024).decode())

        print("Synchronized time at the client is: " +
              str(Synchronized_time),
              end="\n\n")


# function used to Synchronize client process time
def initiateSlaveClient(port=8080):

    slave_client = socket.socket()

    # connect to the clock server on local computer
    slave_client.connect(('127.0.0.1', port))

    # start sending time to server
```

```
    print("Starting to receive time from server\n")
    send_time_thread = threading.Thread(
        target=startSendingTime,
        args=(slave_client, ))
    send_time_thread.start()

    # start recieving synchronized from server
    print("Starting to recieving " +
          "synchronized time from server\n")
    receive_time_thread = threading.Thread(
        target=startReceivingTime,
        args=(slave_client, ))
    receive_time_thread.start()


# Driver function
if __name__ == '__main__':

    # initialize the Slave / Client
    initiateSlaveClient(port=8080)
```

## Output:

Server:


Client:

1:


2:


3:


## **Conclusion**:

Berkeley clock synchronization algorithm is implemented.

# PROGRAM-3

## Aim:

Lamport logical clock synchronization between processes with different clocks and update intervals. The processes must exchange messages and correct clocks if required. After each interaction the process must show the clock after updating if required.

## Theory:

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 in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender (causality).

## Algorithm:

- All the process counters start with value 0.
- A process increments its counter for each event (internal event, message sending, message receiving) in that process.
- When a process sends a message, it includes its (incremented) counter value with the message.
- On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.

## Code:

```
import signal
import sys
import time
import threading
from queue import Queue

initially_granted_proc = "A"
procs = {"A", "B", "C"}
resource_usage_counts = {"A": 0, "B": 0, "C": 0}
message_queues = {"A": Queue(), "B": Queue(), "C": Queue()}


class Message(object):
    def __init__(self, msg_type, timestamp, sender, receiver):
        self.msg_type = msg_type
        self.timestamp = timestamp
        self.sender = sender
        self.receiver = receiver

    def __repr__(self):
        return "Message {} at {} from {} to {}".format(
            self.msg_type, self.timestamp,
            self.sender, self.receiver)


class Process(threading.Thread):
```

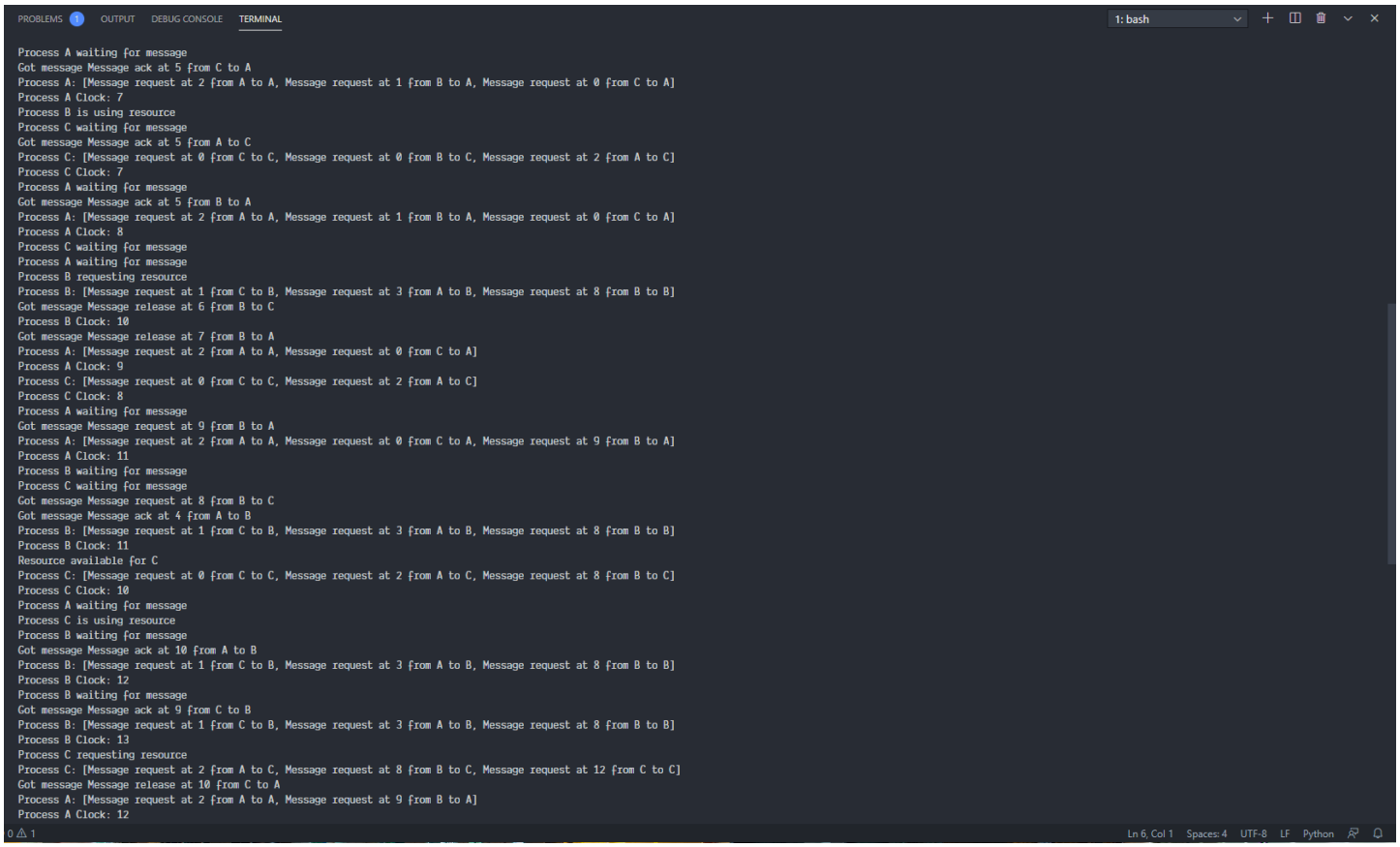
```
    def __init__(self, name, initially_granted, other_processes):
        super(Process, self).__init__()
        self.name = name
        self.has_resource = initially_granted == name
        self.other_processes = other_processes
        self.lamport_clock = 0  # tick after each "event"
        self.request_queue = []
        self.requested = False
        self.request_queue.append(Message("request",
                                          -1, initially_granted,
initially_granted))

    def remove_request(self, msg_type, sender):
        index_of_req = -1
        for i in range(len(self.request_queue)):
            if self.request_queue[i].msg_type == msg_type and \
               self.request_queue[i].sender == sender:
                index_of_req = i
                break
        if i == -1:
            print("Unable to remove")
        else:
            del self.request_queue[i]

    def use_resource(self):
        print("Process {} is using resource".format(self.name))
        resource_usage_counts[self.name] += 1
        time.sleep(2)

    def process_message(self, msg):
        # Based on msg_type handle appropriately
        if msg.msg_type == "request":
            # Put in our request queue and send an ack
            # to the sender
            self.request_queue.append(msg)
            for proc in self.other_processes:
                if proc == msg.sender:
                    message_queues[proc].put(Message(
                        "ack", self.lamport_clock,
                        self.name, msg.sender))
        elif msg.msg_type == "release":
            # Got a release, remove it from our queue
            self.remove_request("request", msg.sender)
        elif msg.msg_type == "ack":
            pass
        else:
            print("Unknown message type")

    def run(self):
        while True:
            if self.has_resource:
                self.use_resource()
```

```
            self.remove_request("request", self.name)
            # Tell everyone that we are done
            for proc in self.other_processes:
                message_queues[proc].put(Message(
                    "release", self.lamport_clock,
                    self.name, proc))
                self.lamport_clock += 1
            self.has_resource, self.requested = False, False
            continue
        # Want to get the resource
        if not self.requested:
            # Request it
            print("Process {} requesting resource".format(
                self.name))
            self.request_queue.append(Message(
                "request", self.lamport_clock,
                self.name, self.name))
            # Broadcast this request
            for proc in self.other_processes:
                message_queues[proc].put(Message(
                    "request", self.lamport_clock,
                    self.name, proc))
                self.lamport_clock += 1
            self.requested = True
        else:
            # Just wait until it is available by processing messages
            print("Process {} waiting for message".format(self.name))
            msg = message_queues[self.name].get(block=True)
            # Got a message, check if the timestamp
            # is greater than our clock, if so advance it
            if msg.timestamp >= self.lamport_clock:
                self.lamport_clock = msg.timestamp + 1
            print("Got message {}".format(msg))
            self.process_message(msg)
            self.lamport_clock += 1
            # Check after processing if the resource is
            # available for me now, if so, grab it.
            # We need earliest request to be ours and check that we
            # have received an older message from everyone else
            if self.check_available():
                print("Resource available for {}".format(self.name))
                self.has_resource = True
        print("Process {}: {}".format(self.name, self.request_queue))
        print("Process {} Clock: {}".format(self.name, self.lamport_clock))
        time.sleep(1)

def check_available(self):
    got_older = {k: False for k in self.other_processes}
    # Get timestamp of our req
    our_req = None
    for req in self.request_queue:
        if req.sender == self.name:
            our_req = req
```

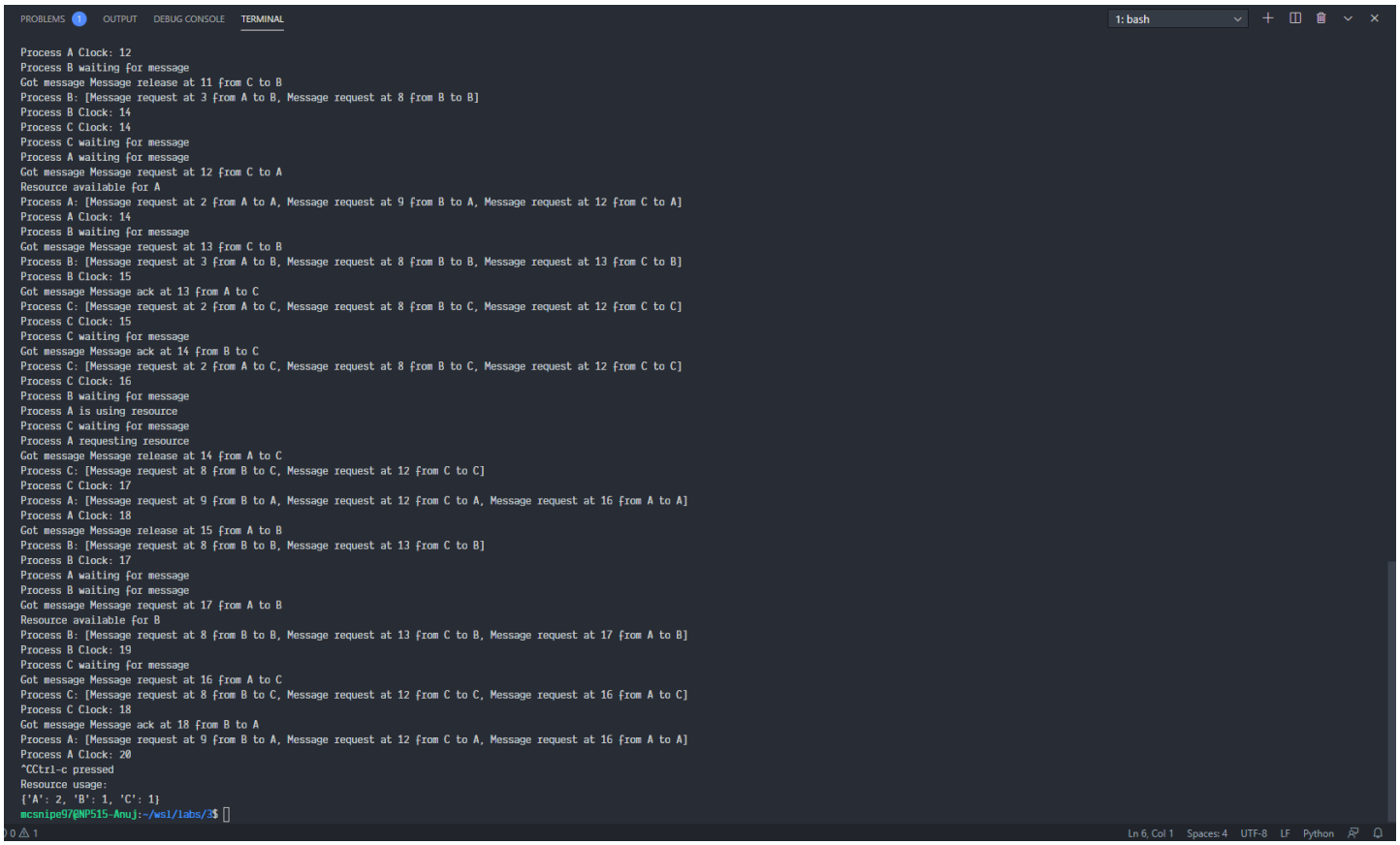
```
        if our_req is None:
            return False
        # We found our req make sure it is younger than
        # all the others and we have an older one from
        # the other guys
        for req in self.request_queue:
            if req.sender in got_older and req.timestamp > our_req.timestamp:
                got_older[req.sender] = True
        if all(got_older.values()):
            return True
        return False


t1 = Process("A", initially_granted_proc, list(procs - set("A")))
t2 = Process("B", initially_granted_proc, list(procs - set("B")))
t3 = Process("C", initially_granted_proc, list(procs - set("C")))

# Daemonizing threads means that if main thread dies, so do they.
# That way the process will exit if the main thread is killed.
t1.setDaemon(True)
t2.setDaemon(True)
t3.setDaemon(True)

try:
    t1.start()
    t2.start()
    t3.start()
    while True:
        # Need some arbitrary timeout here, seems a bit hackish.
        # If we don't do this then the main thread will just block
        # forever waiting for the threads to return and the
        # keyboardinterrupt never gets hit. Interestingly regardless of the
        # timeout, the keyboard interrupt still occurs immediately
        # upon ctrl-c'ing
        t1.join(100)
        t2.join(100)
        t3.join(100)
except KeyboardInterrupt:
    print("Ctrl-c pressed")
    print("Resource usage:")
    print(resource_usage_counts)
    sys.exit(1)
```

# Output:

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL                                  1: bash

mcsnipe97@NP515-Anuj:~/wsl/labs/3$ python3 code.py
Process A is using resource
Process B requesting resource
Process B: [Message request at -1 from A to A, Message request at 0 from B to B]
Process C requesting resource
Process B Clock: 2
Process C: [Message request at -1 from A to A, Message request at 0 from C to C]
Process C Clock: 2
Process B waiting for message
Got message Message request at 1 from C to B
Process B: [Message request at -1 from A to A, Message request at 0 from B to B, Message request at 1 from C to B]
Process B Clock: 3
Process C waiting for message
Got message Message request at 0 from B to C
Process C: [Message request at -1 from A to A, Message request at 0 from C to C, Message request at 0 from B to C]
Process C Clock: 3
Process A requesting resource
Process A: [Message request at 2 from A to A]
Process A Clock: 4
Process B waiting for message
Process C waiting for message
Got message Message ack at 2 from C to B
Got message Message ack at 2 from B to C
Process B: [Message request at -1 from A to A, Message request at 0 from B to B, Message request at 1 from C to B]
Process B Clock: 4
Process C: [Message request at -1 from A to A, Message request at 0 from C to C, Message request at 0 from B to C]
Process C Clock: 4
Process A waiting for message
Got message Message request at 1 from B to A
Process A: [Message request at 2 from A to A, Message request at 1 from B to A]
Process A Clock: 5
Process C waiting for message
Got message Message release at 0 from A to C
Process C: [Message request at 0 from C to C, Message request at 0 from B to C]
Process C Clock: 5
Process B waiting for message
Got message Message release at 1 from A to B
Process B: [Message request at 0 from B to B, Message request at 1 from C to B]
Process B Clock: 5
Process A waiting for message
Got message Message request at 0 from C to A
Process A: [Message request at 2 from A to A, Message request at 1 from B to A, Message request at 0 from C to A]
Process A Clock: 6
Process C waiting for message
Got message Message request at 2 from A to C
Process C: [Message request at 0 from C to C, Message request at 0 from B to C, Message request at 2 from A to C]
Process C Clock: 6
Process B waiting for message
Got message Message request at 3 from A to B
Resource available for B
Process B: [Message request at 0 from B to B, Message request at 1 from C to B, Message request at 3 from A to B]
Process B Clock: 6
0 ⚠ 1                                  Ln 6, Col 1  Spaces: 4  UTF-8  LF  Python
```



```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL                                  1: bash

Process A waiting for message
Got message Message ack at 5 from C to A
Process A: [Message request at 2 from A to A, Message request at 1 from B to A, Message request at 0 from C to A]
Process A Clock: 7
Process B is using resource
Process C waiting for message
Got message Message ack at 5 from A to C
Process C: [Message request at 0 from C to C, Message request at 0 from B to C, Message request at 2 from A to C]
Process C Clock: 7
Process A waiting for message
Got message Message ack at 5 from B to A
Process A: [Message request at 2 from A to A, Message request at 1 from B to A, Message request at 0 from C to A]
Process A Clock: 8
Process C waiting for message
Process A waiting for message
Process B requesting resource
Process B: [Message request at 1 from C to B, Message request at 3 from A to B, Message request at 8 from B to B]
Got message Message release at 6 from B to C
Process B Clock: 10
Got message Message release at 7 from B to A
Process A: [Message request at 2 from A to A, Message request at 0 from C to A]
Process A Clock: 9
Process C: [Message request at 0 from C to C, Message request at 2 from A to C]
Process C Clock: 8
Process A waiting for message
Got message Message request at 9 from B to A
Process A: [Message request at 2 from A to A, Message request at 0 from C to A, Message request at 9 from B to A]
Process A Clock: 11
Process B waiting for message
Process C waiting for message
Got message Message request at 8 from B to C
Got message Message ack at 4 from A to B
Process B: [Message request at 1 from C to B, Message request at 3 from A to B, Message request at 8 from B to B]
Process B Clock: 11
Resource available for C
Process C: [Message request at 0 from C to C, Message request at 2 from A to C, Message request at 8 from B to C]
Process C Clock: 10
Process A waiting for message
Process C is using resource
Process B waiting for message
Got message Message ack at 10 from A to B
Process B: [Message request at 1 from C to B, Message request at 3 from A to B, Message request at 8 from B to B]
Process B Clock: 12
Process B waiting for message
Got message Message ack at 9 from C to B
Process B: [Message request at 1 from C to B, Message request at 3 from A to B, Message request at 8 from B to B]
Process B Clock: 13
Process C requesting resource
Process C: [Message request at 2 from A to C, Message request at 8 from B to C, Message request at 12 from C to C]
Got message Message release at 10 from C to A
Process A: [Message request at 2 from A to A, Message request at 9 from B to A]
Process A Clock: 12
0 ⚠ 1                                  Ln 6, Col 1  Spaces: 4  UTF-8  LF  Python
```




## Conclusion:

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

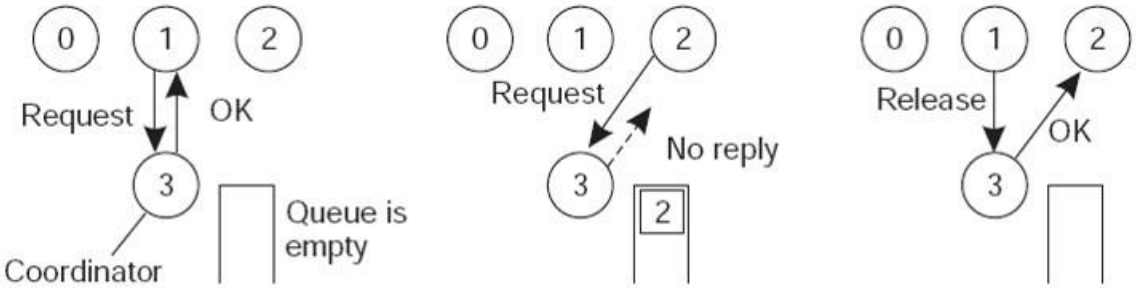
# PROGRAM – 4

**Aim:** Program to implement Mutual Exclusion using centralized algorithm.

## Theory:

In centralized algorithm one process is elected as the coordinator which may be the machine. Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission). When the reply arrives, the requesting process enters the critical region. When another process asks for permission to enter the same critical region. Now the coordinator knows that a different process is already in the critical region, so it cannot grant permission.

The coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply or it could send a reply 'permission denied.' When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. When it sees the grant, it can enter the critical region




## Algorithm:

Coordinator Loop

receive(msg);

case msg of REQUEST:

if nobody in CS

then reply GRANTED

else queue the REQ;

reply DENIED

RELEASE:

if queue not empty

then remove 1st on the queue

reply GRANTED

end case

end loop

Client:
send(REQUEST);

receive(msg);

if msg != GRANTED

then receive(msg);

enter CS;

send(RELEASE)

## Code:

Server:

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

public class Server implements Runnable {
    Socket socket = null;
    static ServerSocket ss;
    Server(Socket newSocket) {
        this.socket = newSocket;
    }

    public static void main(String args[]) throws IOException {
        ss = new ServerSocket(7000);
        System.out.println("Server Started");

        while (true) {
            Socket s = ss.accept();
            Server es = new Server(s);
            Thread t = new Thread(es);
            t.start();
        }
    }

    public void run() {
        try {
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            while (true) {
                System.out.println(in.readLine());
            }
        } catch (Exception e) {
        }
    }
}
```

Client1:

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

public class Client1 {
    public static void main(String args[]) throws IOException {
        Socket s = new Socket("localhost", 7000);
        PrintStream out = new PrintStream(s.getOutputStream());

        ServerSocket ss = new ServerSocket(7001);
        Socket s1 = ss.accept();
        BufferedReader in1 = new BufferedReader(new
InputStreamReader(s1.getInputStream()));
        PrintStream out1 = new PrintStream(s1.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = "Token";
        while (true) {
            if (str.equalsIgnoreCase("Token")) {
                System.out.println("Do you want to send some data");
                System.out.println("Enter Yes or No");
                str = br.readLine();
                if (str.equalsIgnoreCase("Yes")) {
                    System.out.println("Enter the data");
                    str = br.readLine();
                    out.println(str);
                }
                out1.println("Token");
            }
            System.out.println("Waiting for Token");
            str = in1.readLine();
        }

    }
}
```

Client2:

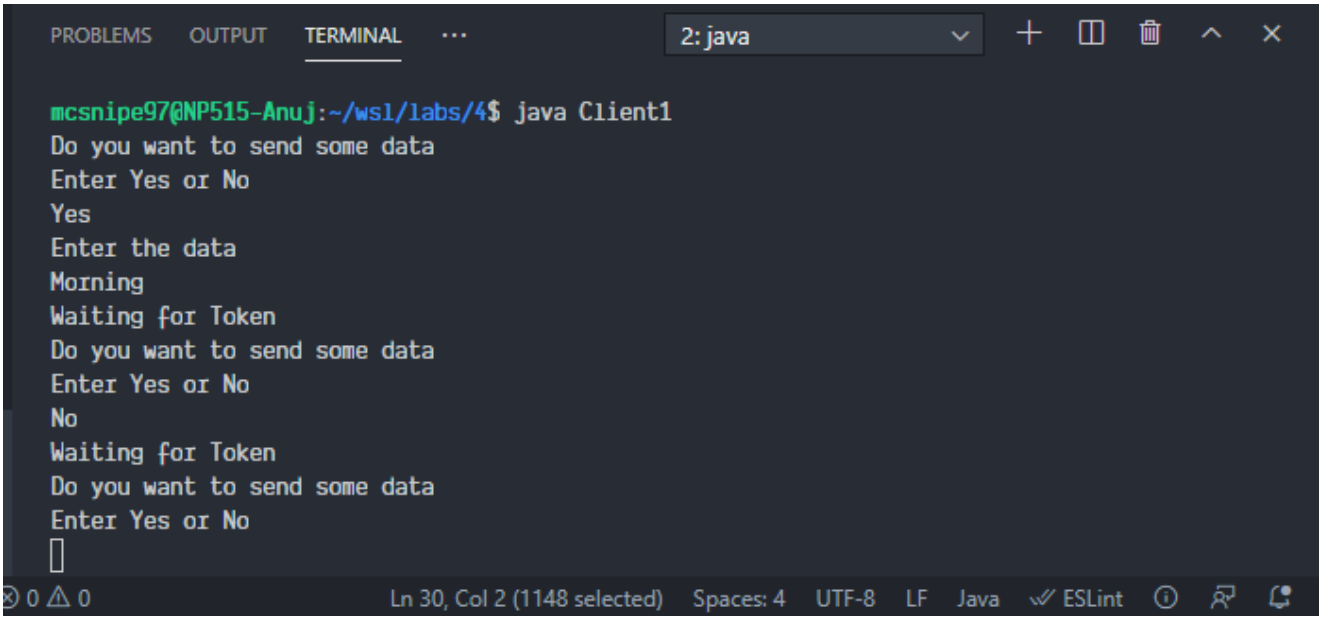
```
import java.io.*;
import java.net.*;

public class Client2 {
    public static void main(String args[]) throws IOException {
        Socket s = new Socket("localhost", 7000);
        PrintStream out = new PrintStream(s.getOutputStream());

        Socket s2 = new Socket("localhost", 7001);
        BufferedReader in2 = new BufferedReader(new
InputStreamReader(s2.getInputStream()));
        PrintStream out2 = new PrintStream(s2.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        while (true) {
            System.out.println("Waiting for Token");
            str = in2.readLine();
```

```
            if (str.equalsIgnoreCase("Token")) {
                System.out.println("Do you want to send some data");
                System.out.println("Enter Yes or No");
                str = br.readLine();
                if (str.equalsIgnoreCase("Yes")) {
                    System.out.println("Enter the data");
                    str = br.readLine();
                    out.println(str);
                }
                out2.println("Token");
            }
        }
    }
}
```

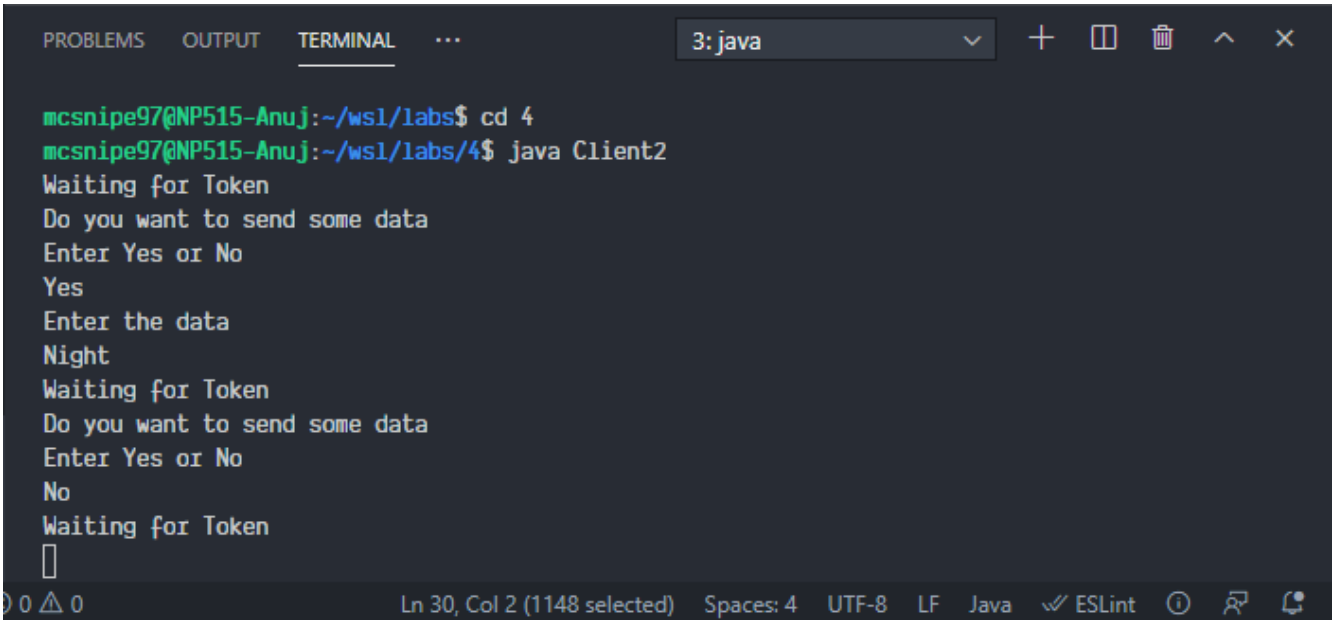
**Output:**

Server:


Client 1:



Client 2:



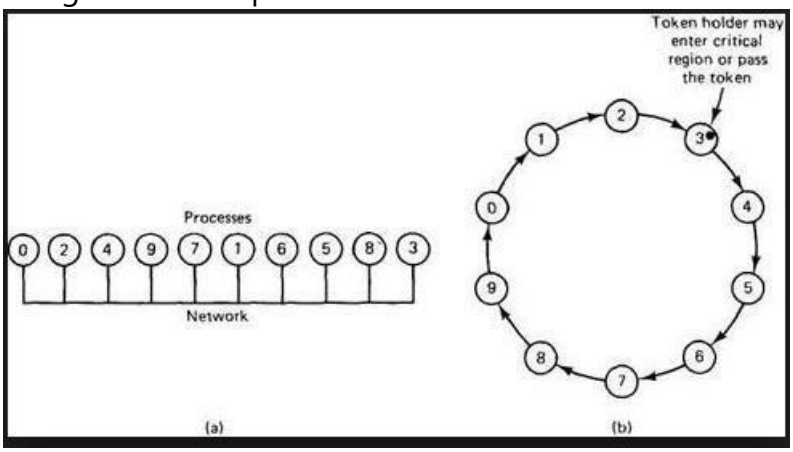

## Conclusion:

In this experiment, we implemented mutual exclusion using a centralized server. It simulates centralized locking with blocking calls. It is a simple method with no starvation and is fair. Although, it has the disadvantage of single point failure and is a performance bottleneck. This algorithm guarantees mutual exclusion by letting one process at a time into each critical region. It is also fair as requests are granted in the order in which they are received. It is easy to implement since it requires only three messages per use of a critical region (request, grant, release). This algorithm is also used for more general resource allocation rather than just managing critical regions.

**Aim:** Implement Mutual Exclusion using Token Ring Algorithm

## Theory:

Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes. A logical ring is constructed with these processes and each process is assigned a position in the ring. Each process knows who is next in line after itself. When the ring is initialized, process 0 is given a token. The token circulates around the ring. When a process acquires the token from its neighbour, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token to the next process in the ring. It is not allowed to enter the critical region again using the same token. If a process is handed the token by its neighbour and is not interested in entering a critical region, it just passes the token along to the next process.




Advantages:

- The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS.
- Since the token circulates among processes in a well-defined order, starvation cannot occur.

Disadvantages:

- Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region.
- If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is not a constant. The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.
- The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbour tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can pass the token to the next member down the line.

## Code:

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;
```

```
#define PR 6
int token_holder;

class Token
{
public:
    int id;             //Id of the site having token
    queue<int> token_q; //Token queue
    int ln[PR];         //Token Array for sequence no
    void init()         //Initializing token
    {
        id = 0;
        for (int i = 0; i < PR; i++)
        {
            ln[i] = 0;
        }
    }
} token;

//Description of each Site
class Site
{
public:
    int rn[PR];    //Site's Array for sequence no.
    bool exec;     //For checking whether site is executing
    bool isreq;    //For checking whether site is  requesting
    bool hastoken; //For checking whether site has token
    void init()    //Initializing sites
    {
        exec = 0;
        isreq = 0;
        hastoken = 0;
        for (int i = 0; i < PR; i++)
        {
            rn[i] = 0;
        }
    }
    void req(int pid, int seqno);
} site[PR];

//For a site to execute request of site pid with sequenceno seqno
void Site::req(int pid, int seqno)
{
    int i;
    rn[pid] = max(rn[pid], seqno);
    if (hastoken == 1)
    {
        if (exec == 0 && token.ln[pid] + 1 == rn[pid])
        {
            hastoken = 0;
            token_holder = pid;
```

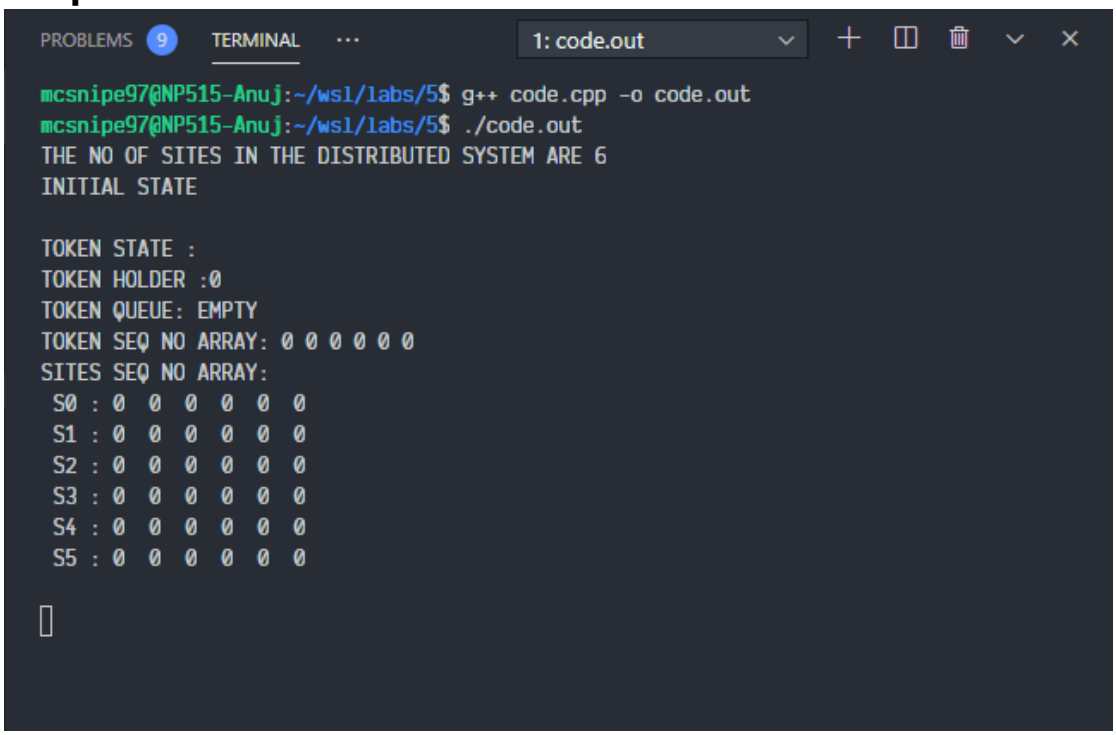
```
        }
        else if (token.ln[pid] + 1 == rn[pid])
        {
            token.token_q.push(pid);
        }
    }
}

//Initialize
void initialize()
{
    int i;
    token.init();
    for (i = 0; i < PR; i++)
    {
        site[i].init();
    }
}

//For a site with id pid to request for C.S.
void request(int pid)
{
    int i, seqno;
    seqno = ++site[pid].rn[pid];
    //Checking whether it has already requested
    if (site[pid].isreq == 1 || site[pid].exec == 1)
    {
        printf("SITE %d is already requesting \n", pid);
        return;
    }
    site[pid].isreq = 1;
    //Checking if it has the token
    if (token_holder == pid)
    {
        site[pid].isreq = 0;
        site[pid].exec = 1;
        printf("SITE  %d  already  has  the  token  and  it  enters  the  critical
section\n", pid);
        return;
    }

    //Sending Request
    if (token_holder != pid)
    {
        for (i = 0; i < PR; i++)
        {
            if (i != pid)
                site[i].req(pid, seqno);
        }
    }
    //Checking if it has got the token
    if (token_holder == pid)
```

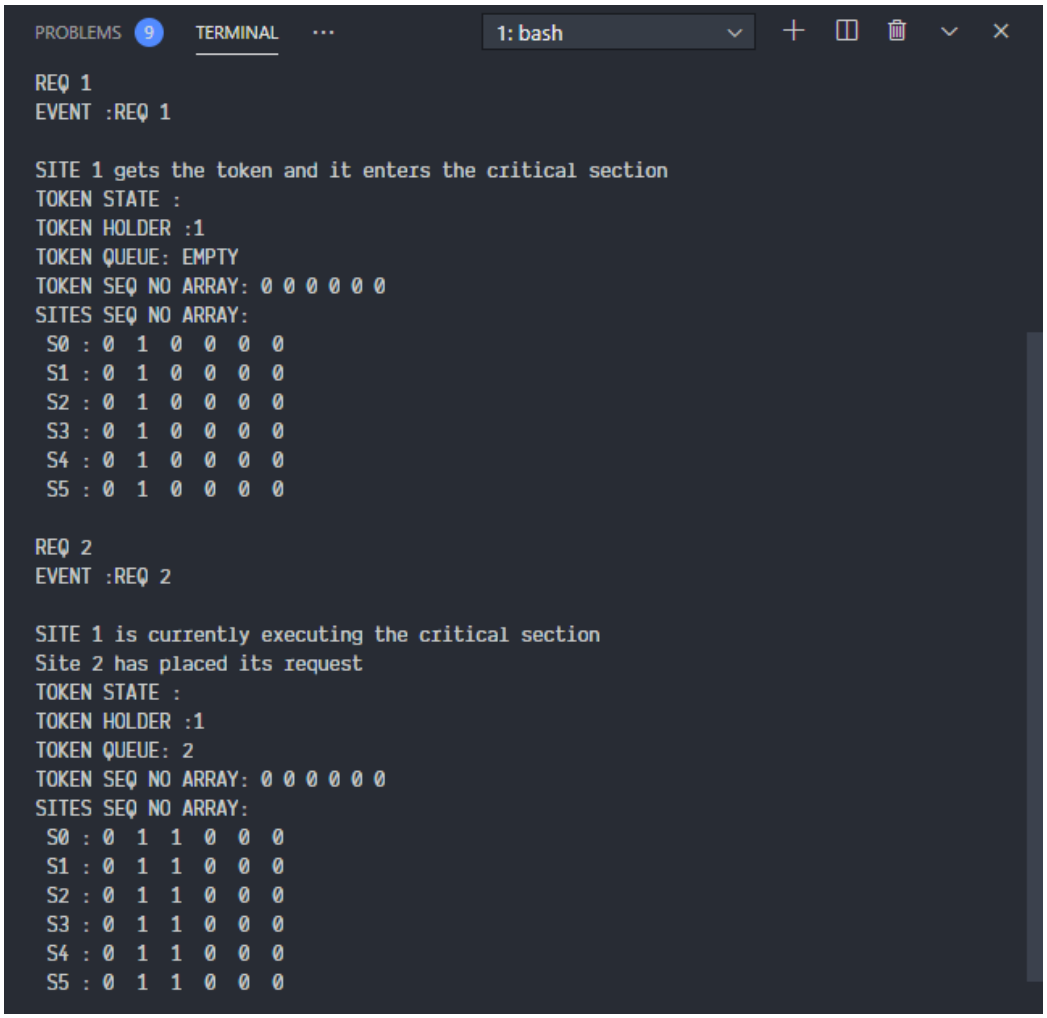
```
    {
        site[pid].hastoken = 1;
        site[pid].exec = 1;
        site[pid].isreq = 0;
        printf("SITE %d gets the token and it enters the critical section\n",
pid);
    }
    else
    {
        printf("SITE %d is currently executing the critical section \nSite %d has
placed its request\n", token_holder, pid);
    }
}

//For a site with id pid to request for C.S.
void release(int pid)
{

    if (site[pid].exec != 1)
    {
        printf("SITE %d is not currently executing the critical section \n", pid);
        return;
    }
    int i, siteid;
    token.ln[pid] = site[pid].rn[pid];
    site[pid].exec = 0;
    printf("SITE %d releases the critical section\n", pid);
    //Checking if deffred requests are there by checking token queue
    //And Passing the token if queue is non empty
    if (!token.token_q.empty())
    {
        siteid = token.token_q.front();
        token.token_q.pop();
        token.id = siteid;
        site[pid].hastoken = 0;
        token_holder = siteid;
        site[siteid].hastoken = 1;
        site[siteid].exec = 1;
        site[siteid].isreq = 0;
        printf("SITE %d gets the token and it enters the critical section\n",
siteid);
        return;
    }
    printf("SITE %d still has the token\n", pid);
}

//Printing the state of the system
void print()
{
    int i, j, k = 0;
    queue<int> temp;
    printf("TOKEN STATE :\n");
```

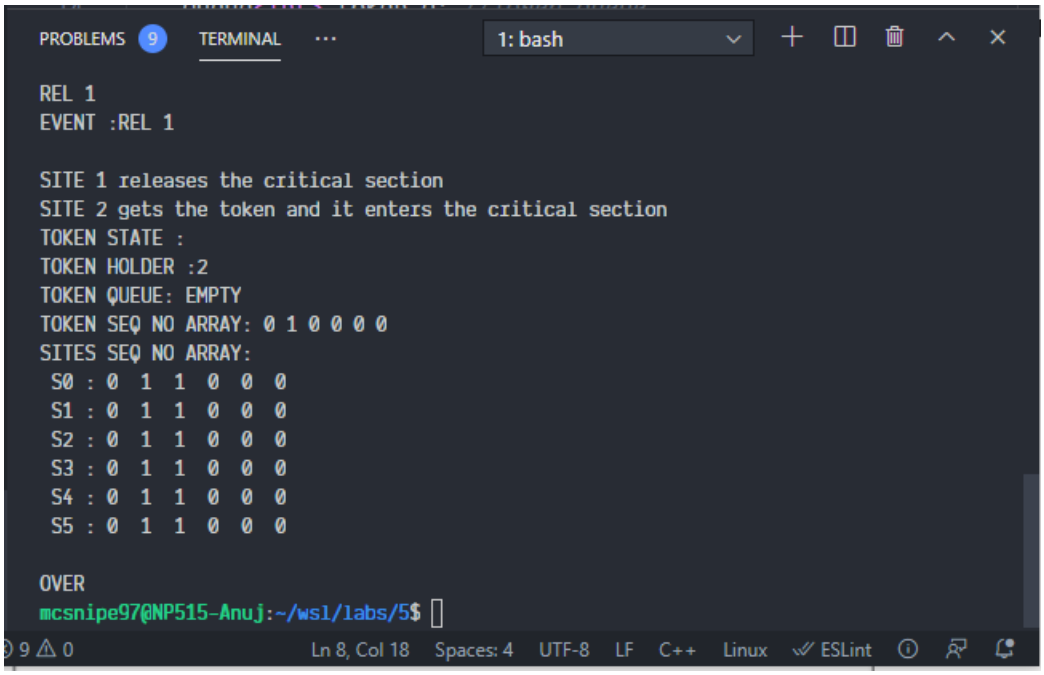
```
    printf("TOKEN HOLDER :%d\n", token_holder);
    printf("TOKEN QUEUE: ");
    if (token.token_q.empty())
    {
        printf("EMPTY");
        j = 0;
    }
    else
    {
        j = token.token_q.size();
    }
    while (k < j)
    {
        i = token.token_q.front();
        token.token_q.pop();
        token.token_q.push(i);
        printf("%d ", i);
        k++;
    }
    printf("\n");
    printf("TOKEN SEQ NO ARRAY: ");
    for (i = 0; i < PR; i++)
        printf("%d ", token.ln[i]);
    printf("\n");

    printf("SITES SEQ NO ARRAY: \n");
    for (i = 0; i < PR; i++)
    {
        printf(" S%d :", i);
        for (j = 0; j < PR; j++)
            printf(" %d ", site[i].rn[j]);
        printf("\n");
    }
}

int main()
{
    int i, j, time, pid;
    string str;
    initialize();
    token_holder = 0;
    site[0].hastoken = 1;

    time = 0;

    cout << "THE NO OF SITES IN THE DISTRIBUTED SYSTEM ARE " << PR << endl;
    cout << "INITIAL STATE\n"
         << endl;
    print();
    printf("\n");
    while (str != "OVER")
    {
```

```
        cin >> str;
        if (str == "REQ")
        {
            cin >> pid;
            cout << "EVENT :" << str << " " << pid << endl
                 << endl;
            request(pid);
            print();
            printf("\n");
        }
        else if (str == "REL")
        {
            cin >> pid;
            cout << "EVENT :" << str << " " << pid << endl
                 << endl;
            release(pid);
            print();
            printf("\n");
        }
    }
}
```

**Output:**






## Conclusion:

1. We successfully implemented Token-Ring Mutual Exclusion.
2. This avoids Starvation
3. Lost Key is a major issue.

# PROGRAM-6

**Aim:** Implement Bully Election Algorithm

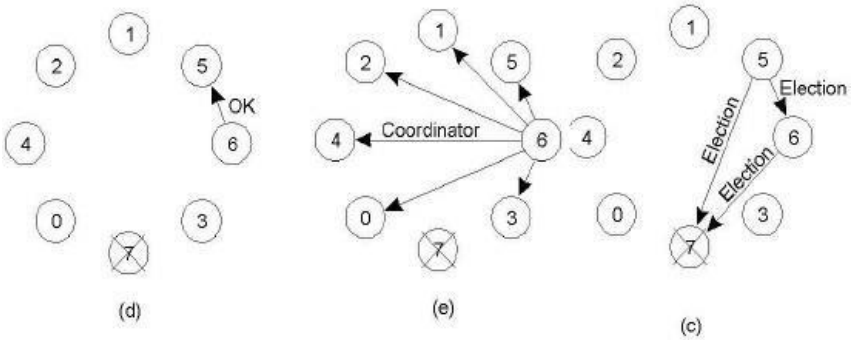
## Theory:

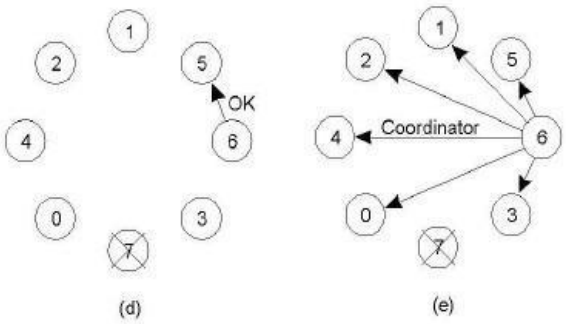
### Election Algorithms

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

The Bully Election Process

1. P sends a message to the coordinator.
2. If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
3. Now process P sends election message to every process with high priority number.
4. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
5. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
6. However, if an answer is received within time T from any other process Q,
   (I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.
   (II) If Q doesn't respond within time interval T' then it is assumed to have failed and algorithm is restarted.



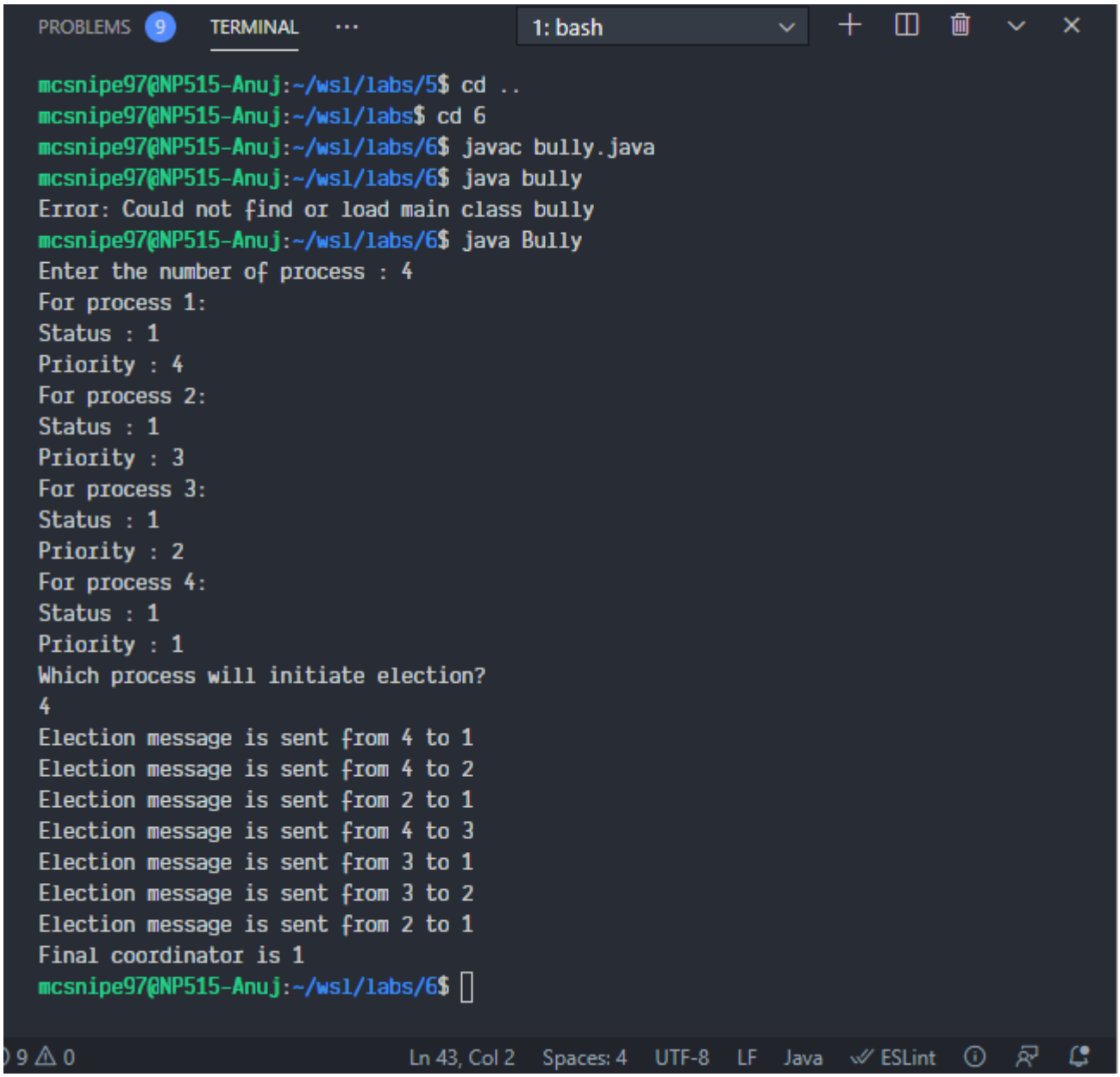
Disadvantages:

- A large number of messages are sent, this can overload the system.
- There may be cases in very large systems that multiple coordinators get elected.

## Code:

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

class Bully {
    static int n;
    static int pro[] = new int[100];
    static int sta[] = new int[100];
    static int co;

    public static void main(String args[]) throws IOException {
        System.out.print("Enter the number of process : ");
        Scanner in = new Scanner(System.in);
        n = in.nextInt();

        int i, j, k, l, m;

        for (i = 0; i < n; i++) {
            System.out.println("For process " + (i + 1) + ":");
            System.out.print("Status : ");
            sta[i] = in.nextInt();
            System.out.print("Priority : ");
            pro[i] = in.nextInt();
        }

        System.out.println("Which process will initiate election?");
        int ele = in.nextInt();

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

    static void elect(int ele) {
        ele = ele - 1;
        co = ele + 1;
        for (int i = 0; i < n; i++) {
            if (pro[ele] < pro[i]) {
                System.out.println("Election message is sent from " + (ele + 1) +
" to " + (i + 1));
                if (sta[i] == 1)
                    elect(i + 1);
            }
        }
    }
}
```

## Output:




## Conclusion:

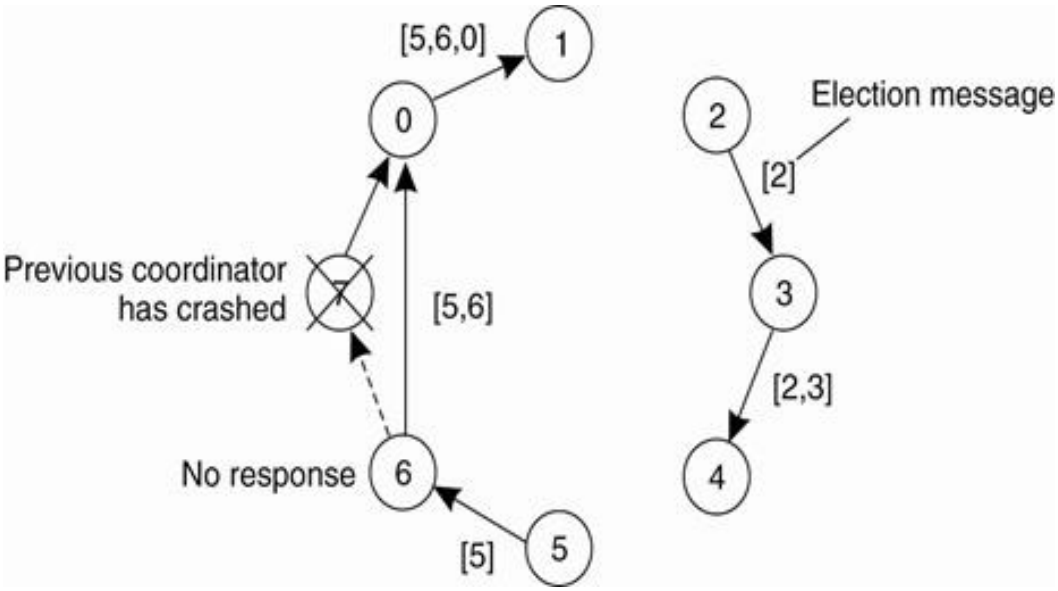
1. We successfully implemented Bully-Election Algorithm.

**PROGRAM – 7**

**Aim:** Program to implement Ring Election Algorithm.

## Theory:

Another election algorithm is based on the use of a ring, but without a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message. Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.



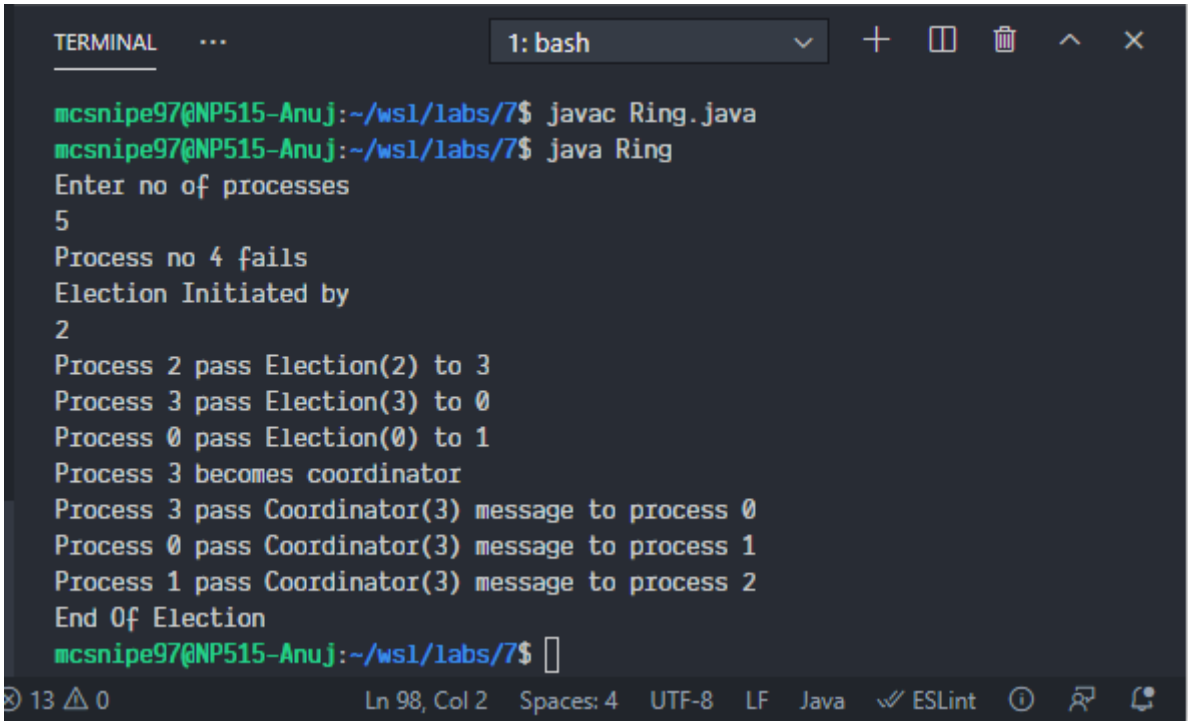
**Code:**

```
import java.util.Scanner;

class Process {
    public int id;
    public boolean active;

    public Process(int id) {
        this.id = id;
        active = true;
    }

}

public class Ring {
    int noOfProcesses;
    Process[] processes;
    Scanner sc;

    public Ring() {
        sc = new Scanner(System.in);
    }

    public void initialiseRing() {
        System.out.println("Enter no of processes");

        noOfProcesses = sc.nextInt();
        processes = new Process[noOfProcesses];
        for (int i = 0; i < processes.length; i++) {
            processes[i] = new Process(i);
        }
    }

    public int getMax() {
        int maxId = -99;
        int maxIdIndex = 0;
        for (int i = 0; i < processes.length; i++) {
            if (processes[i].active && processes[i].id > maxId) {
                maxId = processes[i].id;
                maxIdIndex = i;

            }
        }
        return maxIdIndex;
    }

    public void performElection() {

        System.out.println("Process no " + processes[getMax()].id + " fails");
```

```
        processes[getMax()].active = false;
        System.out.println("Election Initiated by");
        int initiatorProcesss = sc.nextInt();

        int prev = initiatorProcesss;
        int next = prev + 1;

        while (true) {
            if (processes[next].active) {
                System.out.println("Process " + processes[prev].id + " pass
Election(" + processes[prev].id + ") to "
                        + processes[next].id);
                prev = next;
            }

            next = (next + 1) % noOfProcesses;
            if (next == initiatorProcesss) {
                break;
            }
        }

        System.out.println("Process " + processes[getMax()].id + " becomes
coordinator");
        int coordinator = processes[getMax()].id;

        prev = coordinator;
        next = (prev + 1) % noOfProcesses;

        while (true) {

            if (processes[next].active) {
                System.out.println("Process " + processes[prev].id + " pass
Coordinator(" + coordinator
                        + ") message to process " + processes[next].id);
                prev = next;
            }
            next = (next + 1) % noOfProcesses;
            if (next == coordinator) {
                System.out.println("End Of Election ");
                break;
            }

        }

    }

    public static void main(String arg[]) {
        Ring r = new Ring();
        r.initialiseRing();
```

```
        r.performElection();
    }

}
```

## Output:




## Conclusion:

We successfully implemented Ring Election Algorithm.

# PROGRAM – 8

**Aim:** Implement Election Algorithm for Wireless Network.

## Theory:

### Election Algorithms

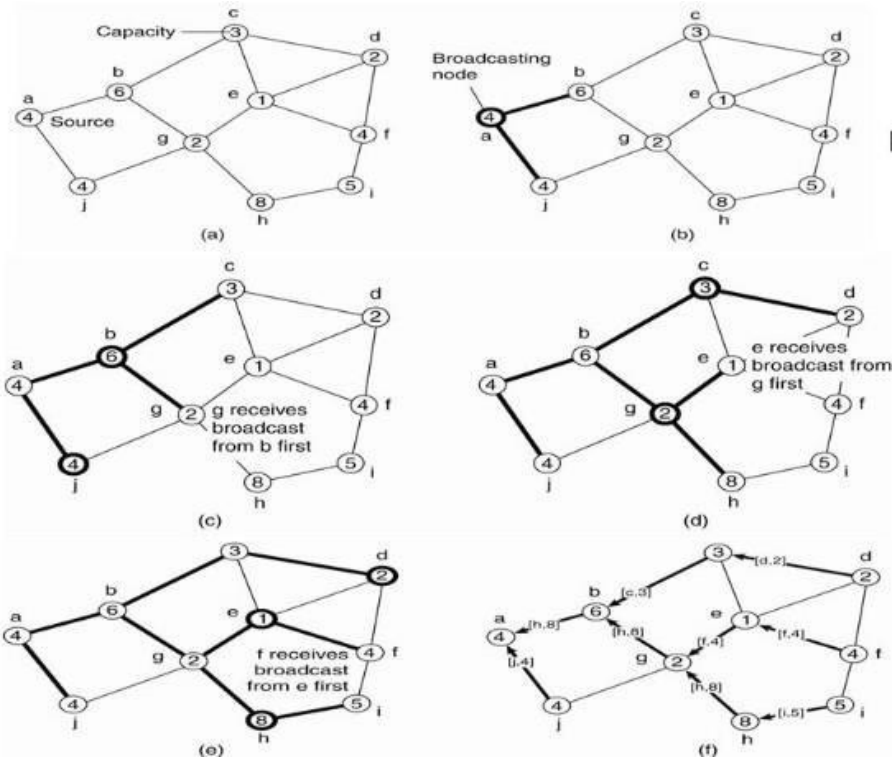
Election algorithms choose a process from group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted. Election algorithm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

Wireless Election process

1. Any node can initiate the election.
2. When a node receives its first ELECTION message, it makes the sender as its parent.
3. After this it forwards the ELECTION to all its neighbours.
4. If a node already has set its parent, it simply acknowledges.
5. If a node is a leaf it sends its own priority otherwise it waits for its children to finish.
6. When a node has collected all values, it passes it on to its parent.

Salient Points

- At each point only, the best possible candidate is passed.
- Once the source gets the results back it can select the coordinator, which it then broadcasts.
- The messages are tagged with process IDs and in case of multiple ELECTIONS, only the one from a higher pid is entertained.




## Code:

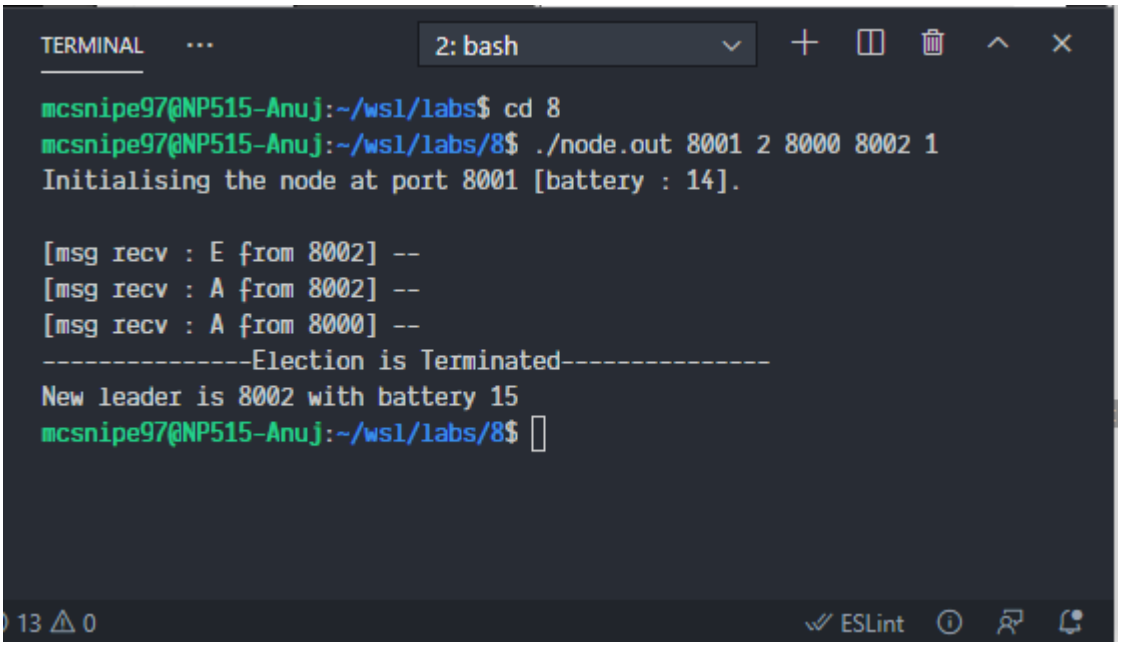
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

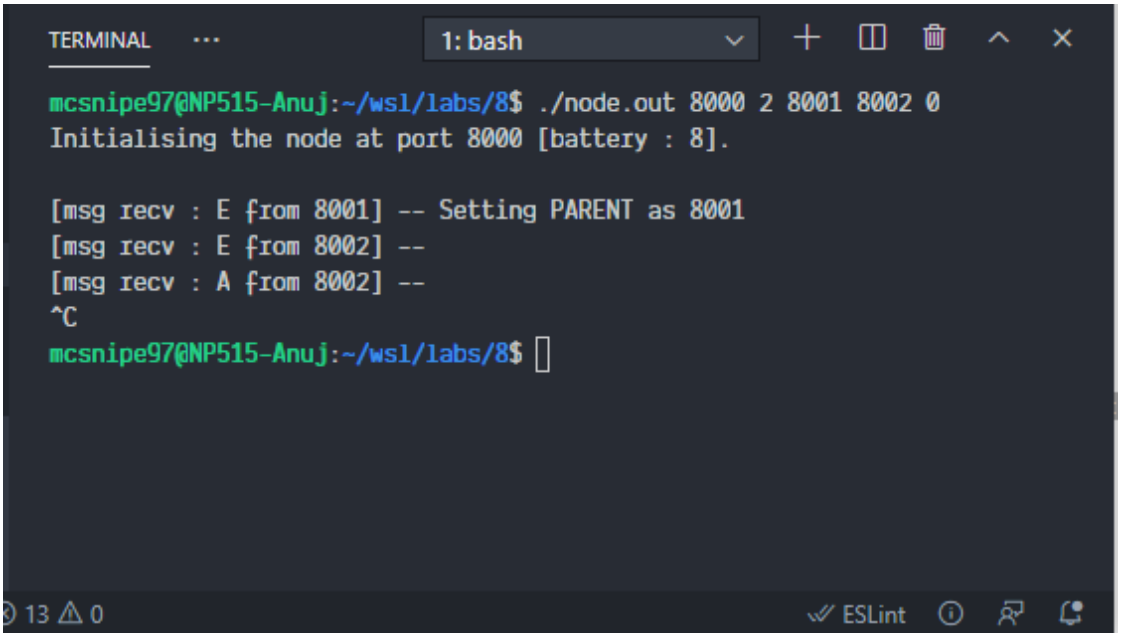
```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>

#define MAXLINE 1024
#define MAX_PROCESS 5

int ELECTION_FLAG = 0;

struct Message
{
    char type;    // E -> Election, A -> Election Ack
    int bestNode; // best node suited for Leader
    int bestNodeScore;
};

struct Candidate
{
    int port;
    int score;
};

int create_connection(int PORT)
{
    int sockfd;
    struct sockaddr_in servaddr;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }
    int optval = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval,
sizeof(int));
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);
    if (bind(sockfd, (const struct sockaddr *)&servaddr,
             sizeof(servaddr)) < 0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    return sockfd;
}

void send_message(int dest_port, int sockfd, struct Message newMsg)
{
    sleep(3);
    struct sockaddr_in client_addr;
```

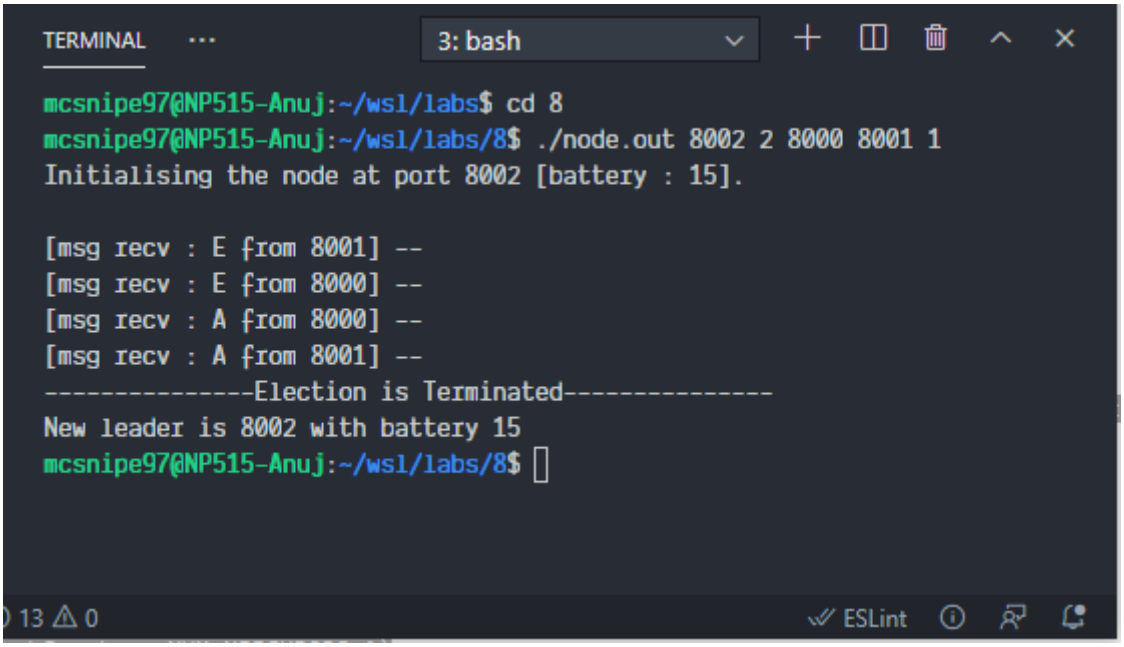
```
    memset(&client_addr, 0, sizeof(client_addr));
    client_addr.sin_family = AF_INET; // IPv4
    client_addr.sin_addr.s_addr = INADDR_ANY;
    client_addr.sin_port = htons(dest_port);
    sendto(sockfd, (struct Message *)&newMsg, (1024 + sizeof(newMsg)), 0,
(struct sockaddr *)&client_addr, sizeof(client_addr));
}

void send_elec_ack(int port, int sockfd, struct Candidate bestDescendant)
{
    struct Message newMsg;
    newMsg.type = 'A';
    newMsg.bestNode = bestDescendant.port;
    newMsg.bestNodeScore = bestDescendant.score;
    send_message(port, sockfd, newMsg);
}

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

    srand(time(NULL));
    int MY_PORT = atoi(argv[1]);
    int NUM_NEIGHBORS = atoi(argv[2]);
    int NEIGHBORS[MAX_PROCESS];
    for (int i = 0; i < NUM_NEIGHBORS; i++)
        NEIGHBORS[i] = atoi(argv[3 + i]);
    int IS_INITIATOR = atoi(argv[3 + NUM_NEIGHBORS]);
    struct Message *temp = malloc(sizeof(struct Message)), newMsg;
    int myScore = (rand() % 100 + rand() % 100) % 100;
    int sockfd = create_connection(MY_PORT);
    printf("Initialising the node at port %d [battery : %d].\n", MY_PORT,
myScore);
    struct sockaddr_in recv_client_addr, send_client_addr;
    int len = sizeof(struct sockaddr_in), n;
    int PARENT = -1, eackCount = 0;
    struct Candidate bestDescendant = {MY_PORT, myScore};
    if (IS_INITIATOR)
    {
        ELECTION_FLAG = 1;
        newMsg.type = 'E';
        for (int i = 0; i < NUM_NEIGHBORS; i++)
        {
            send_message(NEIGHBORS[i], sockfd, newMsg);
        }
    }
    while (1)
    {
        printf("\n");
        n = recvfrom(sockfd, temp, sizeof(*temp), MSG_WAITALL, (struct
sockaddr *)&recv_client_addr, &len);
        printf("[msg recv : %c from %d] -- ", temp->type,
htons(recv_client_addr.sin_port));
        if (temp->type == 'E')
        {
```

```
            if (ELECTION_FLAG)
            {
                send_elec_ack(htons(recv_client_addr.sin_port), sockfd,
bestDescendant);
                continue;
            }
            PARENT = htons(recv_client_addr.sin_port);
            printf("Setting PARENT as %d", PARENT);
            ELECTION_FLAG = 1;
            int IS_LEAF = 1;
            for (int i = 0; i < NUM_NEIGHBORS; i++)
            {
                if (NEIGHBORS[i] != PARENT)
                {
                    newMsg.type = 'E';
                    send_message(NEIGHBORS[i], sockfd, newMsg);
                    IS_LEAF = 0;
                }
            }
            if (IS_LEAF)
            {
                send_elec_ack(htons(recv_client_addr.sin_port), sockfd,
bestDescendant);
            }
        }
        else if (temp->type == 'A')
        {
            eackCount++;
            if (bestDescendant.score < temp->bestNodeScore)
            {
                bestDescendant.score = temp->bestNodeScore;
                bestDescendant.port = temp->bestNode;
            }
            if (IS_INITIATOR)
            {
                if (eackCount == NUM_NEIGHBORS)
                {
                    printf("\n---------------Election is Terminated------
---------\n");
                    printf("New leader is %d with battery %d\n",
bestDescendant.port, bestDescendant.score);
                    exit(0);
                }
            }
            else
            {
                if (eackCount == NUM_NEIGHBORS - 1)
                {
                    send_elec_ack(PARENT, sockfd, bestDescendant);
                    eackCount = 0;
                    ELECTION_FLAG = 0;
                }
            }
        } } }
```

## Output:




## Conclusion:

We successfully implemented election in wireless networks.

# PROGRAM-9

**Aim:** Implement entry eventual consistency between processes with mutual exclusive update replicated datastore.

## Theory:

Consistency models are used in distributed systems like distributed shared memory systems or distributed data stores (such as a filesystem, databases, optimistic replication systems or web caching). The system is said to support a given model if operations on memory follow specific rules. The data consistency model specifies a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of reading, writing, or updating memory will be predictable. This is different from coherence, which occurs in systems that are cached or cache-less and is consistency of data with respect to all processors. Coherence deals with maintaining a global order in which writes to a single location or single variable are seen by all processors.

Consistency deals with the ordering of operations to multiple locations with respect to all processors. There are two methods to define and categorize consistency models; issue and view.

- Issue: Issue method describes the restrictions that define how a process can issue operations.
- View: View method which defines the order of operations visible to processes.

Entry consistency

- Acquire and release are still used, and the data-store meets the following conditions:
- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
- After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

## Code:

Server:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
```

```
#include <string.h>
// #define MSG_CONFIRM 0

#define TRUE 1
#define FALSE 0
#define ML 1024
#define MPROC 32

typedef struct Resource
{
    int a;
    int b;
    int c;
    int d;
    int e;
} Resource;

void serealize(Resource S, char output[ML])
{
    sprintf(output, "MCON %d\t%d\t%d\t%d\t%d\t", S.a, S.b, S.c, S.d, S.e);
}

Resource unserealize(char input[ML])
{
    char temp[ML];
    int ix = 0, itr = 5;
    Resource S;
    for (itr; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.a = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.b = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.c = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.d = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
```

```
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.e = atoi(temp);
    ix = 0;
    return S;
}

int connect_to_port(int connect_to)
{
    int sock_id;
    int opt = 1;
    struct sockaddr_in server;
    if ((sock_id = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("unable to create a socket");
        exit(EXIT_FAILURE);
    }
    setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, (const void *)&opt,
   sizeof(int));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(connect_to);

    if (bind(sock_id, (const struct sockaddr *)&server, sizeof(server)) < 0)
    {
        perror("unable to bind to port");
        exit(EXIT_FAILURE);
    }
    return sock_id;
}

void send_to_id(int to, int from, char message[ML])
{
    struct sockaddr_in cl;
    memset(&cl, 0, sizeof(cl));

    cl.sin_family = AF_INET;
    cl.sin_addr.s_addr = INADDR_ANY;
    cl.sin_port = htons(to);

    sendto(from, (const char *)message, strlen(message), MSG_CONFIRM, (const
   struct sockaddr *)&cl, sizeof(cl));
}

void make_consistent(int from, int procs[], int n_procs, Resource S)
{
    char message[ML];
    int i;
    serealize(S, message);
    for (i = 0; i < n_procs; i++)
        send_to_id(procs[i], from, message);
```

```
}

int main(int argc, char *argv[])
{
    int self = atoi(argv[1]);
    int n_procs = atoi(argv[2]);
    int itr, ix = 0;
    int procs[MPROC];
    int key_avail = 1;
    int dest;

    int sock_id, len, n;
    char buffer[ML], msg[ML];
    char flag[256], p_id[256];

    struct sockaddr_in from;

    Resource S = {0, 0, 0, 0, 0};
    for (itr = 0; itr < n_procs; itr++)
        procs[itr] = atoi(argv[3 + itr]);
    printf("Creating node at %d\n", self);
    sock_id = connect_to_port(self);

    while (TRUE)
    {
        memset(&from, 0, sizeof(from));
        n = recvfrom(sock_id, (char *)buffer, ML, MSG_WAITALL, (struct sockaddr
   *)&from, &len);
        buffer[n] = '\0';
        printf("Recieved: %s\n", buffer);

        for (itr = 0; itr < 4; itr++)
            flag[itr] = buffer[itr];
        flag[itr] = '\0';
        printf("Extracted flag %s\n", flag);

        // process asks for key
        if (strcmp(flag, "KEYR") == 0)
        {
            ix = 0;
            for (itr = 5; itr < 9; itr++)
                p_id[ix++] = buffer[itr];
            p_id[ix] = '\0';
            dest = atoi(p_id);
            printf("Extracted dest %d\n", dest);
            if (key_avail)
            {
                send_to_id(dest, sock_id, "PASS");
                key_avail = 0;
            }

            else
```

```
            {
                send_to_id(dest, sock_id, "WAIT");
            }
        }
        // process releases key
        else if (strcmp(flag, "DONE") == 0)
        {
            printf("Key released\n");
            S = unserealize(buffer);
            key_avail = 1;
        }
        // process calls for consistency
        else if (strcmp(flag, "MCON") == 0)
        {
            printf("Forcing consistency \n");
            make_consistent(sock_id, procs, n_procs, S);
            for (itr = 5; itr < 9; itr++)
                p_id[5 - itr] = buffer[itr];
            p_id[5 - itr] = '\0';
            dest = atoi(p_id);
            send_to_id(dest, sock_id, "CNOK");
        }
    }
}
```

Client:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <string.h>
// #define MSG_CONFIRM 0

#define TRUE 1
#define FALSE 0
#define ML 1024
#define MPROC 32

typedef struct Resource
{
    int a;
    int b;
    int c;
    int d;
    int e;
} Resource;
```

```
void serealize(Resource S, char output[ML])
{
    sprintf(output, "DONE %d\t%d\t%d\t%d\t%d\t", S.a, S.b, S.c, S.d, S.e);
}

Resource unserealize(char input[ML])
{
    char temp[ML];
    int ix = 0, itr = 5;
    Resource S;
    for (itr; input[itr] != '\t'; itr += 1)
    {
        printf("%d %c\n", itr, input[itr]);
        temp[ix++] = input[itr];
    }
    temp[ix] = '\0';
    S.a = atoi(temp);
    ix = 0;
    printf("here\n");
    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.b = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.c = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.d = atoi(temp);
    ix = 0;

    for (itr = itr + 1; input[itr] != '\t'; itr += 1)
        temp[ix++] = input[itr];
    temp[ix] = '\0';
    S.e = atoi(temp);
    ix = 0;
    return S;
}

int connect_to_port(int connect_to)
{
    int sock_id;
    int opt = 1;
    struct sockaddr_in server;
    if ((sock_id = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
```

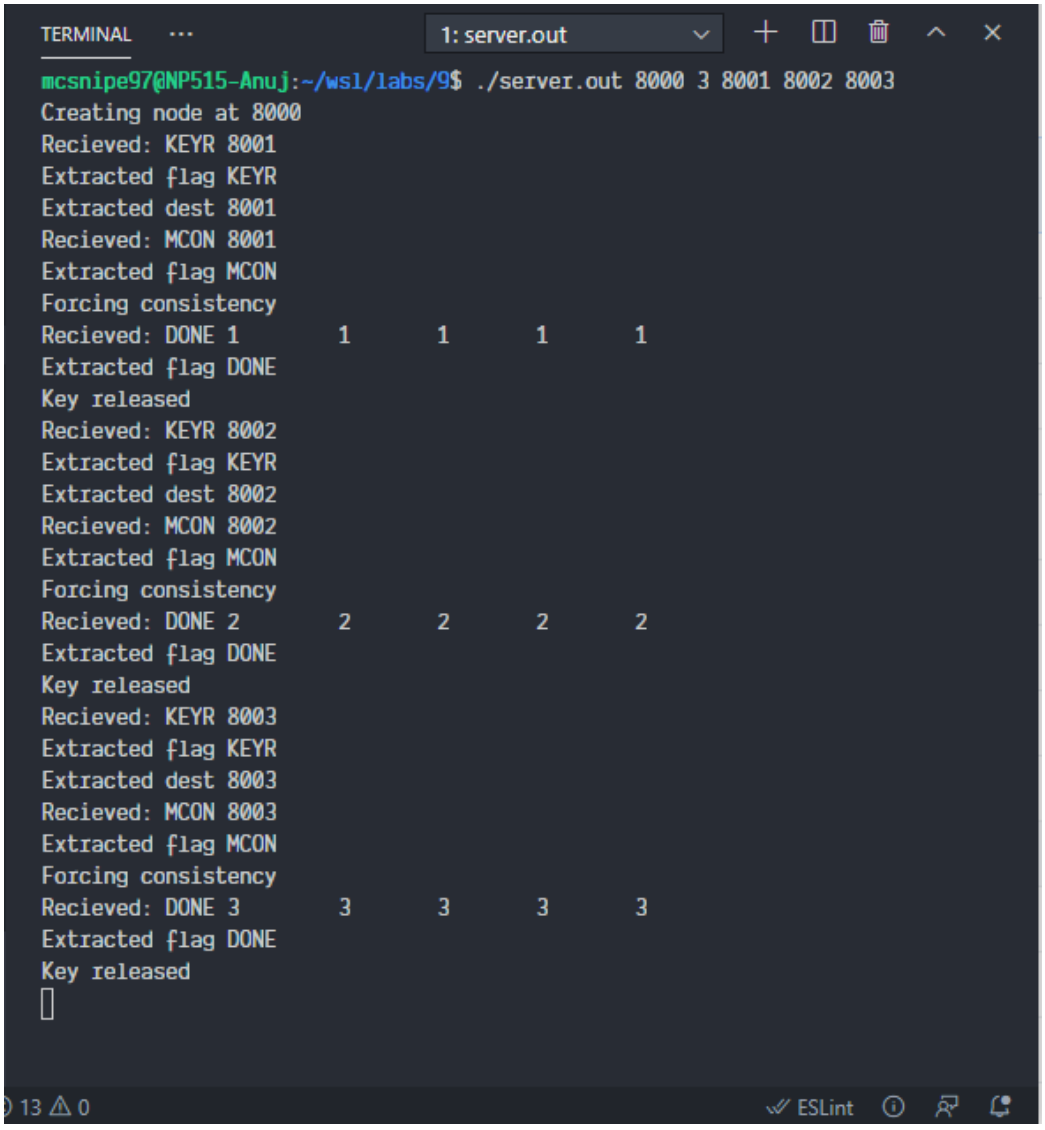
```
    {
        perror("unable to create a socket");
        exit(EXIT_FAILURE);
    }
    setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, (const void *)&opt,
   sizeof(int));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(connect_to);

    if (bind(sock_id, (const struct sockaddr *)&server, sizeof(server)) < 0)
    {
        perror("unable to bind to port");
        exit(EXIT_FAILURE);
    }
    return sock_id;
}

void send_to_id(int to, int from, char message[ML])
{
    struct sockaddr_in cl;
    memset(&cl, 0, sizeof(cl));

    cl.sin_family = AF_INET;
    cl.sin_addr.s_addr = INADDR_ANY;
    cl.sin_port = htons(to);

    sendto(from, (const char *)message, strlen(message), MSG_CONFIRM, (const
   struct sockaddr *)&cl, sizeof(cl));
}

void request_key(int server, int sock_id, int a)
{
    char msg[256];
    sprintf(msg, "KEYR %d", a);
    send_to_id(server, sock_id, msg);
}

int main(int argc, char *argv[])
{
    int self = atoi(argv[1]);
    int server = atoi(argv[2]);
    int start = atoi(argv[3]);
    int udelay = atoi(argv[4]);
    int itr;
    int dest;
    int key = 0;

    int sock_id, len, n;
    char buffer[ML], msg[ML];
    char flag[256], p_id[256];
```

```
 struct sockaddr_in from;

 Resource S = {0, 0, 0, 0, 0};
 printf("Creating node at %d\n", self);
 sock_id = connect_to_port(self);

 if (start)
 {
     request_key(server, sock_id, self);
 }
 else
 {
     sleep(udelay);
     request_key(server, sock_id, self);
 }

 while (TRUE)
 {
     // sleep(udelay);
     memset(&from, 0, sizeof(from));
     n = recvfrom(sock_id, (char *)buffer, ML, MSG_WAITALL, (struct sockaddr
*)&from, &len);
     buffer[n] = '\0';
     printf("Recieved: %s\n", buffer);

     for (itr = 0; itr < 4; itr++)
         flag[itr] = buffer[itr];
     flag[itr] = '\0';
     printf("Extracted flag %s\n", flag);

     // server denies key
     if (strcmp(flag, "WAIT") == 0)
     {
         sleep(udelay);
         request_key(server, sock_id, self);
     }
     // process releases key
     else if (strcmp(flag, "PASS") == 0)
     {
         printf("Key recieved\n");
         key = 1;
         sprintf(msg, "MCON %d", self);
         send_to_id(server, sock_id, msg);
     }
     // process calls for consistency
     else if (strcmp(flag, "MCON") == 0)
     {
         printf("Pulling data from server before update\n");
         S = unserealize(buffer);
         printf("Pulled file\n %5d, %5d %5d %5d %5d\n", S.a, S.b, S.c, S.d,
S.e);
```

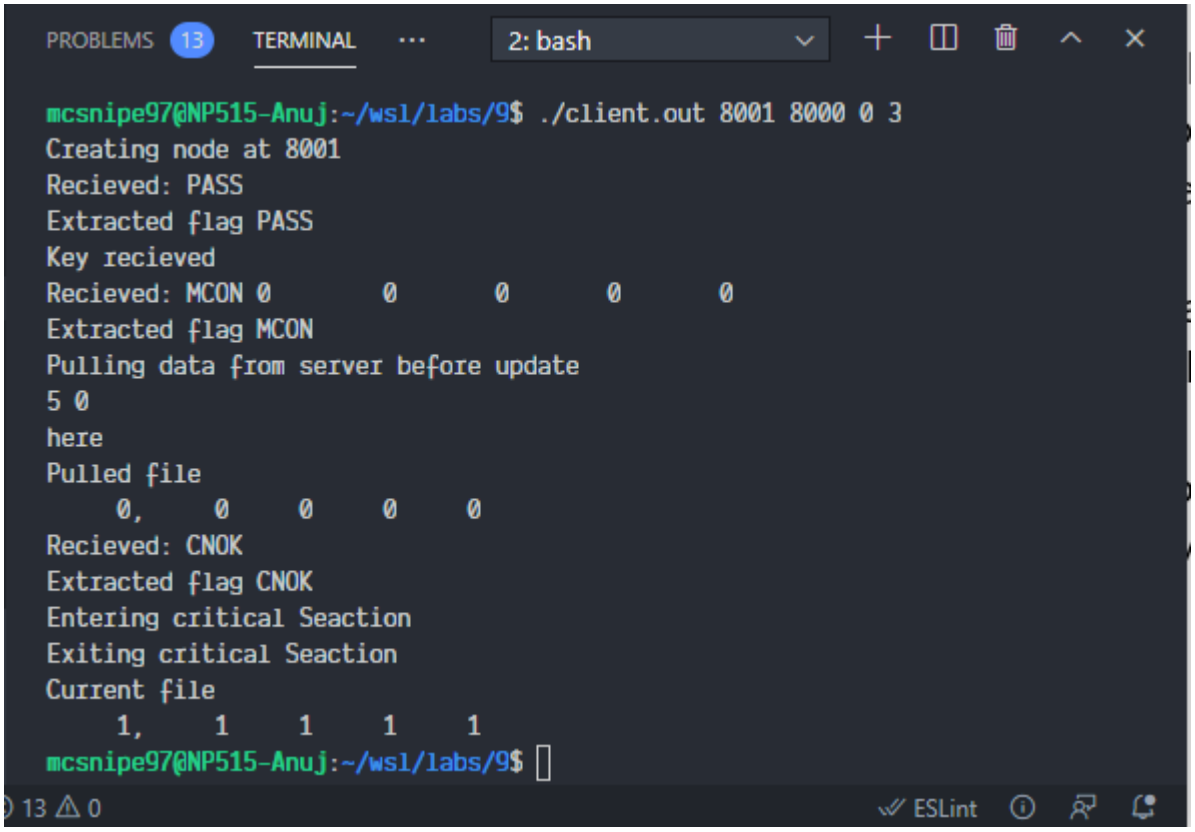
```
        }

        else if (strcmp(flag, "CNOK") == 0 && key)
        {
            printf("Entering critical Seaction\n");
            S.a++;
            S.b++;
            S.c++;
            S.d++;
            S.e++;
            printf("Exiting critical Seaction\n");
            printf("Current file\n %5d, %5d %5d %5d %5d\n", S.a, S.b, S.c, S.d,
    S.e);
            serealize(S, msg);
            send_to_id(server, sock_id, msg);
            exit(EXIT_SUCCESS);
        }
     }
     return 0;
}
```

**Output:**

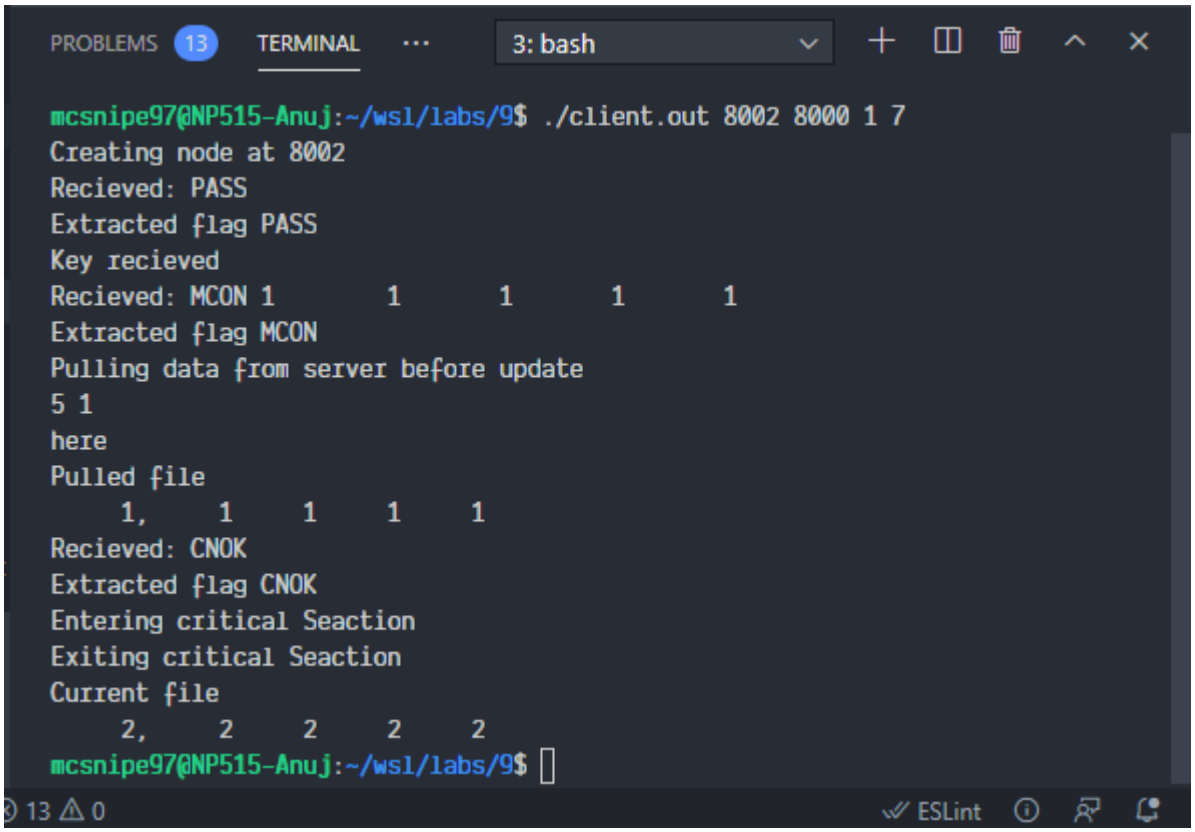
Server:



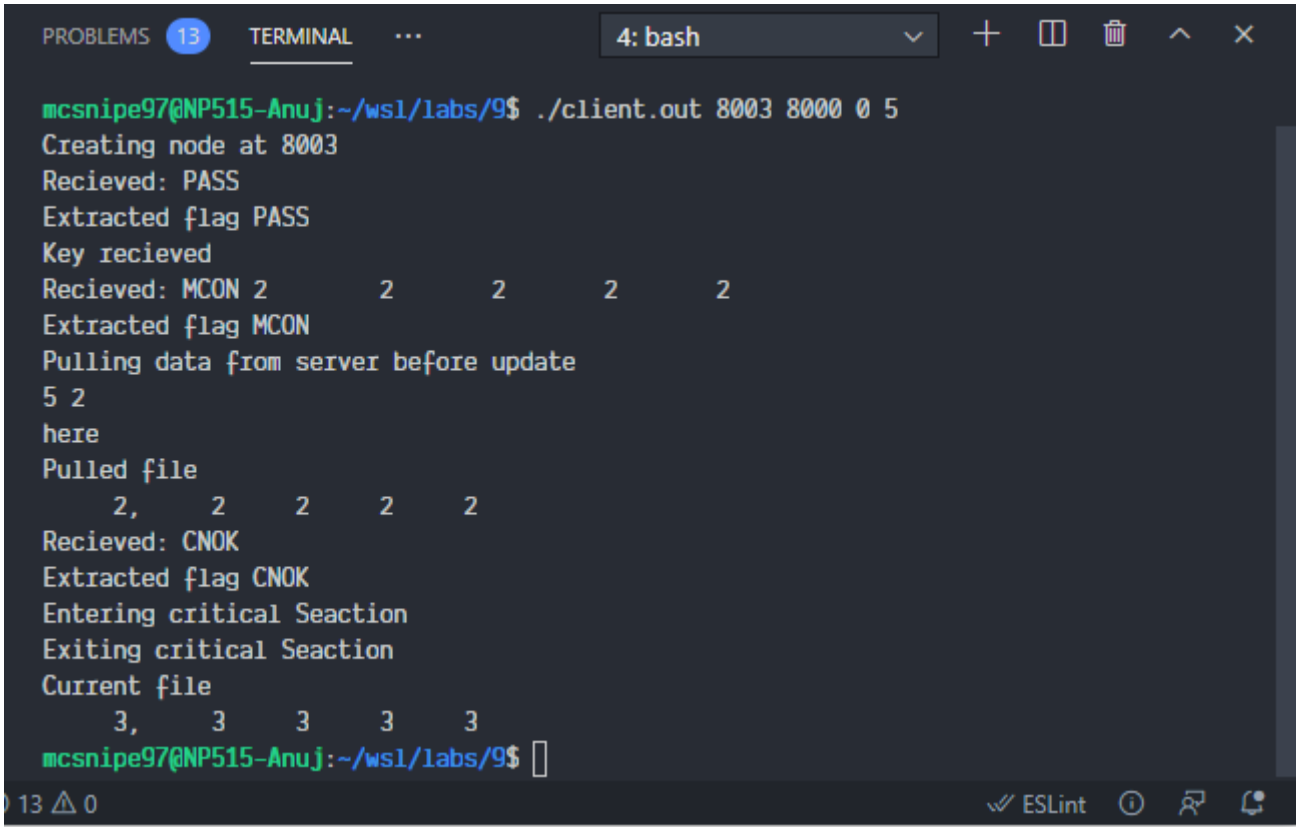
Client1:




Client2:



Client3:




**Conclusion:** We successfully implemented entry eventual consistency between processes with mutual exclusive update replicated datastore.

# PROGRAM – 10

**Aim:** To implement 2-Phase Commit client-server.

## Theory:

In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

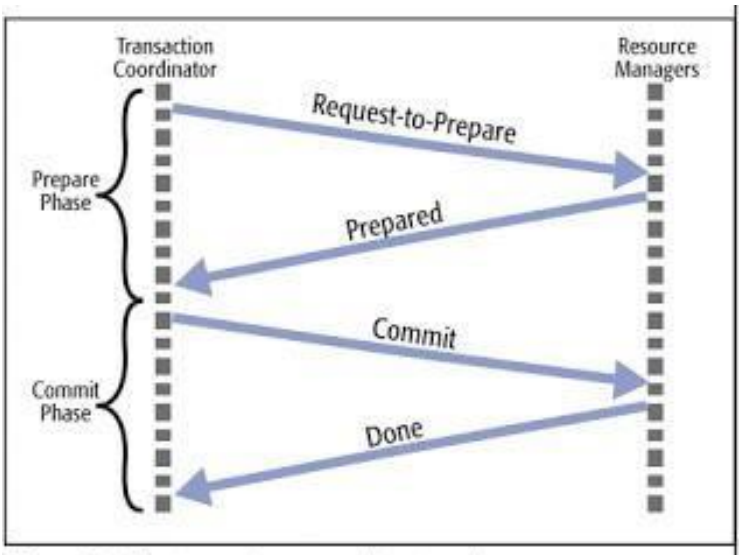


**Figure 1** • The two-phase commit protocol

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows –

## Phase 1: Prepare Phase

- After each slave has locally completed its transaction, it sends a "DONE" message to the controlling site. When the controlling site has received "DONE" message from all slaves, it sends a "Prepare" message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a "Ready" message.
- A slave that does not want to commit sends a "Not Ready" message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

## Phase 2: Commit/Abort Phase

- After the controlling site has received "Ready" message from all the slaves –
  - The controlling site sends a "Global Commit" message to the slaves.
  - The slaves apply the transaction and send a "Commit ACK" message to the controlling site.
  - When the controlling site receives "Commit ACK" message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first "Not Ready" message from any slave –
  - The controlling site sends a "Global Abort" message to the slaves.
  - The slaves abort the transaction and send a "Abort ACK" message to the controlling site.
  - When the controlling site receives "Abort ACK" message from all the slaves, it considers the transaction as aborted.

## Code:

Server:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <sys/poll.h>

#define MAXLINE 1024
#define MAX_PROCESS 5

int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

struct Message
{
    char type; // V -> Vote Request, c -> Commit OK, a -> Commit Abort, C ->
Global commit, A -> Global abort
};

int create_connection(int PORT)
{
    int sockfd;
    struct sockaddr_in servaddr;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }
    int optval = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval,
sizeof(int));
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);
    if (bind(sockfd, (const struct sockaddr *)&servaddr,
             sizeof(servaddr)) < 0)
    {
        perror("bind failed");
```

```
        exit(EXIT_FAILURE);
    }
    return sockfd;
}

void send_message(int dest_port, int sockfd, struct Message newMsg)
{
    struct sockaddr_in client_addr;
    memset(&client_addr, 0, sizeof(client_addr));
    client_addr.sin_family = AF_INET; // IPv4
    client_addr.sin_addr.s_addr = INADDR_ANY;
    client_addr.sin_port = htons(dest_port);
    sendto(sockfd, (struct Message *)&newMsg, (1024 + sizeof(newMsg)), 0, (struct
sockaddr *)&client_addr, sizeof(client_addr));
}

void _2pc(int sockfd, int OTHER_PROCESS_PORTS[], int NUM_PROCESSES, int MY_PORT)
{
    printf("Starting the Voting Phase \n");
    struct sockaddr_in recv_client_addr[NUM_PROCESSES], send_client_addr;
    struct Message *temp = malloc(sizeof(struct Message));
    int len = sizeof(struct sockaddr_in), n, i, flag = 1;
    struct Message newMsg;

    newMsg.type = 'V';

    for (i = 0; i < NUM_PROCESSES; i++)
    {
        send_message(OTHER_PROCESS_PORTS[i], sockfd, newMsg);
    }
    int commit_flag = 1;
    for (i = 0; i < NUM_PROCESSES; i++)
    {
        n = recvfrom(sockfd, temp, sizeof(*temp), MSG_WAITALL, (struct sockaddr
*)&recv_client_addr[i], &len);
        if (temp->type == 'c')
        {
            commit_flag = commit_flag && 1;
        }
        else if (temp->type == 'a')
        {
            commit_flag = commit_flag && 0;
        }
    }
    printf("Starting the Decision Phase \n");
    char final_decision = (commit_flag ? 'C' : 'A');
    for (i = 0; i < NUM_PROCESSES; i++)
    {
        newMsg.type = final_decision;
        send_message(OTHER_PROCESS_PORTS[i], sockfd, newMsg);
    }
    //     return flag;
}
```

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

    int MY_PORT = atoi(argv[1]);
    int NUM_PROCESSES = atoi(argv[2]);
    int OTHER_PROCESS_PORTS[MAX_PROCESS];
    int i;
    for (i = 0; i < NUM_PROCESSES; i++)
    {
        OTHER_PROCESS_PORTS[i] = atoi(argv[3 + i]);
    }

    struct Message *temp = malloc(sizeof(struct Message));

    printf("Initialising the time server at port %d.\n", MY_PORT);
    int sockfd = create_connection(MY_PORT);

    struct sockaddr_in recv_client_addr, send_client_addr;
    _2pc(sockfd, OTHER_PROCESS_PORTS, NUM_PROCESSES, MY_PORT);
}
```

Client:

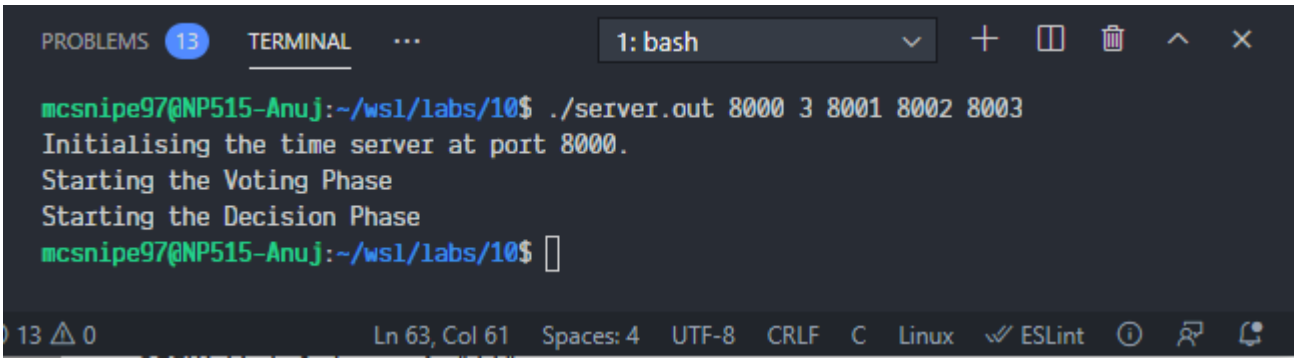
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>

#define MAXLINE 1024
#define MAX_PROCESS 5

struct Message
{
    char type; // V -> Vote Request, c -> Commit OK, a -> Commit Abort, C ->
Global commit, A -> Global abort
};

int create_connection(int PORT)
{
    int sockfd;
    struct sockaddr_in servaddr;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }
```

```
    int optval = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval,
sizeof(int));
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);
    if (bind(sockfd, (const struct sockaddr *)&servaddr,
             sizeof(servaddr)) < 0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    return sockfd;
}

void send_message(int dest_port, int sockfd, struct Message newMsg)
{
    struct sockaddr_in client_addr;
    memset(&client_addr, 0, sizeof(client_addr));
    client_addr.sin_family = AF_INET; // IPv4
    client_addr.sin_addr.s_addr = INADDR_ANY;
    client_addr.sin_port = htons(dest_port);
    sendto(sockfd, (struct Message *)&newMsg, (1024 + sizeof(newMsg)), 0, (struct
sockaddr *)&client_addr, sizeof(client_addr));
}

int main(int argc, char *argv[])
{
    srand(time(NULL));
    int MY_PORT = atoi(argv[1]);
    int COORDINATOR_PORT = atoi(argv[2]);
    double random_number = (double)rand() / (double)((unsigned)RAND_MAX + 1);
    char my_status = (random_number >= 0.4 ? 'a' : 'c');
    struct Message *temp = malloc(sizeof(struct Message));

    printf("Initialising the client at port %d\n", MY_PORT);
    int sockfd = create_connection(MY_PORT);

    struct sockaddr_in recv_client_addr, send_client_addr;
    int len = sizeof(struct sockaddr_in), n, i, flag = 1;

    while (1)
    {
        n = recvfrom(sockfd, temp, sizeof(*temp), MSG_WAITALL, (struct sockaddr
*)&recv_client_addr, &len);
        struct Message newMsg;
        if (temp->type == 'V')
        {
            printf("COORDINATOR sent a VOTE REQUEST.");
            newMsg.type = my_status;
            printf(" Local status : %s\n", (random_number >= 0.6 ? "abort" :
"commit"));
```

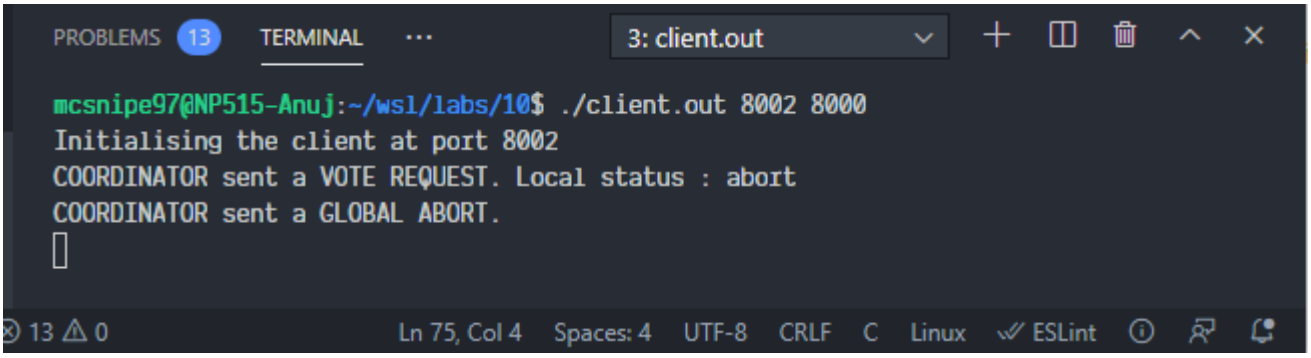
```
            send_message(COORDINATOR_PORT, sockfd, newMsg);
        }
        else if (temp->type == 'C')
        {
            printf("COORDINATOR sent a GLOBAL COMMIT.\n");
        }
        else if (temp->type == 'A')
        {
            printf("COORDINATOR sent a GLOBAL ABORT.\n");
        }
    }
}
```

## Output:

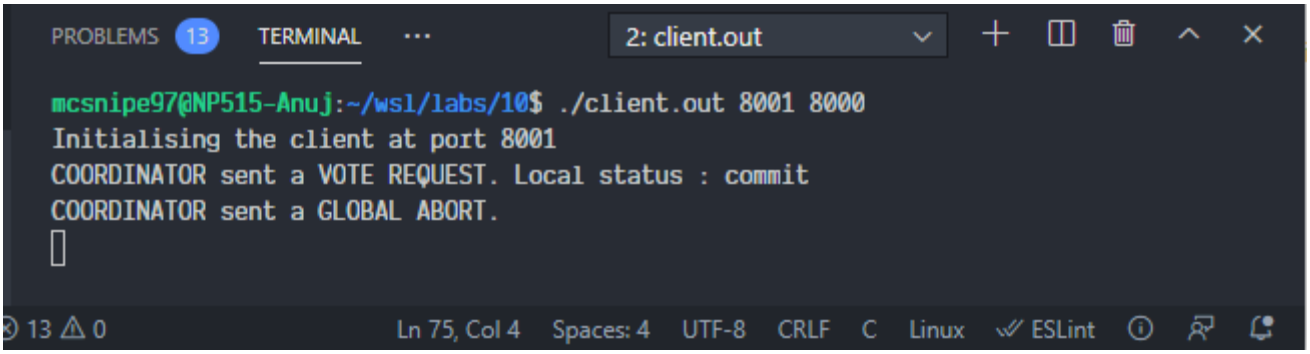
Server:



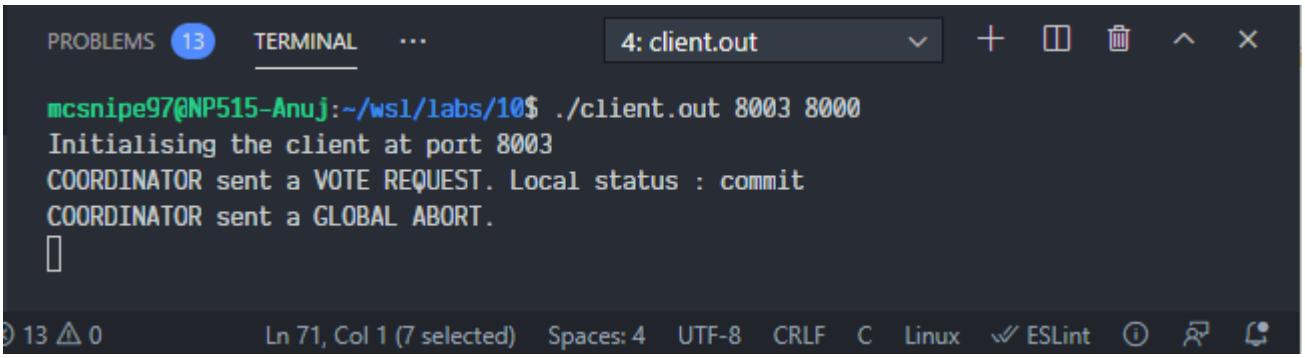

Client1:




Client2:



Client3:




**Conclusion:** We successfully implemented 2-phase commit.