U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems
Project 2 Report
2020

Project developed by Group 22 of Class 4:

- Gaspar Pinheiro - up201704700@fe.up.pt
- Gustavo Torres - up201706473@fe.up.pt
- João Araújo - up201705577@fe.up.pt
- Joaquim Rodrigues - up201704844@fe.up.pt

Overview

Our project implements a backup system that supports all the features that the previous project supported, namely backup of a file, deletion of the backed up file, restore the file from the backed up chunks, reclaiming the space of a peer used for saving chunks of files, sending those chunks to another peer and showing the state of a peer.

For increasing the ceiling of our grade, we used SSLSockets, Chord, Thread-pools and fault-tolerance.

Protocols

For communication between the TestApp and the Peer, we used TCP

The messages sent have the following possible formats (the code for this part is in the PeerMessageHandler) :

- BACKUP <filename> <replication_degree>
- DELETE <filename>
- RESTORE <filename>
- RECLAIM <space>
- STATUS
- PUT <key, value>
- GET <key>
- REMOVE <key>

For communication between peers for the Chord architecture , we used TCP with SSLSockets.

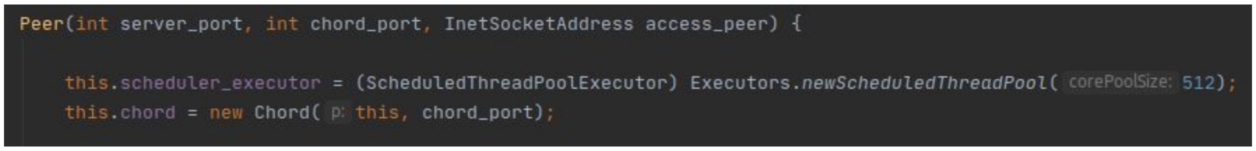
The messages sent have the following possible formats (the code for this part is in the *ChordMessageHandler*, and the variables are handled in the *ChordMessage*):

- LOOKUP <key>
- GETSUCCESSOR <key> <ip> <port>

- SETSUCCESSOR <unnecessary?> <ip> <port>
- GETPREDECESSOR
- SETPREDECESSOR <unnecessary?> <ip> <port>
- NEWFINGER <key> <ip> <port>
- DELETEFINGER <key> <old_key> <ip> <port>
- PUT <key> <old_key> <replication_degree>
- GET <key> <replication_degree>
- REMOVE <key> <replication_degree>
- GETDATA <key> <ip> <port>
- NOTIFY <unnecessary?> <ip> <port>
- FAIL
- VALID

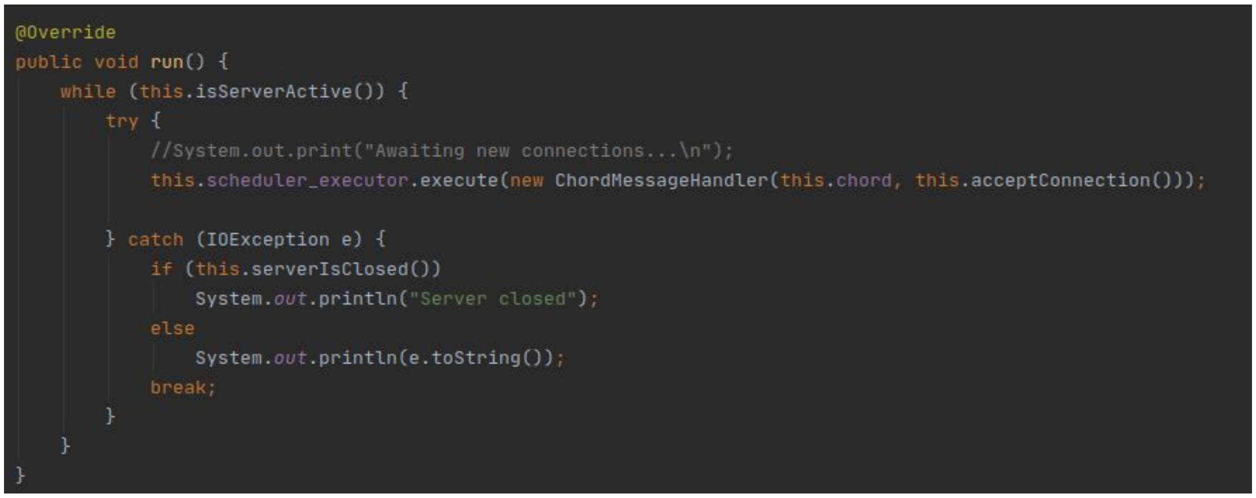
Concurrency Design

We used **thread pools** to optimize concurrency. The peer initializes a *ScheduledThreadPoolExecutor* object with a **512** core pool size. This object is then used to execute concurrent tasks.



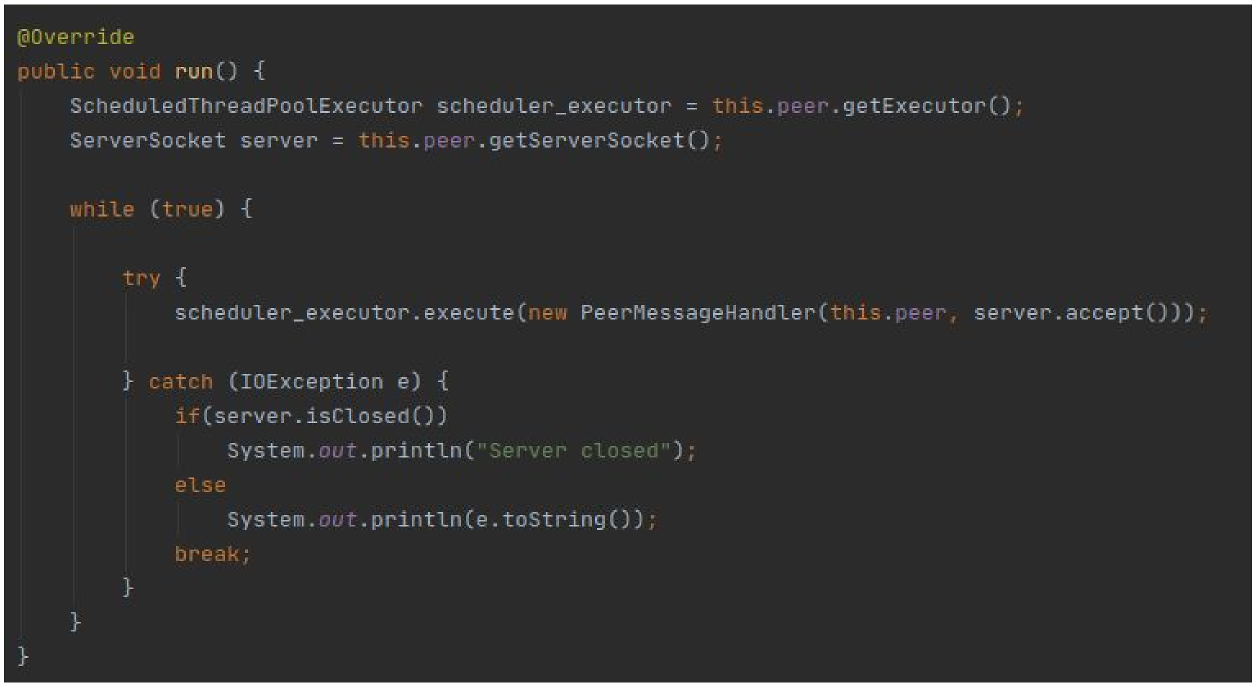
```
Peer(int server_port, int chord_port, InetSocketAddress access_peer) {

    this.scheduler_executor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool( corePoolSize: 512);
    this.chord = new Chord( p: this, chord_port);
```

In the case of communication between peers for the *Chord* protocol, the class *ChordServer* is responsible for keeping the peer listening to incoming messages from other peers that pertain to the Chord protocol. This class implements the *Runnable* interface and keeps running a while loop until the peer closes the server. In this loop, the program waits for a new TCP connection from another Peer. When it accepts the new connection, it uses the *ScheduledThreadPoolExecutor* object mentioned above to execute a new *ChordMessageHandler* object that reads the incoming messages and handles them appropriately. This way, we assure the peer is always ready to receive and handle new messages.



```java
@Override
public void run() {
    while (this.isServerActive()) {
        try {
            //System.out.print("Awaiting new connections...\n");
            this.scheduler_executor.execute(new ChordMessageHandler(this.chord, this.acceptConnection()));

        } catch (IOException e) {
            if (this.serverIsClosed())
                System.out.println("Server closed");
            else
                System.out.println(e.toString());
            break;
        }
    }
}
```

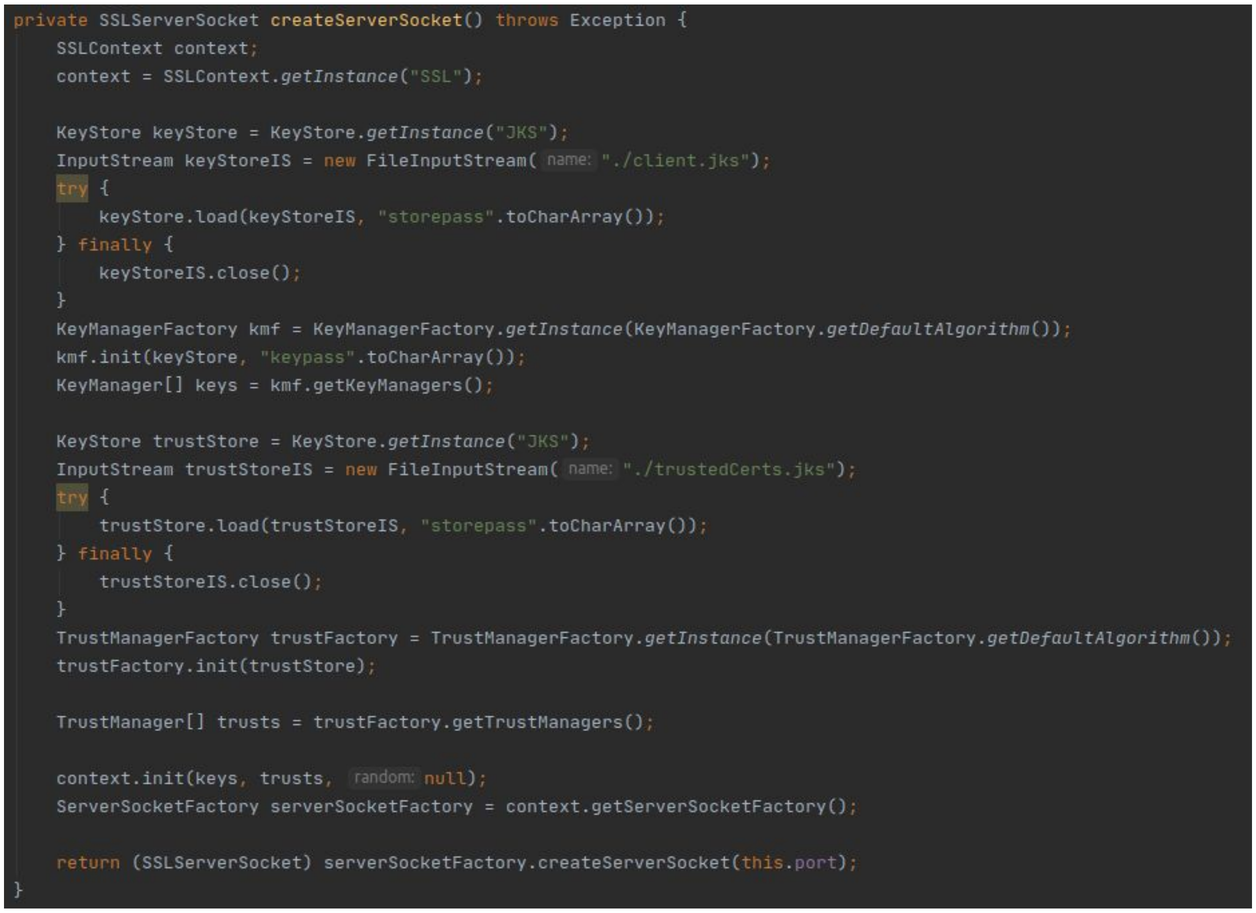
We also do something similar to the mentioned above when it comes to the communication between the Peer and the TestApp. In this case, we have a *PeerServer* object that waits for a new TCP connection and once it accepts it, a concurrent task is executed by the *ScheduledThreadPoolExecutor* object by creating a new *PeerMessageHandler* object to handle the requests received by the TestApp. This way it is possible for the Peer to handle simultaneous requests.



```java
@Override
public void run() {
    ScheduledThreadPoolExecutor scheduler_executor = this.peer.getExecutor();
    ServerSocket server = this.peer.getServerSocket();

    while (true) {

        try {
            scheduler_executor.execute(new PeerMessageHandler(this.peer, server.accept()));

        } catch (IOException e) {
            if(server.isClosed())
                System.out.println("Server closed");
            else
                System.out.println(e.toString());
            break;
        }
    }
}
```

JSSE

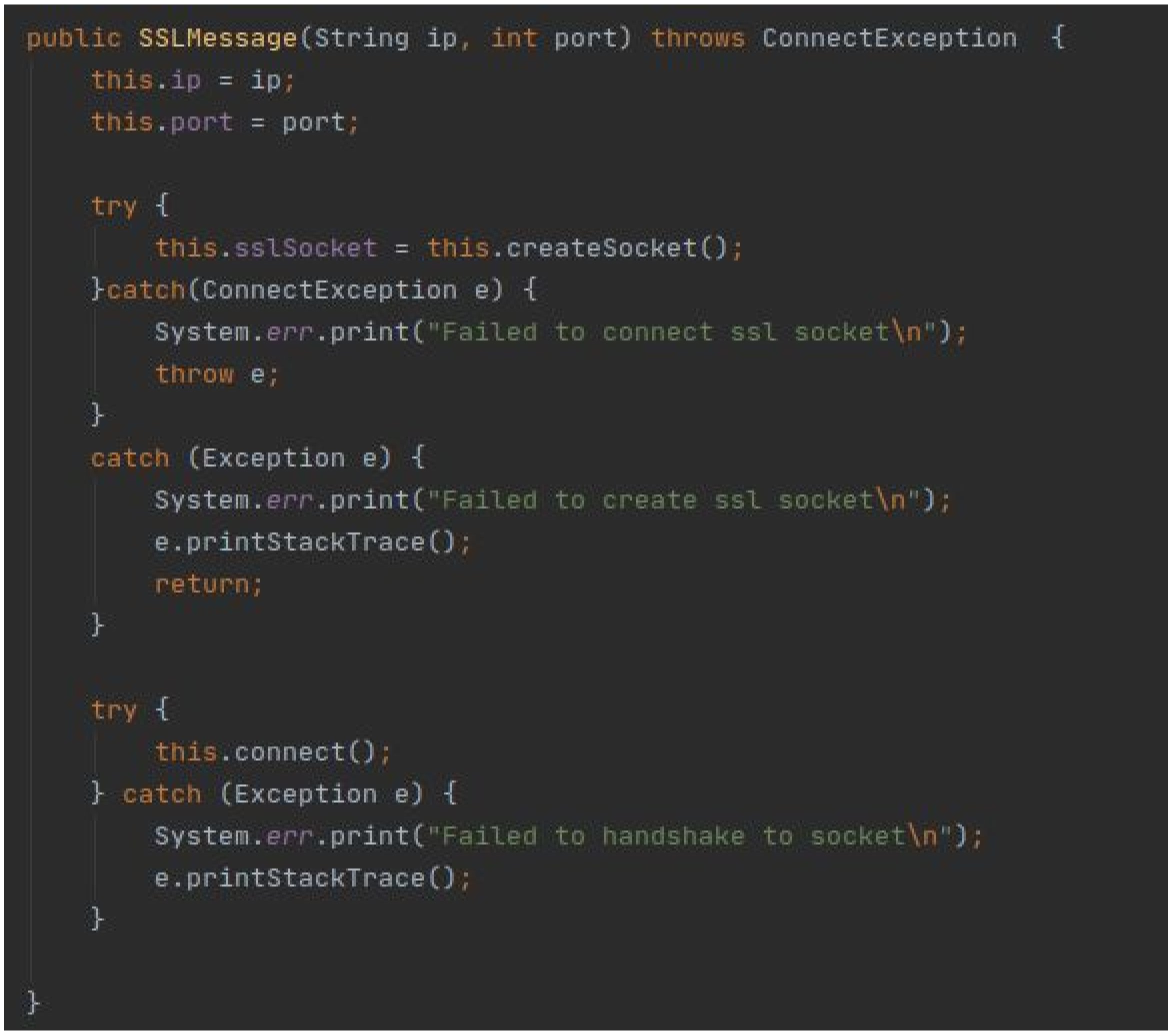
We use SSLSockets in all communications between peers. The *ChordServer* mentioned above extends the class *SSLServer*. This last one is responsible for setting up the SSLServerSocket object that is used to receive messages from other peers.



```java
private SSLServerSocket createServerSocket() throws Exception {
    SSLContext context;
    context = SSLContext.getInstance("SSL");

    KeyStore keyStore = KeyStore.getInstance("JKS");
    InputStream keyStoreIS = new FileInputStream( name: "./client.jks");
    try {
        keyStore.load(keyStoreIS, "storepass".toCharArray());
    } finally {
        keyStoreIS.close();
    }
    KeyManagerFactory kmf = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(keyStore, "keypass".toCharArray());
    KeyManager[] keys = kmf.getKeyManagers();

    KeyStore trustStore = KeyStore.getInstance("JKS");
    InputStream trustStoreIS = new FileInputStream( name: "./trustedCerts.jks");
    try {
        trustStore.load(trustStoreIS, "storepass".toCharArray());
    } finally {
        trustStoreIS.close();
    }
    TrustManagerFactory trustFactory = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
    trustFactory.init(trustStore);

    TrustManager[] trusts = trustFactory.getTrustManagers();

    context.init(keys, trusts,  random: null);
    ServerSocketFactory serverSocketFactory = context.getServerSocketFactory();

    return (SSLServerSocket) serverSocketFactory.createServerSocket(this.port);
}
```

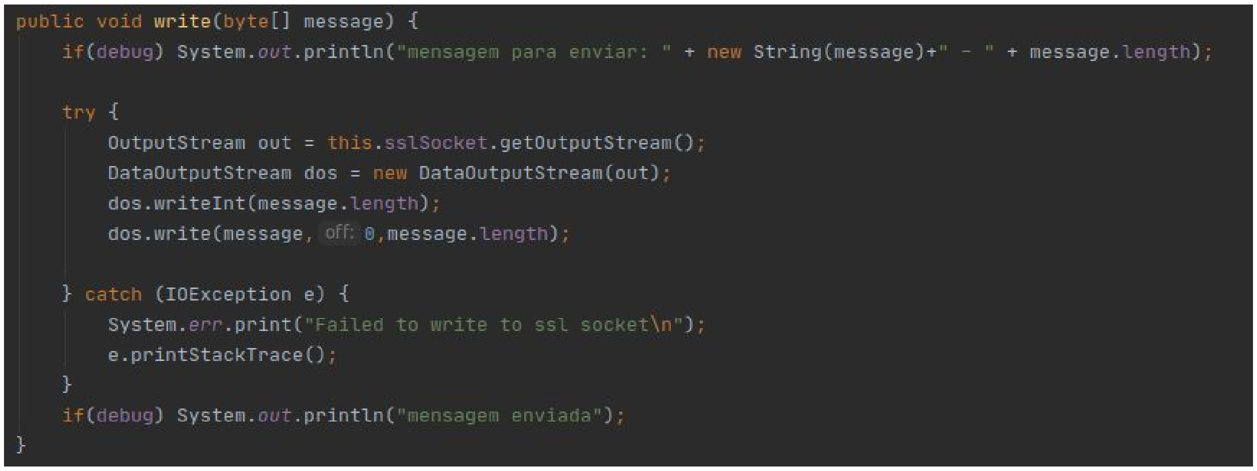
Aftar calling this method the SSLServer object sets the requirement of **client authentication** to true.

When a peer wants to send a new message to another peer, it uses the class SSLMessage, that creates a new SSLSocket object with the given hostname and port number and then starts the Handshake protocol by invoking the *startHandshake* method of the SSLSocket object created.

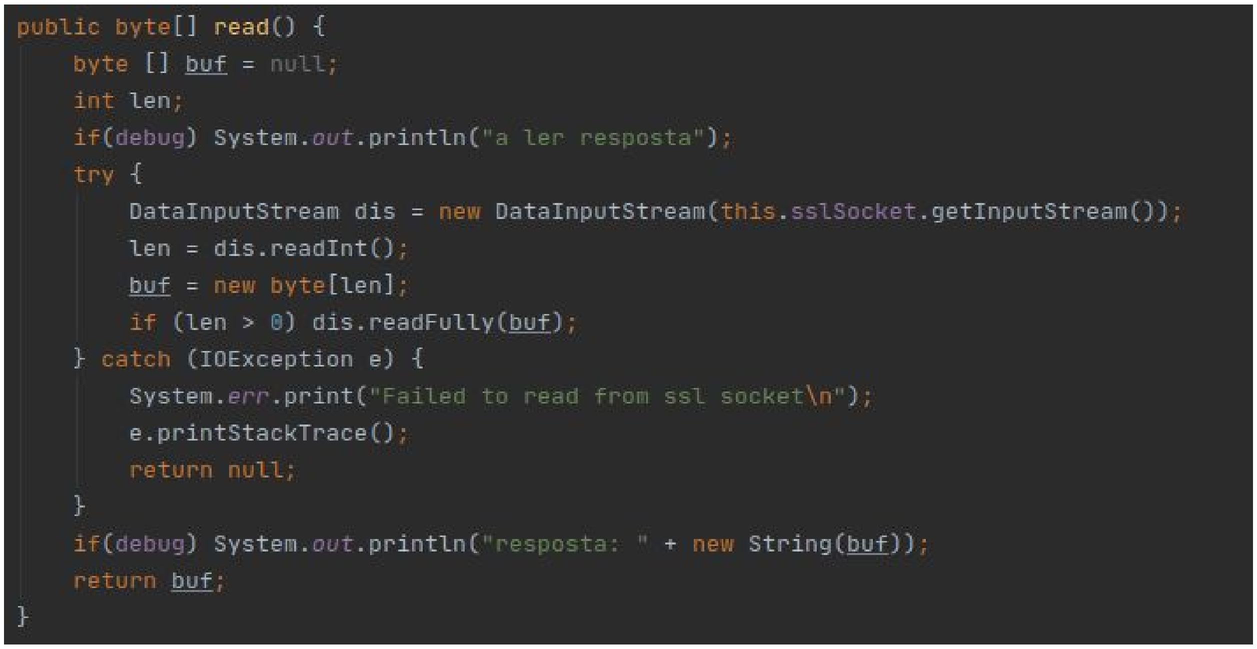


```java
public SSLMessage(String ip, int port) throws ConnectException  {
    this.ip = ip;
    this.port = port;

    try {
        this.sslSocket = this.createSocket();
    }catch(ConnectException e) {
        System.err.print("Failed to connect ssl socket\n");
        throw e;
    }
    catch (Exception e) {
        System.err.print("Failed to create ssl socket\n");
        e.printStackTrace();
        return;
    }

    try {
        this.connect();
    } catch (Exception e) {
        System.err.print("Failed to handshake to socket\n");
        e.printStackTrace();
    }

}
```

The methods write and read of this class are used to send and receive messages from that socket.



```java
public void write(byte[] message) {
    if(debug) System.out.println("mensagem para enviar: " + new String(message)+" - " + message.length);

    try {
        OutputStream out = this.sslSocket.getOutputStream();
        DataOutputStream dos = new DataOutputStream(out);
        dos.writeInt(message.length);
        dos.write(message, off: 0,message.length);

    } catch (IOException e) {
        System.err.print("Failed to write to ssl socket\n");
        e.printStackTrace();
    }
    if(debug) System.out.println("mensagem enviada");
}
```

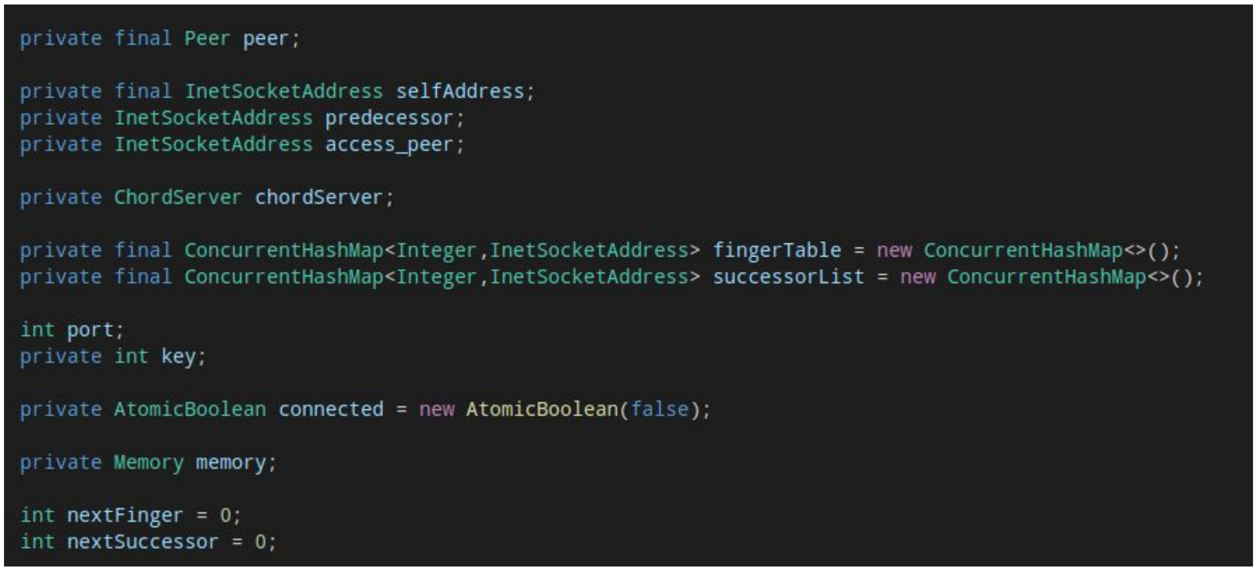


```java
public byte[] read() {
    byte [] buf = null;
    int len;
    if(debug) System.out.println("a ler resposta");
    try {
        DataInputStream dis = new DataInputStream(this.sslSocket.getInputStream());
        len = dis.readInt();
        buf = new byte[len];
        if (len > 0) dis.readFully(buf);
    } catch (IOException e) {
        System.err.print("Failed to read from ssl socket\n");
        e.printStackTrace();
        return null;
    }
    if(debug) System.out.println("resposta: " + new String(buf));
    return buf;
}
```

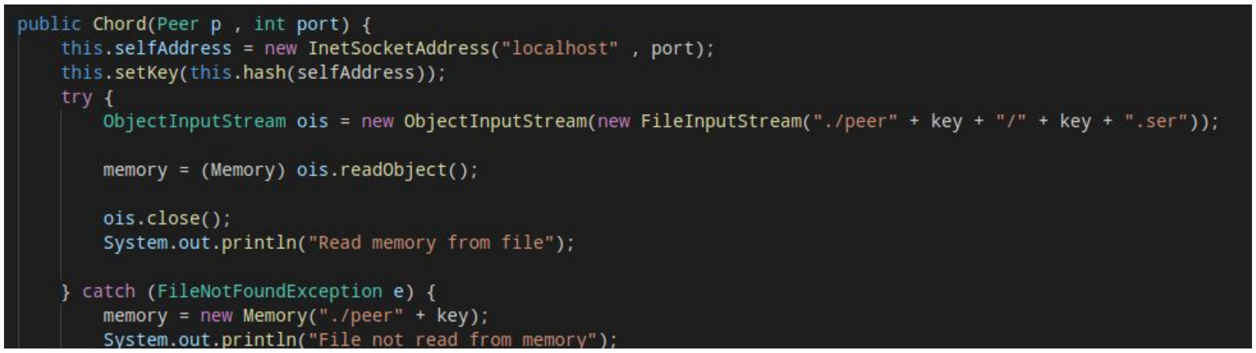
Scalability

For scalability, we implemented a Chord architecture and implemented the concurrency design mentioned above.

Each Peer has a hash key, which allows to add and remove Peers easily from the fingerTable. The chord manages the successors of each Peer.



```java
private final Peer peer;

private final InetSocketAddress selfAddress;
private InetSocketAddress predecessor;
private InetSocketAddress access_peer;

private ChordServer chordServer;

private final ConcurrentHashMap<Integer,InetSocketAddress> fingerTable = new ConcurrentHashMap<>();
private final ConcurrentHashMap<Integer,InetSocketAddress> successorList = new ConcurrentHashMap<>();

int port;
private int key;

private AtomicBoolean connected = new AtomicBoolean(false);

private Memory memory;

int nextFinger = 0;
int nextSuccessor = 0;
```

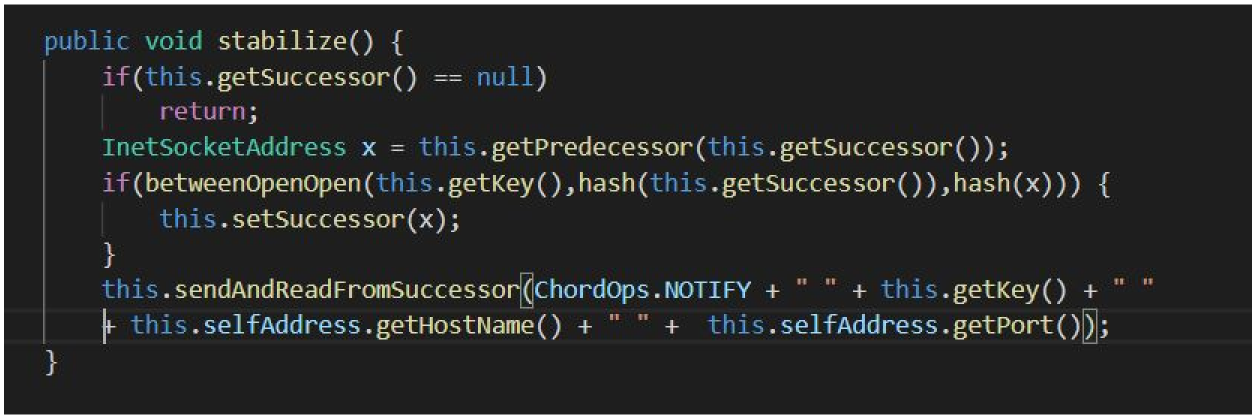


```
public Chord(Peer p , int port) {
    this.selfAddress = new InetSocketAddress("localhost" , port);
    this.setKey(this.hash(selfAddress));
    try {
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("./peer" + key + "/" + key + ".ser"));

        memory = (Memory) ois.readObject();

        ois.close();
        System.out.println("Read memory from file");

    } catch (FileNotFoundException e) {
        memory = new Memory("./peer" + key);
        System.out.println("File not read from memory");
```

Fault-Tolerance

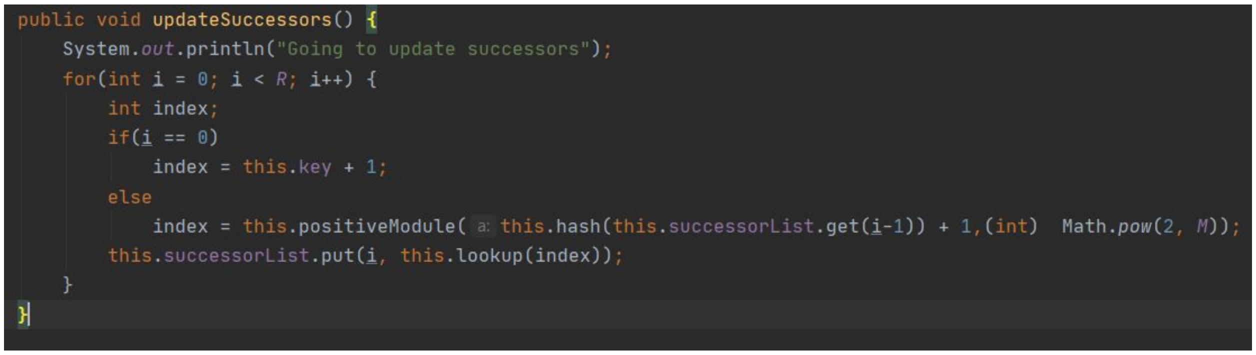
We implemented 2 main features to improve the fault tolerance, and they were the stabilize function (in Chord.java) and a successor list (also in Chord.java)

The stabilize function is called periodically in each peer and if it detects that the successor of my predecessor is between me and my predecessor, it makes that peer my successor and sends it a notify message.



```
public void stabilize() {
    if(this.getSuccessor() == null)
        return;
    InetSocketAddress x = this.getPredecessor(this.getSuccessor());
    if(betweenOpenOpen(this.getKey(),hash(this.getSuccessor()),hash(x))) {
        this.setSuccessor(x);
    }
    this.sendAndReadFromSuccessor(ChordOps.NOTIFY + " " + this.getKey() + " "
    + this.selfAddress.getHostName() + " " +  this.selfAddress.getPort());
}
```

The successor list is a ConcurrentHashMap<Integer, InetSocketAddress> called successorList, it's a field of the Chord class that contains the list of the successors of that peer, having 30 successors. The successors are found through the updateSuccessors function, which calculates the successors based on iterations, according to its key hash module



```java
public void updateSuccessors() {
    System.out.println("Going to update successors");
    for(int i = 0; i < R; i++) {
        int index;
        if(i == 0)
            index = this.key + 1;
        else
            index = this.positiveModule( a: this.hash(this.successorList.get(i-1)) + 1,(int)  Math.pow(2, M));
        this.successorList.put(i, this.lookup(index));
    }
}
```