Group communication

Client:

```
import socket
import struct
MULTICAST_GROUP = '224.3.29.71'
MULTICAST_PORT = 10000
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
ttl = struct.pack('b', 1)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)
while True:
   message = input("Enter message to send (type 'exit' to quit): ")
   if message.lower() == 'exit':
      break
   sock.sendto(message.encode(), (MULTICAST_GROUP, MULTICAST_PORT))
sock.close()
```

SERver

```
import socket
import struct
import threading  # Import the threading module
MULTICAST_GROUP = '224.3.29.71'
MULTICAST_PORT = 10000
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_address = ('', MULTICAST_PORT)
sock.bind(server_address)
group = socket.inet_aton(MULTICAST_GROUP)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
def receive_messages():
   while True:
      data, address = sock.recvfrom(1024)
      print("Received message from {}: {}".format(address, data.decode()))

 # Start a thread to receive messages
receive_thread = threading.Thread(target=receive_messages)
receive_thread.start()
# # Wait for the thread to finish
receive_thread.join()
# # Close the socket
sock.close()
```

Interprocess:- write

```
import java.io.*;
public class write {
   public static void main(String[] args) {
   try{
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
      PrintWriter writer = new PrintWriter(new FileWriter("data.txt"));
      String input;
         while ((input = reader.readLine()) != null) {
            writer.println(input);
            writer.flush();
         }
         reader.close();
         writer.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

read

```
import java.io.*;
public class read {
  public static void main(String[] args) {
   try {
      BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
      String input;
      while ((input = reader.readLine()) != null) {
         System.out.println("Received: " + input);

      }
      reader.close();
   } catch (IOException e) {
      e.printStackTrace();
   }
 }
}
```

Election:

```
class Pro:
    def __init__(self, id):
        self.id = id
        self.act = True
class GFG:
    def __init__(self):
        self.TotalProcess = 0
        self.process = []
    def initialiseGFG(self):
        print("No of processes 5")
        self.TotalProcess = 5
        self.process = [Pro(i) for i in range(self.TotalProcess)]
    def Election(self):
        max_id_process_index = self.FetchMaximumActive()
        print("Process no " + str(self.process[max_id_process_index].id) + " fails")
        self.process[max_id_process_index].act = False
        initialized_process = self.FetchMaximumActive()
        print("Election Initiated by " + str(self.process[initialized_process].id))
        # If there are no active processes, end the election
        if initialized_process == -1:
            print("No active processes. End of Election.")
            return
        for newer in range(initialized_process + 1, initialized_process + self.TotalProcess):
            newer %= self.TotalProcess
            if self.process[newer].act:
                print("Process " + str(self.process[initialized_process].id) + " pass Election(" +
str(self.process[initialized_process].id) + ") to " + str(self.process[newer].id))
                if self.process[newer].id > self.process[initialized_process].id:
                    print("Process " + str(self.process[newer].id) + " responds 'OK'")
                else:
```

```
                print("Process " + str(self.process[newer].id) + " doesn't respond")
        coord = self.FetchMaximumActive()
        if coord != -1:
            print("Process " + str(self.process[coord].id) + " becomes coordinator")
            old = coord
            newer = (old + 1) % self.TotalProcess
            while True:
                if self.process[newer].act:
                    print("Process " + str(self.process[old].id) + " pass Coordinator(" + str(coord) + ") message
to process " + str(self.process[newer].id))
                    old = newer
                newer = (newer + 1) % self.TotalProcess
                if newer == coord:
                    print("End Of Election ")
                    break
        else:
            print("No active processes. End of Election.")
    def FetchMaximumActive(self):
        max_id = -1
        max_id_process_index = -1
        for i in range(self.TotalProcess):
            if self.process[i].act and self.process[i].id > max_id:
                max_id = self.process[i].id
                max_id_process_index = i
        return max_id_process_index
def main():
    obj = GFG()
    obj.initialiseGFG()
    obj.Election()
if __name__ == "__main__":
    main()
```

Synchorinizartion

```
import java.util.*;
public class ClockSync {
   // time server node index
   private static final int TIME_SERVER_NODE = 0;
   // time server offset in seconds
   private static final int TIME_SERVER_OFFSET = 0;
   // a node in network
   public static class Node {
      public int id;
      public long clock;
      public long offset;
      // initialize the clock with the current time in seconds
      Node(int id, long offset) {
         this.id = id;
         this.offset = offset;
         this.clock = System.currentTimeMillis() / 1000;
      }
      public void adjustClock(long offset) {
         clock += offset;
      }
   }
   public static void synchronizeClocks(List<Node> nodes) {
      long serverTime = nodes.get(TIME_SERVER_NODE).clock + TIME_SERVER_OFFSET;
      for (Node node : nodes) {
         if (node.id != TIME_SERVER_NODE) {
            long nodeTime = node.clock + node.offset;
            long offset = serverTime - nodeTime;
            node.adjustClock(offset);
         }
      }
   }
   public static void main(String[] args) {
      List<Node> nodes = new ArrayList<>();
      nodes.add(new Node(1, 10)); // node 1 with 10 seconds offset
      nodes.add(new Node(2, -5)); // node 2 with -5 seconds offset
      nodes.add(new Node(3, 20)); // node 3 with 20 seconds offset
      System.out.println("Before:");
      for (Node node : nodes)
         System.out.println("Node " + node.id + " clock: " + node.clock);
      synchronizeClocks(nodes);
      System.out.println("After:");
      for (Node node : nodes)
         System.out.println("Node " + node.id + " clock: " + node.clock);

   }
}
```

Token based

```
from collections import deque

class Site:
    def __init__(self, site_id, num_sites):
        self.site_id = site_id
        self.RN = [0] * num_sites
        self.LN = [0] * num_sites


class SuzukiKasami:
    def __init__(self, num_sites, initial_token_site_id):
        self.num_sites = num_sites
        self.sites = [Site(i + 1, num_sites) for i in range(num_sites)]
        self.current_token_holder = initial_token_site_id
        self.token_queue = deque()


    def simulate(self):
        while True:
            print(f"Site {self.current_token_holder} has the token.")
            print(f"Site {self.current_token_holder} executes critical section.")
            current_LNs = self.sites[self.current_token_holder - 1].LN
            current_RNs = self.sites[self.current_token_holder - 1].RN
            current_LNs[self.current_token_holder - 1] = current_RNs[self.current_token_holder - 1]
            for j in range(self.num_sites):
                if j + 1 not in self.token_queue and current_RNs[j] == current_LNs[j] + 1:
                    self.token_queue.append(j + 1)
            if self.token_queue:
                next_token_holder = self.token_queue.popleft()
                print(f"Token sent from Site {self.current_token_holder} to Site {next_token_holder}")
                self.current_token_holder = next_token_holder
            else:
                print(f"Site {self.current_token_holder} retains the token.")
```

```
            requesting_site_id = self.request_critical_section()
            if requesting_site_id is None:
                print("Invalid site ID.")
                return
            self.sites[requesting_site_id - 1].RN[requesting_site_id - 1] += 1
            for i in range(self.num_sites):
                if i != requesting_site_id - 1:
                    self.sites[i].RN[requesting_site_id - 1] = max(self.sites[i].RN[requesting_site_id - 1],
self.sites[requesting_site_id - 1].RN[requesting_site_id - 1])
                    if self.current_token_holder == i + 1 and self.sites[i].RN[requesting_site_id - 1] ==
self.sites[i].LN[requesting_site_id - 1] + 1:
                        print(f"Token sent from Site {self.current_token_holder} to Site {requesting_site_id}")
                        self.current_token_holder = requesting_site_id
    def request_critical_section(self):
        while True:
            try:
                requesting_site_id = int(input(f"Enter the site ID that wants to enter the critical section (1-
{self.num_sites}): "))
                if 1 <= requesting_site_id <= self.num_sites:
                    return requesting_site_id
            except ValueError:
                pass
def main():
    num_sites = int(input("Enter the number of sites: "))
    initial_token_site_id = int(input(f"Enter the site ID which initially has the token (1-{num_sites}): "))
    if not (1 <= initial_token_site_id <= num_sites):
        print("Invalid site ID for initial token holder.")
        return
    suzuki_kasami = SuzukiKasami(num_sites, initial_token_site_id)
    suzuki_kasami.simulate()
if __name__ == "__main__":
    main()
```

```
NON TOKEN
class Message:
    def __init__(self, messageType, timestamp, siteId):
        self.messageType = messageType
        self.timestamp = timestamp
        self.siteId = siteId

class Site:
    def __init__(self, siteId):
        self.siteId = siteId
        self.requesting = False
        self.executing = False
        self.timestamp = 0
        self.deferredQueue = []

    def requestCriticalSection(self, sites):
        self.requesting = True
        self.timestamp += 1
        for site in sites:
            if site.siteId != self.siteId:
                requestMessage = Message("REQUEST", self.timestamp,
self.siteId)
                self.sendMessage(requestMessage, site)
        self.waitForReplies(sites)

    def sendMessage(self, message, destination):
        print(f"Site {self.siteId} sends {message.messageType} message to
Site {destination.siteId}")
        destination.receiveMessage(message, self)

    def receiveMessage(self, message, sender):
        print(f"Site {self.siteId} receives {message.messageType} message
from Site {sender.siteId}")
        if message.messageType == "REQUEST":
            if not self.requesting and not self.executing:
                self.sendMessage(Message("REPLY", 0, self.siteId), sender)
            elif self.requesting and message.timestamp < self.timestamp:
                self.deferredQueue.append(message)
        elif message.messageType == "REPLY":
            if self.requesting:
                self.deferredQueue = [m for m in self.deferredQueue if
m.siteId != sender.siteId]
                if not self.deferredQueue:
                    self.executing = True
                    print(f"Site {self.siteId} enters critical section.")

    def waitForReplies(self, sites):
        repliesExpected = len(sites) - 1
        repliesReceived = 0
        while repliesReceived < repliesExpected:
            pass  # Wait for replies

    def releaseCriticalSection(self, sites):
        self.requesting = False
        self.executing = False
        for site in sites:
            if site.siteId != self.siteId:
                for message in self.deferredQueue:
                    self.sendMessage(Message("REPLY", 0, self.siteId),
site)
```

```
        self.deferredQueue.clear()
        print(f"Site {self.siteId} releases critical section.")

def main():
    numberOfSites = int(input("Enter the number of sites: "))
    sites = [Site(i + 1) for i in range(numberOfSites)]
    for site in sites:
        site.requestCriticalSection(sites)
        site.releaseCriticalSection(sites)

if __name__ == "__main__":
    main()
```

Load Balancing

```
class RoundRobinLoadBalancer:
    def __init__(self, numServers):
        self.numServers = numServers
        self.servers = [[] for _ in range(numServers)]

    def addProcesses(self, processes):
        currentIndex = 0
        for process in processes:
            self.servers[currentIndex].append(process)
            currentIndex = (currentIndex + 1) % self.numServers  # Round robin distribution

    def printProcesses(self):
        for i, server in enumerate(self.servers):
            print(f"Server {i + 1} Processes: {server}")

def main():
    # Initial processes in the servers
    initialProcesses = [1, 2, 3, 4, 5, 6, 7]
    # Number of servers
    numServers = 4
    loadBalancer = RoundRobinLoadBalancer(numServers)
    print("Processes before balancing:")
    print(*initialProcesses)
    loadBalancer.addProcesses(initialProcesses)
    print("\nProcesses after balancing:")
    loadBalancer.printProcesses()

if __name__ == "__main__":
    main()
```

```
// client code
// RUN all the 3 codes seperately
import java.rmi.Naming;
import java.util.Scanner;
public class RMI_Client {
   public static void main(String[] args) {
      Scanner sc = null;
      try {
         RMI_interface remoteObject = (RMI_interface) Naming.lookup("rmi://localhost:1878/hello");
         sc = new Scanner(System.in);
         System.out.print("Enter a number to calculate its square root: ");
         double number = sc.nextDouble();
         double result = remoteObject.calculateSquareRoot(number);
         System.out.println("Square root of " + number + " is: " + result);
      } catch (Exception e) {
         System.out.println("The RMI APP is Not running...");
         e.printStackTrace();
      } finally {
         if (sc != null) {
            sc.close();
         }
      }
   }
}
//server code
// import java.nio.channels.AlreadyBoundException;
// import java.rmi.RemoteException;
// import java.rmi.registry.LocateRegistry;
// import java.rmi.registry.Registry;
// import java.rmi.server.UnicastRemoteObject;
```

```
// public class RMI_Server extends UnicastRemoteObject implements RMI_interface {
//     protected RMI_Server() throws RemoteException {
//         super();
//     }

//     public static void main(String[] args)throws RemoteException, AlreadyBoundException {
//         try {
//             Registry registry = LocateRegistry.createRegistry(1878);
//             registry.bind("hello", new RMI_Server());
//             System.out.println("The RMI_Server is running and ready...");
//         }
//         catch (Exception e) {
//             System.out.println("The RMI_Server is not running...");
//         }
//     }

//     @Override
//     public double calculateSquareRoot(double number) throws RemoteException {
//         double result = Math.sqrt(number);
//         System.out.println("Square Root of"+ number+ "is: "+result);
//         return result;
//     }
// }

//interface code
// import java.rmi.Remote;
// import java.rmi.RemoteException;

// public interface RMI_interface extends Remote {
//     double calculateSquareRoot(double number) throws RemoteException;
// }
```