PARALLEL AND DISTRIBUTED COMPUTATION

Distributed and Partitioned Key-Value Store

Deborah Marques Lago - 201806102

Flávio Lobo Vaz - 201509918

José António Dantas Macedo - 201705226

Porto

2021/2022

Table of contents

Project main classes

TestClient.java

This class is used to invoke any of the membership events (join or leave) as well as to invoke any of the operations on key-value pairs (put, get and delete) that will be interpreted by the Store.

The TestClient should be invoked as follows.

```
java main/TestClient <IP address> <port number> <operation> [<opnd>]
```

Example:

```
java main/TestClient 127.0.0.1 8000 put file.txt
```

In the case of a put, this class reads the file and sends the value of the file to the Store, the stores compute the key and return it to the client, in order for the client to be able to invoke *get key* and *delete key*.

The communication between clients and nodes is done using a TCP protocol.

```java
try (Socket socket = new Socket(hostname, port)) {

    OutputStream output = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(output, true);

    switch (operation) {
        case "join":
            writer.println("join".toString());
            break;

        case "leave":
            writer.println("leave".toString());
            break;

        case "put":
            opnd = args[3]; //file pathname of the file with the value to add
            String fileContent = readFile(opnd);
            writer.println("put "+ fileContent);
            break;

        case "get":
            opnd = args[3]; //string of hexadecimal symbols encoding the sha-256 key returned by put
            writer.println("get " + opnd);      You, 3 weeks ago • start client and server logic
            break;

        case "delete":
            opnd = args[3]; //string of hexadecimal symbols encoding the sha-256 key returned by put
            writer.println("delete " + opnd);
            break;
```

Fig.1: Client

Store.java

This class represents a node.

Is used to interpret the client messages and perform the membership events and the key-value operations.

The Store should be invoked as follows.

```
java main/Store <IP_mcast_addr> <IP_mcast_port> <node_id> <Store_port>
```

Example:

```
java main/Store 224.0.0.1 6789 127.0.0.1 8000
```

When the Store is invoked, we open a thread for TCP connections and a thread for UDP connections, and create a folder with the name of the node ip address.

To ensure persistency, the data items and their keys must are stored in this folder.

This folder allows us to test if the cluster key-value operations are working as supposed besides the fact we are running all the nodes on the same machine.

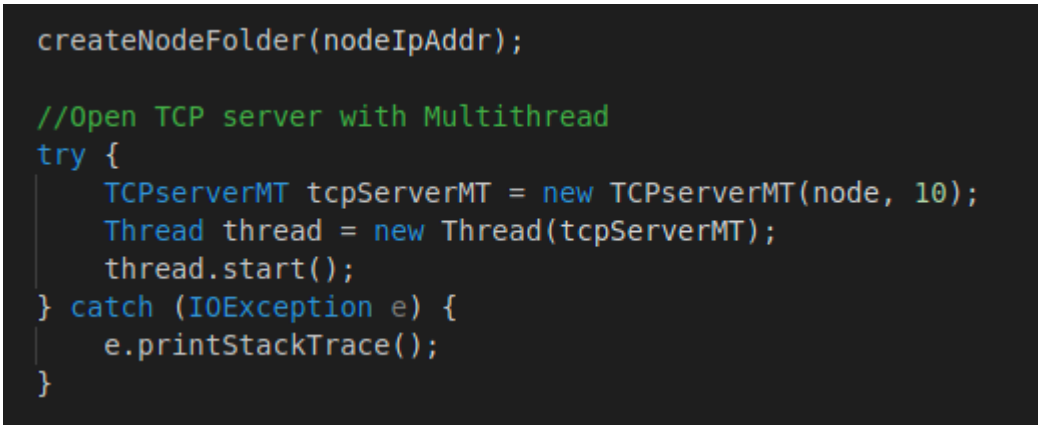


Fig.2: Create a folder and open a thread for TCP

TCP connection

The TCP is multi-threaded with pools, listening for a connection to be made to its socket, when the connection is accepted, a pool executes a handler to deal with the message.

This handler depending on the message received proceeds to perform the membership events and the key-value operations.

```java
public class TCPserverMT implements Runnable {

    private final ServerSocket serverSocket;
    private final ExecutorService pool;
    private Store node;
      You, a week ago • add ip address to the tcp socket

    public TCPserverMT(Store node, int poolSize) throws IOException {
        this.node = node;
        InetAddress ip =InetAddress.getByName(node.nodeIpAddr);

        serverSocket = new ServerSocket(node.storePort,50,ip);
        pool = Executors.newFixedThreadPool(poolSize);
    }

    public void run() { // run the service
      try {
        for (;;) {
          System.out.println("\nServer " + node.nodeIpAddr + " is listening on storePort " + node.storePort);
          pool.execute(new Handler(node, serverSocket.accept()));
        }
      } catch (IOException ex) {
        pool.shutdown();
      }
    }

}
```

Fig.3: TCP multi-thread with pools

UDP connection

UDP allows a node to send join or leave messages to other nodes in the cluster, in order to receive multicast messages from existing nodes in the cluster. A message is sent over UDP which is received in a multicast group. All nodes in the cluster are linked to this group.

It was implemented using thread pools, as we will see below, not only for the Handler that handles multicast messages, but we will also need a TCPServerMT as seen above, to handle the join and leave messages from the TestClient to the node that we want it to do some leave or join operation. But also another TCP channel with Thread to receive the response messages from cluster nodes.

Who sends message to the multicast group does not need to belong to it.

```java
public class MulticastServerMT implements Runnable{

    //private int storePort; preciso disto?Ou só depois para TCP ?
    public MulticastSocket multicastSocket;
    public final ExecutorService pool;
    private Store node;
    public InetAddress multicastGroupAdrr ;


    public MulticastServerMT(Store node, int poolSize, MulticastSocket multicastSocket ) throws IOException{
        //We bound a server(node) to connect a multicast group
        this.node = node;
        InetAddress multicastGroupAdrr = InetAddress.getByName(node.ip_mcast_addr);
        this. multicastGroupAdrr = multicastGroupAdrr;
        this.multicastSocket = multicastSocket;


        pool = Executors.newFixedThreadPool(poolSize);

    }


    @Override
    public void run() {
        try {

            System.out.println("\nServer " + node.nodeId + " is connected on multicastGroupPort:  " + node.ip_mcast_port);
            pool.execute(new MulticastHandler( node, this.multicastSocket, this.multicastGroupAdrr));

        } catch (Exception e) {
            e.printStackTrace();
            pool.shutdown();
        }
    }
}
```

Fig.4: UDP and multicast multi-thread with pools

```java
import main.Store;
import membershipService.Message;
import membershipService.ReceiveObjectsTcp;
import storageservice.StorageService;
import storageservice.utils;
import java.net.SocketTimeoutException;
import java.sql.Time;

Receives the messages
public abstract class TCPserverMembership implements Runnable {

    protected final ServerSocket serverSocket;
    protected final ExecutorService pool;
    protected Store node;
    static public int tries;
    List<Message> listOfObjects = new ArrayList<>();
    protected Socket socket;
    StorageService storageService;
    protected final Integer socketTimeout;


    //construtor que usa nova porta tcp para o membership
    public TCPserverMembership(Store node, int poolSize, int tcpMembershipPort, Integer socketTimeOut) throws IOException {
        this.node = node;
        this.node.membershipTcpPort = tcpMembershipPort;
        this.serverSocket = new ServerSocket(this.node.membershipTcpPort);
        serverSocket.setSoTimeout(socketTimeOut);
        this.pool = Executors.newFixedThreadPool(poolSize);
        this.socketTimeout =socketTimeOut;


    }



    public int  readMessages(String checkOperation) throws ClassNotFoundException, IOException {
        String operation = "";
        int numberOfOperations = 0;
        List<Message> message = new ArrayList<>();
```

Fig.5: TCP multi-thread which also participates in the UDP multicast connection

Membership Service

The join and leave operation should be invoked as follows.

```
java main/TestClient 127.0.0.4 8000 join
```

```
java main/TestClient 127.0.0.4 8000 leave
```

Implemented features:

What determines that a node belongs to a cluster is the fact that **it is connected to a UDP multicast connection**, through which it receives join, leave requests or to **periodically** receive a membership message **to avoid failures every one second** as much as possible. This last part we have partially implemented.

Nodes, after receiving via TCP implemented with TheadPool, a message from the TestClient to join or leave, immediately get temporarily access to a TCP channel. Counter values are incremented and stored correctly (Some information has already been explained above.) .

Nodes, after receiving via TCP implemented with TheadPool, a message from the TestClient to join or leave, immediately get temporarily access to a TCP channel. Counter values are incremented and stored correctly. With a preference always for the highest counter, as it indicates the most current operation.

```java
/**
 * Increment join/leave counter and checks if is the first join
 *
 *
 *
 *  **/
public  int processCounter(String operation){
    if( this.counterMS == 0 && operation.equals("joinMS")){
        return this.counterMS;
    }
    else if(this.counterMS % 2 == 0 && operation.equals("leaveMS")){
        counterMS++;
        return this.counterMS;
    }else if(this.counterMS % 2 != 0 && operation.equals("joinMS")){
        counterMS++;
        return this.counterMS;
```

Fig.6: processCounter

Messages:

We have membership message control, so that receiving 3 messages, the node immediately terminates the join or leave operations.

The event log, like where we have information about the cluster nodes, are implemented with structures that make sure we have only one event per node. Also, as a rule, events with the highest counter are stored. All this information is also stored in files. Being part of the cluster or leaving it respectively.

We implement the sending of messages, using objects of the Message class, which is serializable.

In a Message list, where each Message has a String like the format: idNode counter for the event logs and sha256(idNode) idNode. In the last two Messages, we have the JoinMS or leaveMS operation as a response to the nodes that want to Join or leave .

```java
public class SendObjectsTcp {

    Socket socket;


    Store nodeClient;
    public SendObjectsTcp(String nodeIp, int tcpPort,Store nodeCLient) throws UnknownHostException, IO
        System.out.println("On SendObjects");
        this.nodeClient = nodeCLient;
        this.socket = new Socket(nodeIp,tcpPort);
        System.out.println("Server "+ nodeCLient.nodeIpPort +" connected!" );


    }


    public void sendObjects(List<Message> objects) throws IOException{
        OutputStream outputStream = this.socket.getOutputStream();
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);


        System.out.println("Sending messages to the ServerSocket");

        objectOutputStream.writeObject(objects);


        System.out.println("Closing socket ");
        socket.close();
    }
```

Fig.7: Messages with objects

Fig.8: non-volatile storage

Each node that responds, before doing so, randomly waits between 2 to 1 second.

```
private void sendMemberShipMsgInfo(int tcpPortToSendMessages,String nodeIdPortReceiver, Store n
    System.out.println("sendMemberShipMsgInfo");
    Random randomTime = new Random();
    //Esperar um x nº aleatório de segundos entre 1 seg e 2 seg para enviar resposta via TCP ao

    int time = 1000 + randomTime.nextInt(2001 - 1000);
    Thread.sleep(time);


    List<Message> memberShipInfo = new ArrayList<>();
    System.out.println("Sending objects to Server "+ node.nodeIpAddr  + " with membershipTcpPor
    SendObjectsTcp sendObjectsTcp = new SendObjectsTcp(node.nodeIpAddr,tcpPortToSendMessages ,
    System.out.println("Sending objects to Server "+ nodeIdPortReceiver  + " with membershipTcp
    //save eventLogs in a List of Message
    int counterAux = 0; // para saber situar na lista a informação sobre event logs e clusterNo
```

Fig.9: Random time

```
MulticastHandler(Store node, MulticastSocket multicastSocket,InetAddress multicastGroupAdrr) throws IOException{
    this.node = node;
    this.multicastSocket = multicastSocket;
    this.multicastGroupAdrr = multicastGroupAdrr;
}


@Override
public void run() {
    // TODO Auto-generated method stub
    // read and service request on socket
    try{
        while (true) {
            String threadName = Thread.currentThread().getName();
            System.out.println("\nServer " + node.nodeId + " is connected on multicastGroupPort:  " + node.ip_mcast_port + " waiting for join or leave messages !");

            System.out.println("MulticastThread Name: " + threadName);
            // Aqui estou a ouvir se recebo um Join ou Leave com a info de um nó que quer join/leave no Cluster
            System.out.println("waiting for join or leave messages");
            byte[] buf = new byte[1024 * 4];
            String operation = "";
            // ler a mensagem do socket que deve ter join id_node counter TCPPortNova
            DatagramPacket pack= new DatagramPacket(buf, buf.length);
            String[] messageArgs = new String[4];
            int tcpPortToSendMessages = 0;


            this.multicastSocket.receive(pack);
            //Deserialze object
            ByteArrayInputStream bais = new ByteArrayInputStream(buf);
            ObjectInputStream ois = new ObjectInputStream(bais);


            String aux = (String) ois.readObject();

            if(aux.contains(":")){
                String[] listOfObjects = aux.split(":",0);


            if(listOfObjects.length != 0 && listOfObjects[listOfObjects.length - 1] =="update"){
                operation = listOfObjects[listOfObjects.length - 1];
                List<Message> eventLogs= new ArrayList<>();
                for(int i = 0; i < listOfObjects.length ; i++)
                    eventLogs.add(new Message(listOfObjects[i]));

                for (Message s: eventLogs){
                        System.out.println(s.getMsgToSend());
                }
            }
```

Fig.10: Incomplete update of cluster nodes

Conclusion: In general, everything that was expected regarding the Membership Service was practically implemented. Being only incomplete, the periodic update request is limited to 32 event logs in the files.

Store Service

The store service is responsible for handling the key-value operations (put, get and delete), and is implemented as a distributed partitioned hash table in which each cluster node stores the key-value pairs in a bucket.

To generate the keys we used SHA-256 for hashing the value, that way we don't have hash collisions.

In order to have a distributed partitioned hash table, we use consistent hashing to partition the key-value pairs among the nodes in the cluster.

```
public String consistentHashing(String hashCode, Store node) {

    String target = hashCode;

    //Processing when not included
    if (!node.nodesInCluster.containsKey(hashCode)) {
        target = node.nodesInCluster.ceilingKey(hashCode);
        if (target == null && !node.nodesInCluster.isEmpty()) {
            target = node.nodesInCluster.firstKey();
        }
    }

    return node.nodesInCluster.get(target);

}
```

Fig.11: Consistent Hashing Function

Thus a client needs only know one of the cluster nodes to be able to access the key-value store.

When a node receives any of the key-value operations from the client we check the node from the cluster that has the closer key, if it is himself it proceeds to perform the key-value operation, if it is not him, the node sends the node responsible a message using TCP.

In case the responsible node is down, the message will be sent to the node ahead.

```java
public String checkNodeWorking(String nodeCloserId){
    String nextNode = null;

    while(true)
    {      You, 17 hours ago • check if node is up
        String[] parts = nodeCloserId.split(":");
        String nodeCloserIp = parts[0];
        int nodeCloserPort = Integer.valueOf(parts[1]);
        System.out.println("Trying node: " + nodeCloserId);

        try (Socket socket = new Socket(nodeCloserIp, nodeCloserPort)) {
            socket.close();
            nextNode = nodeCloserId;
            return nextNode;

        } catch (UnknownHostException ex) {

            System.out.println("Server not found: " + ex.getMessage());

        } catch (IOException ex) {

            System.out.println(ex.getMessage() + " with node ip "+ nodeCloserIp);

            String target = node.nodesInCluster.higherKey(utils.sha_256(nodeCloserId));
            if(target == null)
            {
                break;
            }
            nodeCloserId = node.nodesInCluster.get(target);

        }

    }

    return nextNode;
}
```

Fig.12: Function that checks if the node is down, and gives the next node working

Handling the operations:

- **Put**
    - Saves the key-value pair in the node hash table
    - Creates a file, with the name as the key, and the content as the value
    - Saves that file in the node folder

```java
public void put(String key,String value) {
    node.hashTable.put(key, value);
    String PATH = System.getProperty("user.dir") + "/nodesFolders/" + node.nodeIpAddr;
    try {
        FileWriter myWriter = new FileWriter(PATH+"/"+key+".txt");
        myWriter.write(value);
        myWriter.close();
      } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

Fig.13: Put Function

- **Get**
  - Get from the hash table the value correspondent to the key

```
file_value = node.hashTable.get(key);
```

Fig.14: Get Function

- **Delete**
  - Delete the key-value pair in the node hash table
  - Delete a file, with the name as the key

```
node.hashTable.remove(key);
String PATH = System.getProperty("user.dir") + "/nodesFolders/" + node.nodeIpAddr;
File myObj = new File(PATH+"/"+key+".txt");
if (myObj.delete()) {
    System.out.println("Deleted the file: " + myObj.getName());
} else {
    System.out.println("Failed to delete the file.");
}
delete_status = "Delete Successful";
```

Fig.15: Delete Function

Nodes may have to transfer keys to other nodes upon a membership change.

Handling the events:

- **Join event**

  When a node joins the cluster, it sends a message via TCP to the successor node asking for the keys that are smaller or equal to his id.

  The successor of the node proceeds to check the keys he has, and if a key is smaller or equal to the joining node, it sends that key-value pair to him, via TCP through the put value function.

```
public void askForFiles(){

    String nodeAhead = this.node.nodesInCluster.higherKey(utils.sha_256(this.node.nodeId));
            if(nodeAhead == null)
            {
                return;
            }
            nodeAhead = node.nodesInCluster.get(nodeAhead);


    System.out.println("NodeAhead" + nodeAhead);


    if(nodeAhead==null){
        return;
    }
    nodeAhead = checkNodeWorking(nodeAhead);

    String[] parts = nodeAhead.split(regex: ":");
    String nodeCloserIp = parts[0];
    int nodeCloserPort = Integer.valueOf(parts[1]);

    try (Socket socket = new Socket(nodeCloserIp, nodeCloserPort)) {
        OutputStream output = socket.getOutputStream();
        PrintWriter writer = new PrintWriter(output, autoFlush: true);
        writer.println("join_event "+ this.node.nodeId);

    } catch (UnknownHostException ex) {
        System.out.println("Server not found: " + ex.getMessage());
    } catch (IOException ex) {
        System.out.println("I/O error: " + ex.getMessage());
    }
}
```

Fig.16: Function to ask the successor node for the keys

```
public void joinEvent(String joiningNodeId){

    String nodeId = this.node.nodeId;
    Set<String> keys = this.node.hashTable.keySet();
    for(String key: keys){
        System.out.println("Value of "+key+" is: "+ this.node.hashTable.get(key));
        nodeId = consistentHashing(key, this.node);

        if(!nodeId.equals(this.node.nodeId))
        {
            String[] parts = nodeId.split(regex: ":");
            String nodeCloserIp = parts[0];
            int nodeCloserPort = Integer.valueOf(parts[1]);

            try (Socket socket = new Socket(nodeCloserIp, nodeCloserPort)) {
                OutputStream output = socket.getOutputStream();
                PrintWriter writer = new PrintWriter(output, autoFlush: true);
                writer.println("put "+ this.node.hashTable.get(key));

            } catch (UnknownHostException ex) {
                System.out.println("Server with node ip "+ nodeId + " not found: " + ex.getMessage());
            } catch (IOException ex) {
                System.out.println("I/O error: " + ex.getMessage());
            }
        }
    }
}
```

Fig.17 : Function that sends the key-value pair to the joining node

- **Leave event**

  Before leaving the cluster, the node iterate through his hashtable, sending the key-value pairs to the successor node and deleting key-value pairs.

```
public void sendFilesBeforeLeaving(){

    String nodeAhead = this.node.nodesInCluster.higherKey(utils.sha_256(this.node.nodeId));
    if(nodeAhead == null)
    {
        return;
    }
    nodeAhead = node.nodesInCluster.get(nodeAhead);


    System.out.println("NodeAhead" + nodeAhead);


    if(nodeAhead==null){
        return;
    }
    nodeAhead = checkNodeWorking(nodeAhead);

    String[] parts = nodeAhead.split(regex: ":");
    String nodeAheadIp = parts[0];
    int nodeAheadPort = Integer.valueOf(parts[1]);


    Set<String> keys = this.node.hashTable.keySet();
    for(String key: keys){
        System.out.println("Value of "+key+" is: "+ this.node.hashTable.get(key));


        try (Socket socket = new Socket(nodeAheadIp, nodeAheadPort)) {
            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, autoFlush: true);
            writer.println("put "+ this.node.hashTable.get(key));

        } catch (UnknownHostException ex) {
            System.out.println("Server with node ip "+ nodeAheadIp + " not found: " + ex.getMessage());
        } catch (IOException ex) {
            System.out.println("I/O error: " + ex.getMessage());
        }


        delete(key);


    }       You, 3 days ago • join and leave events functions w/out testing
}
```

Fig.18 : Function that sends the key-value pair successor before leaving the cluster

Fault-Tolerance

On performing key-values operations, if the responsible node is down, the message will be sent to the node ahead,until there are no more nodes in the cluster.

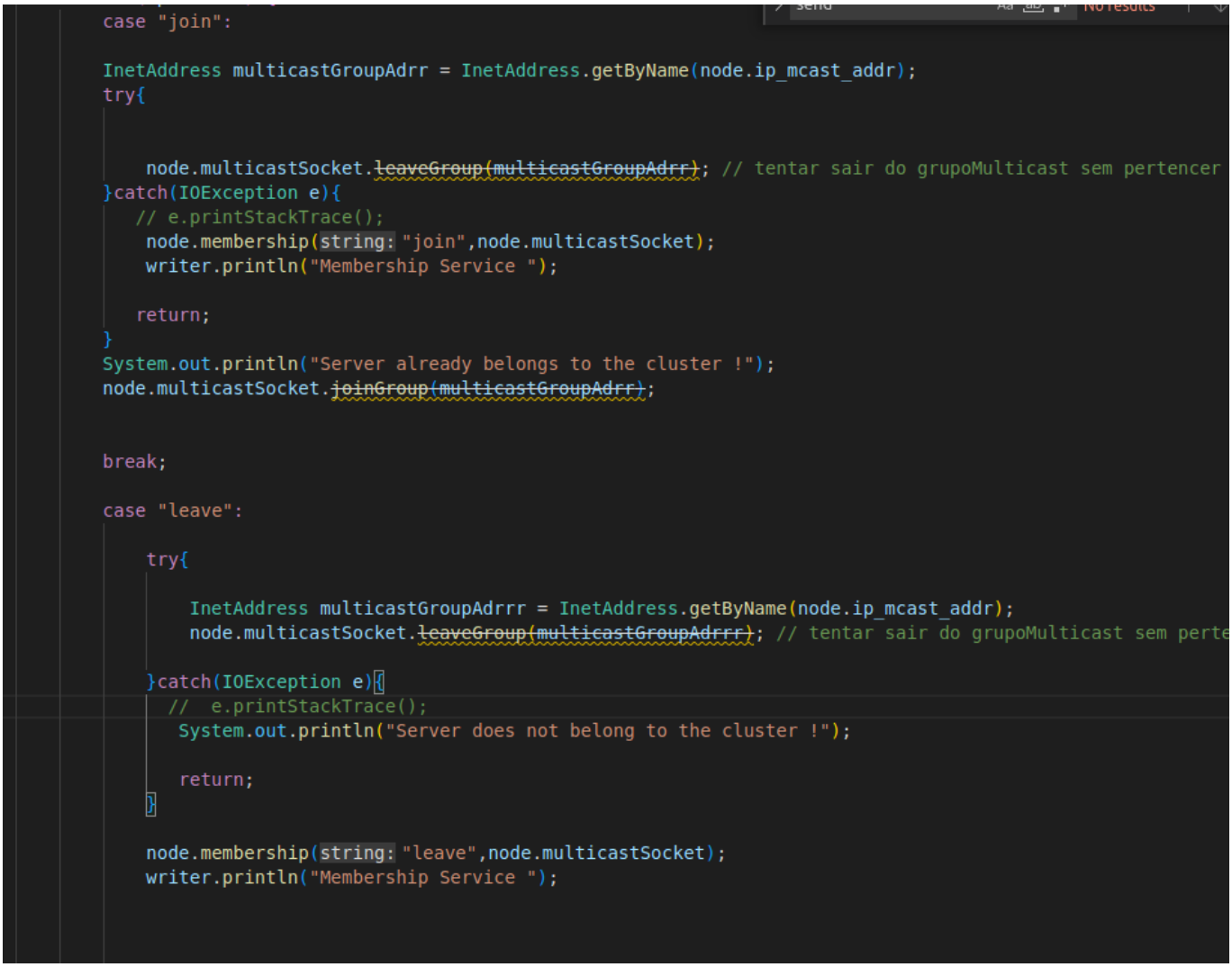
Nodes that do not belong to the cluster, and we try to do leave, this operation is not allowed. Same for join when the node already belongs to the cluster.

Nodes that do not belong to the cluster, and we try to do leave, this operation is not allowed. Same for join when the node already belongs to the cluster. We also make sure that we keep event logs as up-to-date as possible for each node. If by chance in any join or leave operation, the event logs are updated, we always keep the id node with the corresponding highest counter of the operations.

Example:

```
public String checkNodeWorking(String nodeCloserId){
    String nextNode = null;

    while(true)
    {
        String[] parts = nodeCloserId.split(regex: ":");
        String nodeCloserIp = parts[0];
        int nodeCloserPort = Integer.valueOf(parts[1]);
        System.out.println("Trying node: " + nodeCloserId);

        try (Socket socket = new Socket(nodeCloserIp, nodeCloserPort)) {
            socket.close();
            nextNode = nodeCloserId;
            return nextNode;

        } catch (UnknownHostException ex) {

            System.out.println("Server not found: " + ex.getMessage());

        } catch (IOException ex) {

            System.out.println(ex.getMessage() + " with node ip "+ nodeCloserIp);

            String target = node.nodesInCluster.higherKey(utils.sha_256(nodeCloserId));
            if(target == null)
            {
                break;
            }
            nodeCloserId = node.nodesInCluster.get(target);

        }

    }

    return nextNode;
}
```

Fig.19: Function to get the next node up in the cluster



```
case "join":

InetAddress multicastGroupAdrr = InetAddress.getByName(node.ip_mcast_addr);
try{


    node.multicastSocket.leaveGroup(multicastGroupAdrr); // tentar sair do grupoMulticast sem pertencer
}catch(IOException e){
   // e.printStackTrace();
    node.membership(string: "join",node.multicastSocket);
    writer.println("Membership Service ");

   return;
}
System.out.println("Server already belongs to the cluster !");
node.multicastSocket.joinGroup(multicastGroupAdrr);


break;

case "leave":

    try{

        InetAddress multicastGroupAdrrr = InetAddress.getByName(node.ip_mcast_addr);
        node.multicastSocket.leaveGroup(multicastGroupAdrrr); // tentar sair do grupoMulticast sem perte

    }catch(IOException e){
      //  e.printStackTrace();
       System.out.println("Server does not belong to the cluster !");

       return;
    }

    node.membership(string: "leave",node.multicastSocket);
    writer.println("Membership Service ");
```

Fig.20: Fault-Tolerance Example

Concurrency

Thread-pools

We use a TCP connection multi-threaded with pools, listening for a connection to be made to its socket, when the connection is accepted, a pool executes a handler to deal with the message.

We also use, somewhat in line with what was explained above about UDP, a mix of thread pools with simple multiThread. That deal with messages received by nodes via multicast. From the reply to these messages via TCP controlling the number of times, that the Membership Service is intended. In order not to clog the nodes that want to leave or join with messages.

A few more photos, which initially did not appear above:

```java
import main.Store;
import membershipService.Message;
import storageservice.utils;

public class TCPserverMembershipJoin extends TCPserverMembership {
    public TCPserverMembershipJoin(Store node, int poolSize, int tcpMembershipPort,Integer socketTimeOut) throws IOException{

        super(node, poolSize, tcpMembershipPort, socketTimeOut);

    }

    public void run() { // run the service
        String threadName = Thread.currentThread().getName();
        System.out.println("Thread Name: " + threadName);
        //ScheduledThreadPoolExecutor scheduledpool = new ScheduledThreadPoolExecutor(10);

        for (int joinTries = 1; joinTries <=3; joinTries++) { // isto faz sentido ?!??!?
          System.out.println("\nServer " + node.nodeId + " is listening on storePort " + node.storePort);


       //  scheduledpool.schedule(new HandlerJoinAttempt(), 10000, TimeUnit.MILLISECONDS);




        try {
            int numberOfJoins;

                numberOfJoins = readMessages(checkOperation: "joinMS");
                System.out.println("NumberOfJoin: " + numberOfJoins);
```

Fig.21: Extra SnapShot of the code

```java
public class TCPserverMembershipLeave extends TCPserverMembership {
    public TCPserverMembershipLeave(Store node, int poolSize, int tcpMembershipPort,Integer socketTimeOut) throws IOException{

        super(node, poolSize, tcpMembershipPort, socketTimeOut);

    }

    public void run() { // run the service
        String threadName = Thread.currentThread().getName();
        System.out.println("Thread Name: " + threadName);
        //ScheduledThreadPoolExecutor scheduledpool = new ScheduledThreadPoolExecutor(10);

        for (int joinTries = 1; joinTries < 3; joinTries++) { // isto faz sentido ?!??!?
          System.out.println("\nServer " + node.nodeId + " is listening on storePort " + node.storePort);


       //  scheduledpool.schedule(new HandlerJoinAttempt(), 10000, TimeUnit.MILLISECONDS);




        try {
            int numberOfLeaves;

                numberOfLeaves= readMessages(checkOperation: "leaveMS");
                System.out.println("NumberOfLeaves: " + numberOfLeaves);


            if (numberOfLeaves == 3) {
                saveMsgReceived(operation: "leaveMS");
                socket.close();
                System.out.println("NumberOfLeaves: " + numberOfLeaves);
                return;

        }
```

Fig.22: Extra SnapShot of the code