Prac1  
import time  
import threading  
import json  
from queue import Queue  
  
# Simulated network channel  
network\_channel = Queue()  
  
# Server class with RPC methods  
class RPCServer:  
    def \_\_init\_\_(self, incoming\_queue):  
        self.queue = incoming\_queue  
        self.methods = {  
            "add": self.add,  
            "multiply": self.multiply,  
        }  
  
    def add(self, a, b):  
        return a + b  
  
    def multiply(self, a, b):  
        return a \* b  
  
    def serve\_forever(self):  
        print("[Server] Started listening for RPC calls...")  
        while True:  
            if not self.queue.empty():  
                raw\_request = self.queue.get()  
                request = json.loads(raw\_request["data"])  
                print(f"[Server] Received RPC request: {request}")  
  
                method\_name = request.get("method")  
                params = request.get("params", [])  
                response\_queue = raw\_request["response\_queue"]  
  
                if method\_name in self.methods:  
                    try:  
                        result = self.methods[method\_name](\*params)  
                        response = {"status": "success", "result": result}  
                    except Exception as e:  
                        response = {"status": "error", "error": str(e)}  
                else:  
                    response = {"status": "error", "error": "Method not found"}  
  
                response\_queue.put(json.dumps(response))  
  
# Client class  
class RPCClient:  
    def \_\_init\_\_(self, server\_queue):  
        self.server\_queue = server\_queue  
        self.response\_queue = Queue()  
  
    def call(self, method\_name, \*args):  
        request = {  
            "data": json.dumps({  
                "method": method\_name,  
                "params": args  
            }),  
            "response\_queue": self.response\_queue  
        }  
        self.server\_queue.put(request)  
        print(f"[Client] Sent RPC request for method '{method\_name}' with params {args}")  
        response = json.loads(self.response\_queue.get())  
        return response  
  
# Initialize and run server  
server = RPCServer(network\_channel)  
server\_thread = threading.Thread(target=server.serve\_forever, daemon=True)  
server\_thread.start()  
  
# Create client  
client = RPCClient(network\_channel)  
  
# Make RPC calls  
response1 = client.call("add", 10, 5)  
print("RPC Response 1:", response1)  
  
response2 = client.call("multiply", 3, 7)  
print("RPC Response 2:", response2)  
  
response3 = client.call("unknown\_method", 3, 7)  
print("RPC Response 3 (error):", response3)

Prac2  
class MyThread extends Thread {  
    public void run() {  
        for (int i = 1; i <= 5; i++) {  
            System.out.println(Thread.currentThread().getName() + " - Value: " + i);  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                System.out.println(e);  
            }  
        }  
    }  
}  
  
class MyRunnable implements Runnable {  
    public void run() {  
        for (int i = 1; i <= 5; i++) {  
            System.out.println(Thread.currentThread().getName() + " - Value: " + i);  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                System.out.println(e);  
            }  
        }  
    }  
}  
  
public class MultiThreadDemo {  
    public static void main(String[] args) {  
        MyThread thread1 = new MyThread();  
        Thread thread2 = new Thread(new MyRunnable());  
  
        thread1.setName("Thread-1");  
        thread2.setName("Thread-2");  
  
        thread1.start();  
        thread2.start();  
    }  
}

Prac3  
import socket

localIP = "127.0.0.1"

localPort = 20001

bufferSize = 1024

# Create a UDP socket at server side

UDPServerSocket = socket.socket(family=socket.AF\_INET, type=socket.SOCK\_DGRAM)

# Bind to address and ip

UDPServerSocket.bind((localIP, localPort))

print("UDP server up and listening")

# Listen for incoming datagrams

clients = {} # Dictionary to store client addresses

while(True):

bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)

message = bytesAddressPair[0]

address = bytesAddressPair[1]

# Store client address

clients[address] = True

clientMsg = "Message from Client {}:{}: {}".format(address[0], address[1], message.decode())

clientIP = "Client IP Address: {}".format(address)

print(clientMsg)

print(clientIP)

# Get response from server to send to all clients

msgFromServer = input("Enter your message for client {}:{}: ".format(address[0], address[1]))

bytesToSend = str.encode(msgFromServer)

# Send reply to all clients

for client\_address in clients:

UDPServerSocket.sendto(bytesToSend, client\_address)

#client1

import socket

import time

while True:

msgFromClient = input("Enter your message: ")

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

serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

# Create a UDP socket at client side

UDPClientSocket = socket.socket(family=socket.AF\_INET, type=socket.SOCK\_DGRAM)

# Send to server using created UDP socket

UDPClientSocket.sendto(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.recvfrom(bufferSize)

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

time.sleep(1)

print(msg)

#client2

import socket

import time

while True:

msgFromClient = input("Enter your message: ")

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

serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

# Create a UDP socket at client side

UDPClientSocket = socket.socket(family=socket.AF\_INET, type=socket.SOCK\_DGRAM)

# Send to server using created UDP socket

UDPClientSocket.sendto(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.recvfrom(bufferSize)

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

time.sleep(1)

print(msg)

#client3

import socket

import time

while True:

msgFromClient = input("Enter your message: ")

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

serverAddressPort = ("127.0.0.1", 20001)

bufferSize = 1024

# Create a UDP socket at client side

UDPClientSocket = socket.socket(family=socket.AF\_INET, type=socket.SOCK\_DGRAM)

# Send to server using created UDP socket

UDPClientSocket.sendto(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.recvfrom(bufferSize)

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

time.sleep(1)

print(msg)

prac4  
import itertools

class LoadBalancer:

def \_\_init\_\_(self, servers): # Fixed: Added indentation (4 spaces)

self.servers = itertools.cycle(servers)

def get\_server(self): # Fixed: Added indentation (4 spaces)

return next(self.servers)

if \_\_name\_\_ == '\_\_main\_\_':

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

lb = LoadBalancer(servers)

# simulate 10 requests

for i in range(10):

server = lb.get\_server()

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

prac5  
class LamportClock:

def \_\_init\_\_(self):

self.time = 0

def tick(self):

"""Increment the clock for an internal event."""

self.time += 1

def send\_event(self):

"""Increment the clock and return the timestamp for a message being sent."""

self.time += 1

return self.time

def receive\_event(self, received\_time):

"""Update the clock when receiving a message."""

self.time = max(self.time, received\_time) + 1

def get\_time(self):

return self.time

# Example usage

def example():

process\_A = LamportClock()

process\_B = LamportClock()

# Process A performs an event

process\_A.tick()

print("Process A time:", process\_A.get\_time())

# Process A sends a message to Process B

sent\_time = process\_A.send\_event()

print("Process A sent event at time:", sent\_time)

# Process B receives the message

process\_B.receive\_event(sent\_time)

print("Process B received event at time:", process\_B.get\_time())

# Process B performs another event

process\_B.tick()

print("Process B time after internal event:", process\_B.get\_time())

if \_\_name\_\_ == "\_\_main\_\_":

example()

prac6  
class Process:

def \_\_init\_\_(self, pid, total\_processes):

self.pid = pid

self.total\_processes = total\_processes

self.active = True # All processes are initially active

self.coordinator = None # Store current coordinator

def send\_election\_message(self, other):

print(f"Process {self.pid} sends election message to Process {other.pid}")

def receive\_election\_message(self, sender):

print(f"Process {self.pid} received election message from Process {sender.pid}")

class BullyElection:

def \_\_init\_\_(self, total\_processes):

self.processes = [Process(i, total\_processes) for i in range(1, total\_processes + 1)]

def fail\_process(self, pid):

print(f"\nProcess {pid} fails.\n")

self.processes[pid - 1].active = False # Process fails

def start\_election(self, initiator):

print(f"\nProcess {initiator} starts an election.")

initiator\_index = initiator - 1

higher\_processes = [p for p in self.processes if p.pid > initiator and p.active]

if not higher\_processes:

print(f"\nProcess {initiator} wins the election and becomes the coordinator.")

self.processes[initiator\_index].coordinator = initiator

self.announce\_coordinator(initiator)

return

# Send election messages to all higher-numbered active processes

for process in higher\_processes:

self.processes[initiator\_index].send\_election\_message(process)

process.receive\_election\_message(self.processes[initiator\_index])

# The highest active process responds and starts its own election

highest\_pid = max(p.pid for p in higher\_processes)

self.start\_election(highest\_pid) # Start election from the highest process

def announce\_coordinator(self, coordinator):

print(f"\nProcess {coordinator} announces itself as the coordinator.")

for process in self.processes:

if process.active and process.pid != coordinator:

print(f"Process {coordinator} sends Coordinator message to Process {process.pid}")

# Run the Bully Algorithm

bully = BullyElection(5)

bully.fail\_process(5) # Simulating failure of highest process

bully.start\_election(2) # Initiating election from Process 2