U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Class 7 Group 5

Ana Clara Moreira Gadelho	up201806309@fe.up.pt
Diogo Samuel Gonçalves Fernandes	up201806250@fe.up.pt
Leonor Marques Gomes	up201806567@fe.up.pt
Juliane de Lima Marubayashi	up201800175@fe.up.pt

2nd June 2021

SDIS Project - 2020/21 - MIEIC

Teachers

Pedro Souto	pfs@fe.up.pt
Rui Rocha	ruirocha@fe.up.pt

Index

1 Overview

The aim of this project is to elaborate a distributed system service for backing up files on the Internet.

1.1 Compile and Run

Under the root folder of our project, to compile our java code run: `make` .

You can also run the following command if you want to delete the generated files: `make clean`

After this, to initialize the peer you should go to ./out folder and run the following command where `$IP` is the IP of the peer and `$PORT` an open port of the peer. This command will initialize a peer, that will be responsible for also initializing the Chord Ring and system.

```
sh ../scripts/peer.sh $IP $PORT
```

If you already have a Chord Ring initialized, you can run the following command where `$CHORD_NODE_IP` and `$CHORD_NODE_PORT` are respectively the IP and PORT of one peer already connected to the Chord Ring.

```
sh ../scripts/peer.sh $IP $PORT $CHORD_NODE_IP $CHORD_NODE_PORT
```

1.2 Operations supported by the backup service

To run any operation of the backup service, it's necessary to write on the terminal the command:

```
sh ../scripts/protocol.sh $PROTOCOL
```

Where `$PROTOCOL` can be one of the following:

- `$IP $PORT BACKUP $FILE_PATH $REP_DEG`
- `$IP $PORT RESTORE $FILE_NAME`
- `$IP $PORT DELETE $FILE_NAME`
- `$IP $PORT RECLAIM $PEER_ID $SPACE`

Backup

The backup protocol is responsible to save files with the desired replication degree among the peers.

```
$IP $PORT BACKUP $FILE_PATH $REP_DEG
```

- `IP` - IP of the host to invoke the backup
- `PORT` - PORT of the host to invoke the backup
- `FILE_PATH` - Path of the file to backup

Restore

The restore protocol is responsible to restore files that exist at any point of the network. The initiator peer doesn't need to be the one that backed up the file that is asking to restore.

```
$IP $PORT RESTORE $FILE_NAME
```

- `IP` - IP of the host to invoke the restore
- `PORT` - PORT of the host to invoke the restore
- `FILE_NAME` - Name of the file to restore

Delete

The delete protocol is responsible to delete backed up files that exist at any point of the network. The initiator peer doesn't need to be the one that backed up the file that is asking to delete.

```
    $IP $PORT RECLAIM $PEER_ID $SPACE
```

- `IP` - IP of the host to invoke the delete
- `PORT` - PORT of the host to invoke the delete
- `FILE_NAME` - Name of the file backed up to delete

Reclaim

The reclaim protocol is responsible to manage the storage of local and other peers.

```
    $IP $PORT RECLAIM $PEER_ID $SPACE
```

- `IP` - IP of the host to invoke the reclaim
- `PORT` - PORT of the host to invoke the reclaim
- `PEER_ID` - ID of the peer which will be reclaimed
- `SPACE` - Size of space to be reclaimed in KB

1.3 Design Choices

Our implementation takes into account most of the requirements of the project, in order to achieve a higher grade:

- Use of a decentralized design, with Chord;
- Use of JSSE with SSLSockets for secure communication;
- In terms of scalability, use of a Chord implementation, Java NIO and thread pools;
- Regarding the fault tolerance features, use of Chord's fault-tolerant features.

In this report we will further explain our solution to the multi-threading problem of the project and additionally how we implemented the three enhancements mentioned above.

2 Protocols

As mentioned before, the implemented protocols are backup, restore, delete, reclaim. In the next sections, these protocols will be described in detail.

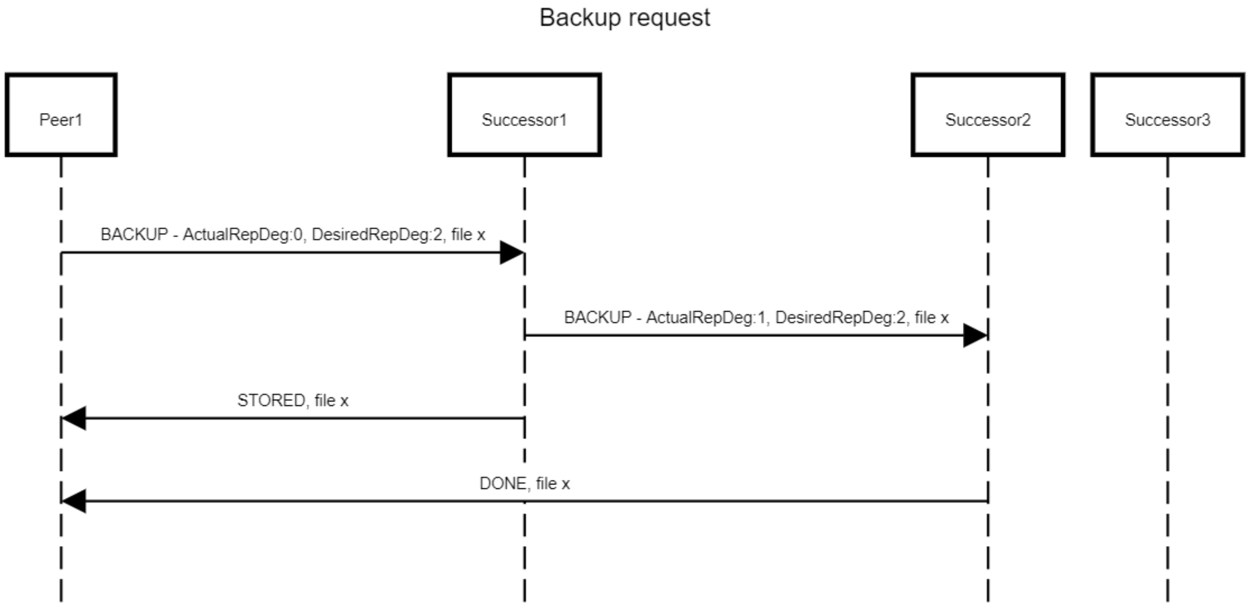
2.1 Backup

To run the BACKUP protocol, it is necessary to run the following command on terminal:

```
$IP $PORT BACKUP $FILE_PATH $REP_DEG
```

Sequence diagrams

Before entering in code details in this section, let's take in consideration the sequence diagram illustrated below for a better understanding of the situation:


Backup request
Peer1
Successor1
Successor2
Successor3
BACKUP - ActualRepDeg:0, DesiredRepDeg:2, file x
BACKUP - ActualRepDeg:1, DesiredRepDeg:2, file x
STORED, file x
DONE, file x

Code explanation

Once the backup of a file has been requested, the current peer will be responsible for initializing the protocol.

The following command invokes the backup function in the Peer that will schedule the `SendBackup` using a Thread Pool. This is a runnable class that will be responsible for sending the backup message to the successor peer.

Listing 2.1: Backup initiate

```
/**
 * Send the backup message
 */
@Override
public void run() {
    try {
        byte[] byteArr = FileHandler.readFile(filePath);
        if (byteArr == null) return;
        Logger.REQUEST(this.getClass().getName(), "Sent message backup");
        MessageBackup message = new MessageBackup(originNode, filePath, byteArr, repDeg, 0);
        Main.threadPool.submit(new SendMessage(ip, port, message));
    } catch (IOException e) {
        Logger.ERR(this.getClass().getName(), "Not possible to send backup message");
    }
}
```

After the backup message was sent, the successor peer will receive that message in the ChordServer.

Listing 2.2: Backup in the server

```
case BACKUP:
    var messageBackup = ((MessageBackup) message);
    if (message.getPortOrigin() == port) {
        backupEndLog(messageBackup.getDesiredRepDeg(), messageBackup.getActualRepDeg());
    } else {
        Main.threadPool.execute(new ProcessBackup(messageBackup));
    }
    break;
case STORED:
    Logger.ANY(this.getClass().getName(), "Received stored from peer " +
        message.getOriginNode().getId());
    break;
case DONE_BACKUP:
    Main.state.printStoredFiles();
    var messageDoneBackup = (MessageDoneBackup) message;
    backupEndLog(messageDoneBackup.getDesiredRepDeg(), messageDoneBackup.getActualRepDeg());
    break;
```

When the message backup is received, if the original PORT and IP is equal to the current peer PORT and IP, it means that the message reached again the initiator, thus nothing will be done and the protocol is finished.

Otherwise, the message will be processed by invoking the `ProcessBackup` class. In this runnable class, it checks if the current peer has enough space to save the file and, if so, it will be saved.

By saving the file, there are two situations that might happen according to the current replication degree of the file:

- If the actual replication degree of the file has achieved the expected, a `DONE` message will be sent to the initiator. In this case, it's known that the backup process has achieved the desired replication degree.
- In case the current replication degree of a file has not yet achieved the desired replication degree, by the time it's backed up in the current peer, the `BACKUP` message will be sent to the successor.

Listing 2.3: Saving file

```java
Logger.ANY(this.getClass().getName(), "Received BACKUP");
int desiredRepDeg = message.getDesiredRepDeg();
String fileName = message.getFileName();
String hash = message.getHash();

if (Main.state.getOccupiedSize() + message.getBytes().length > Main.state.getMaxSize()){
    Logger.INFO(this.getClass().getName(), "Not enough space to save " + fileName + "... Parsing to successor.");
    sendToSuccessor(message.getActualRepDeg());
}
if (Main.state.getStoredFile(hash) != null) {
    sendToSuccessor(message.getActualRepDeg());
    return;
}

Main.state.addStoredFile(hash, message.getBytes().length);

saveFile(hash, message);
int actualRepDeg = message.getActualRepDeg() + 1;

if (actualRepDeg == desiredRepDeg) {
    sendBackupDone(hash);
} else {
    sendToSuccessor(message.getActualRepDeg() + 1);
    storedMessageOrigin(hash);
}
```

The content of each message has specific fields that informs the actual replication degree of a file, the port and ip of the initiator peer and the desired replication degree. Each time the message is parsed to a successor, the actual replication degree might change.

2.2 Restore

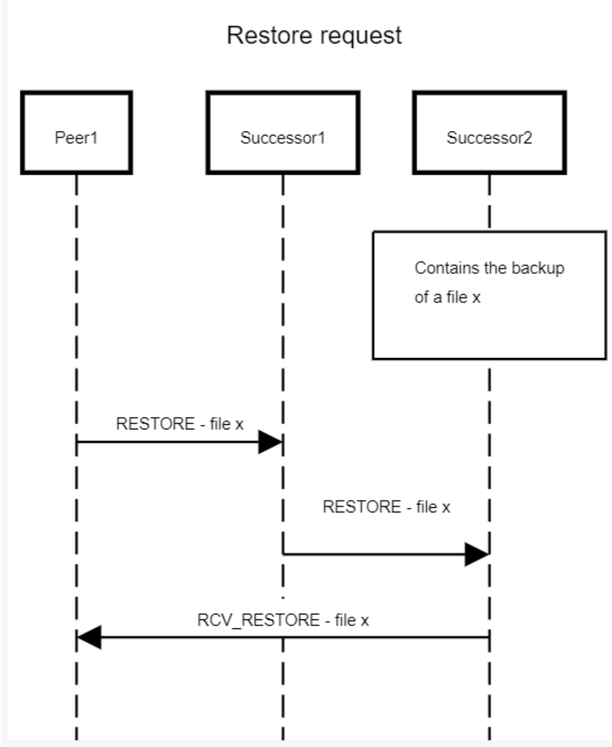
The restore protocol is responsible for restoring a file that has been previously backed up in the network. To initiate this protocol, as mentioned in the previous section, the client must run the java application with the following arguments:

```
$IP $PORT RESTORE $FILE_NAME
```

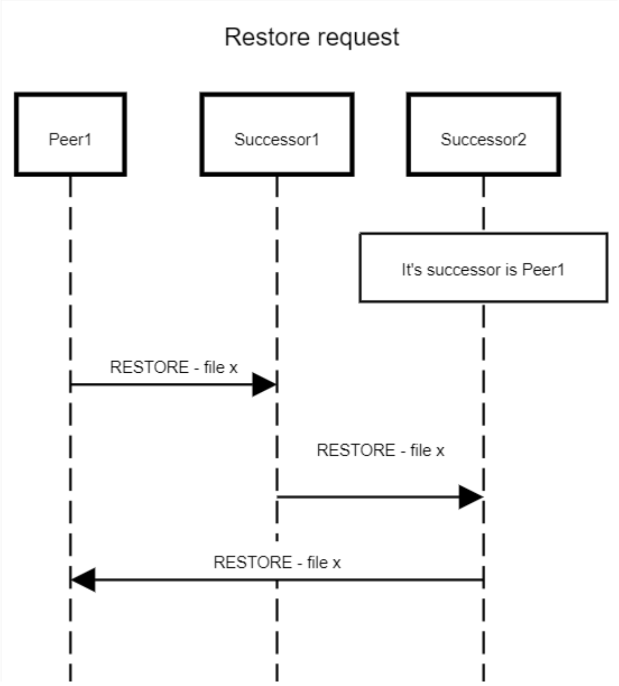
Sequence diagrams

To illustrate how the restore protocol works, consider the following sequence graphs.

The first graph shows how the restore protocol works in case of success:



When there isn't a file in the network, the events succeed in the following way:



Code explanation

After the RMI invokes the respective function in the Peer, it will send the restore message to the successor, just like in the backup protocol.

Listing 2.4: Restore in the server

```
case RESTORE:
    if (message.getPortOrigin() == port) {
        Logger.ANY(this.getClass().getName(), "Can't restore the file");
    } else {
        Logger.ANY(this.getClass().getName(), "Received RESTORE.");
        Main.threadPool.execute(new HandleRestore((MessageRestore) message, port));
    }
    break;
// Confirmation.
case RCV_RESTORE:
    Logger.ANY(this.getClass().getName(), "Received RCV_RESTORE.");
    Main.threadPool.execute(new ProcessRestore((MessageRcvRestore) message));
    break;
```

In case a peer receives the `RESTORE` message, it's checked if the message origin PORT and IP is equal to the current peer PORT and IP. If they're equal it means that the message reached again the initiator and therefore nothing will be done. In this situation, it's considered that the restore operation has failed, since the message went through all the network, but no peer with the file requested was found.

On another hand, if a peer that has backed up the file is found, the `RESTORE` message will be handled in the HandleRestore. If the current peer doesn't have the file to be restored, it parses the same message to its successor. Otherwise it will send the `RCV_RESTORE` message to the initiator peer with the file to be restored, invoking the Process Restore that will write the file to the file system.

Listing 2.5: Handle restore

```java
InfoNode suc = Main.chordNode.getSuccessor();

if (Main.state.getStoredFile(hash) != null) {
    MessageBackup mess = FileHandler.ReadObjectFromFile(Singleton.getBackupFilePath(hash));
    MessageRcvRestore messageRcvRestore = new MessageRcvRestore(message.getOriginNode(),
        mess.getBytes(), mess.getFileName());
    Main.threadPool.submit(new SendMessage(message.getIpOrigin(), message.getPortOrigin(),
        messageRcvRestore));
} else {
    Main.threadPool.submit(new SendMessage(suc.getIp(), suc.getPort(), message));
}
```

2.3 Delete

To run the DELETE protocol, the client must run the application with the following arguments: `$IP $PORT DELETE $FILE_NAME`

After the RMI invokes the respective function in the Peer, it will send the delete message to the successor, just like in the previous protocols.

The initiator peer will first verify if it has the file that is supposed to delete backed up. If it have it will remove that file. After that, the Delete protocol will send a delete message to all elements of his finger table.

Additionally, it is important to mention that the State stores a list of deleted files that the peer has processed recently, not allowing him to reprocess the file if it was already processed in the last 3 seconds. After this 3 seconds, the hash of the file will be removed from that list.

Listing 2.6: Initiating delete

```
Main.state.addBlockDeleteMessages(hash);
if (Main.state.getStoredFile(hash) != null) {
    Main.state.removeFile(hash);
    FileHandler.deleteFile("peers/" + Main.chordNode.getId() + "/backup/", hash + ".ser");
}

Main.schedulerPool.schedule(new RemoveBlockDelete(hash), 3 * 1000L, TimeUnit.MILLISECONDS);
MessageDelete message = new MessageDelete(originNode, hash);

var fingerTableOrder = Main.chordNode.getFingerTableOrder();
var fingerTable = Main.chordNode.getFingerTable();

var iterator = fingerTableOrder.descendingIterator();
while(iterator.hasNext()) {
    BigInteger next = iterator.next();
    InfoNode infoNext = fingerTable.get(next);
    Main.threadPool.submit(new SendMessage(infoNext.getIp(), infoNext.getPort(), message));
}
```

The peers that received the DELETE message will verify if the it hasn't processed a delete yet and if not, they will do the process above again. In other words, they will parse the delete message to all the elements in the finger table and delete the file if it has been stored.

Our solution stores the recently `DELETE` messages received and block future messages with of the same file for two reasons:

- Consider that many entries in a finger table may have the same successor. By sending a `DELETE` message to all the entries in the finger table, it's known that a peer might receive and process the delete protocol multiple times. This is not effective nor desired. By blocking future `DELETE` messages of a same file we're improving the efficiency of the protocol.
- Another essential fact is that by blocking we are avoiding the other peers of propagating those messages to the finger table. With this in mind, the message will "live" in the network even if all the peers have already received the message.

2.4 Reclaim

The reclaim protocol is responsible for squeezing the maximum size that a peer can store. It's generally used when the user wants to minimize the space that is used by this system.

To run the RECLAIM protocol, it is necessary to run the following command on terminal: `$IP $PORT RECLAIM $PEER_ID $SPACE`

Once the client has requested a peer to initiate the request protocol, the peer will try to find the peer that will have the space reclaimed, thus the system uses an approach similar to the delete protocol:

- The `RECLAIM` message will be sent to every instance in the finger table of the initiator. This message has an unique generated identification. Let's call it `reclaimID`.
- A peer that has received the `RECLAIM` stores its `reclaimID` in a structure called `blockedReclaimMessages`. After a certain amount of time the `reclaimID` is automatically removed from this structure.
- In case the peer receives a `RECLAIM` message, the system will check if there's an instance of the message `reclaimID` in the `blockedReclaimMessage` structure. If it's verified the presence of the id in the this structure then the message is blocked and nothing is done. Otherwise, it's checked if the current peer is the destination of the message. If so, the peer will proceed with the reclaim operation, otherwise it will repeat the process above by sending the message to every instance of its finger table.

The approach described above is represented by the following code:

Listing 2.7: Reclaim

```
   try {
    // Message is blocked
    if (Main.state.getBlockReclaimMessages(messageReclaim.getMessageId()) != null) {
        return;
    } else if (Main.chordNode.getId().equals(messageReclaim.getTargetId())) {
        // Proceed with the reclaim and add message to the blocked structure.
        Logger.INFO(this.getClass().getName(), "Received reclaim");
        Main.state.addBlockReclaimMessages(messageReclaim.getMessageId());
        Main.schedulerPool.schedule(new UnbanReclaim(messageReclaim.getMessageId()), 3 * 1000L, TimeUnit.MILLISECONDS);
        reclaim();
    } else {
        // It's not the destine, so add message to the blocked structure and resend the reclaim.
        Main.state.addBlockReclaimMessages(messageReclaim.getMessageId());
        Main.schedulerPool.schedule(new UnbanReclaim(messageReclaim.getMessageId()), 3 * 1000L, TimeUnit.MILLISECONDS);
        Main.threadPool.submit(new SendReclaim(this.messageReclaim));
    }
}catch(Exception e){
    e.printStackTrace();
}
```

As mentioned in the beginning of this section, the reclaim squeezes the maximum amount of data that may be stored in the peer. However, in case there are stored files in the reclaimed peer and the occupied space exceeds the maximum space of information that a peer can store, it implies that files might be deleted.

In order to preserve the replication degree of files in the system, after proceeding with the file deletions, first the peer must choose which files should be deleted:

Listing 2.8: Choose files to delete

```
private void chooseFilesToDelete(){
    Main.state.getStoredFiles().forEach((hash, size)->{
        int occupiedSize = Main.state.getOccupiedSize();
        if (occupiedSize <= Main.state.getMaxSize())
            return;

        Integer fileSize = Main.state.removeFile(hash);
        if (fileSize != null)
            toDelete.add(hash);
    });
}
```

After choosing the files, the peer read those and request the backup those by initiating the same protocol discussed in the section 2.1. After initiating the backup protocol, the peer is immediately deleted.

Listing 2.9: Reclaiming space

```
private void reclaim(){
    Main.state.setMaxSize(messageReclaim.getSize());
    chooseFilesToDelete();

    String successorIp = Main.chordNode.getSuccessor().getIp();
    int successorPort = Main.chordNode.getSuccessor().getPort();
    InfoNode infoNode = Main.chordNode.getInfoNode();
    for (var hash: toDelete){
        Main.threadPool.submit(new ResendBackupFile(successorIp, successorPort, hash,
            infoNode, 1));
        Main.schedulerPool.schedule(new DeleteBackupFile(hash), 2000,
            TimeUnit.MILLISECONDS);
    }

    Logger.ANY(this.getClass().getName(), "Reclaim done with success.");
}
```

3 Concurrency Design

To allow the concurrent execution we used

- Threads (specifically Runnables and Callable which we will approach later), Thread Pools;
- Java NIO;
- Concurrent HashMap.

3.1 Threads and Thread Pools

In this project, some threads are scheduled to run after a fixed time and called with Thread Pools as it's possible to see an example bellow when we run some functions from the Chord protocol.

Listing 3.1: Periodic functions

```
public void initPeriodicFunctions() {
     // Stabilize
     Main.schedulerPool.scheduleWithFixedDelay(new GetPredecessor(), 100,
         Singleton.STABILIZE_TIME * 1000L, TimeUnit.MILLISECONDS);
     // Check predecessor
     Main.schedulerPool.schedule(new CheckPredecessorOrchestrator(), 5000,
         TimeUnit.MILLISECONDS);
     // Schedule the fix fingers.
     Main.schedulerPool.scheduleWithFixedDelay(new FixFingerOrchestrator(), 100,
         Singleton.FIX_FINGERS_TIME * 1000L, TimeUnit.MILLISECONDS);
 }
```

3.2 Java NIO

Listing 3.2: NIO

```
 /**
 * Read a file from the filesystem and return the file information in bytes
 * @param filePath Path of the file to read
 * @return byte[] This return the bytes of the file
 */
 public static byte[] readFile(String filePath) throws IOException {
     Path path = Paths.get(filePath);

     if (Files.size(path) > Integer.MAX_VALUE)
         Logger.ERR("network.etc.FileHandler", "File too large to be read");
     try {
         return readAllBytes(path);
     } catch (Exception e) {
         Logger.INFO("network.etc.FileHandler", "File does not exist, skiping...");
     }
     return null;
 }
```

Java NIO enables non-blocking IO. Because of that, if different threads try to read the same file it will not block.

3.3 Concurrent HashMap

Like in the first project, the Concurrent Hash Map is used to allow simultaneous writing in the Hash. For example, at the class State (class responsible for storing the Peer state), the Peer can have the State updated by multiple threads simultaneous.

For the blocked Delete Messages and Blocked Reclaim Messages, we have used a `ConcurrentHashMap` where the key is equal to its value, since a `ConcurrentHashSet` doesn't exist. Other concurrent structures would fit to the simple purpose of storing information, since the complexity to access and delete elements from those are higher than the time complexity in a `ConcurrentHashMap`

Listing 3.3: ConcurrentHashMaps

```java
    // Blocked Messages
    private ConcurrentHashMap<String, String> blockedDeleteMessages;
    private ConcurrentHashMap<Integer, Integer> blockedReclaimMessages;
    // Files stored.
    private final ConcurrentHashMap<String, Integer> storedFiles;
```

4 JSSE

The comunication between peers is done using the TCP protocol with JSSE. We use JSSE in every message sent between peers to ensure their safety. The keys and credentials are specified when the program is executed.

Each peer contains an instance of a SSLServerSocket. This instance works as a server to receive the messages addressed to this node in specific. In order to make the use of the SSLServerSocket smooth, we have implemented a class to encapsulate the methods and operations of it:

Listing 4.1: SSLServerSocket

```java
public class SSLServerConnection implements Connection {
    private InetAddress ip;
    private final SSLServerSocket sslServerSocket;

    /**
     * SSL Server Connection
     * @param port
     */
    public SSLServerConnection(int port) throws IOException {
        System.out.println("New ServerConnection on port " + port);
        SSLServerSocketFactory sslServerSocketFactory = (SSLServerSocketFactory)
            SSLServerSocketFactory.getDefault();

        sslServerSocket = (SSLServerSocket) sslServerSocketFactory.createServerSocket(port);
    }

    /**
     * Accept SSLSocket
     */
    public SSLSocket accept() throws IOException {
        return (SSLSocket) sslServerSocket.accept();
    }

    /**
     * Get Port of ssl Server
     * @return int port of ssl server
     */
    public int getPort() {
        return sslServerSocket.getLocalPort();
    }

    /**
     * Get the ip of ssl server
     * @return InetAddress ip
     */
    public InetAddress getIp(){
        return ip;
    }
}
```

This class is initialized with the server `ChordServer` and, for each message, a new instance of `SSLSocket` is created by executing the `accept()` function.

As well as the `SSLSocketServer` has its own class to handle operations, a class dedicated to the

SSLSocket was also implemented:

Listing 4.2: SSLConnection

```
public class SSLConnection implements Connection{
    private final SSLSocket sslSocket;
    private final ObjectOutputStream out;
    private final ObjectInputStream in;

    /**
     * SSL Connection class constructor
     * @param ip ip of the connection
     * @param port port of the connection
     */
    public SSLConnection(InetAddress ip, int port) throws IOException {
        SSLSocketFactory sslSocketFactory = (SSLSocketFactory)
            SSLSocketFactory.getDefault();

        this.sslSocket = (SSLSocket) sslSocketFactory.createSocket(ip, port);

        this.out = new ObjectOutputStream(sslSocket.getOutputStream());
        this.in = new ObjectInputStream(sslSocket.getInputStream());
    }

    /**
     * Accept the sslsocket
     */
    public SSLSocket accept() {
        return sslSocket;
    }

   ...
```

While the class SSLConnectionServer is mainly used to receive information, the SSLConnection is mainly used to send messages.

5 Scalability

In this topic, it will be approached how the system handles the growing amount of peers joining the network.

This system uses the Chord protocol to manage the system distribution and its main idea has been adapted to fit the actual context of this project. To understand better the concepts that will be referred here we recommend reading the chord paper: *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*.

This network implements a horizontally scalable system using peer-to-peer communication. In this way, it's possible to add as many peers as necessary and they can leave the network as the user wishes. All the machines on this system play equal roles and do the same thing, making possible the equal distribution in a single layer.

Joining the network

Each peer in this system has a unique identification. The collision of ids is not handled. However, to avoid that, we have implemented a large m, to prevent this to happen, where m is nothing more than the size of the finger table in the chord system.

Now, considering that all peers have a different identification, to join the network, the peer must know the information about an entry in the chord system. With this in mind, it's possible to request the lookup process to this entry, so that the new peer has a successor. The code for the lookup process can be checked below:

Listing 5.1: Stabilize

```java
    @Override
    public void run() {
        try {
            BigInteger targetId = message.getTargetId();
            BigInteger peerId = Main.chordNode.getInfoNode().getId();
            BigInteger successorId = Main.chordNode.getSuccessor().getId();

            if (Singleton.betweenSuccessor(targetId, peerId, successorId)) {
                MessageSuccessor messageSuccessor = new
                    MessageSuccessor(Main.chordNode.getInfoNode(), targetId,
                    Main.chordNode.getSuccessor(), this.returnType);
                new SendMessage(message.getIpOrigin(), message.getPortOrigin(),
                    messageSuccessor).call();
            } else closestPrecedingNode(targetId);

        } catch (Exception e) {
            Logger.ERR(this.getClass().getName(), "Error on processing lookup...");
        }
    }
```

Once a successor is defined, the newly joined peer will adjust itself by executing fixfingers and stabilize functions:

Listing 5.2: FixFingers

```
        try {
            InfoNode successor = Main.chordNode.getSuccessor();

            if ((successor == null) || (successor == Main.chordNode.getInfoNode()))
                return true;

            BigInteger currentId = Main.chordNode.getId();
            BigInteger nextId = new BigInteger(String.valueOf((long) Math.pow(2, currentNext
                - 1)));
            BigInteger targetId = currentId.add(nextId);
            MessageLookup messageLookup = new MessageLookup(Main.chordNode.getInfoNode(),
                targetId, MessageType.FIX_FINGERS);
            new SendMessage(successor.getIp(), successor.getPort(), messageLookup).call();
        } catch (Exception e) {
            Logger.ERR(this.getClass().getName(), "Error on fix fingers.");
            Main.chordNode.fixSuccessor();
        }
        return true;
    }
```

Listing 5.3: Stabilize

```
         try {
            BigInteger currentId = Main.chordNode.getInfoNode().getId();

            if (Objects.isNull(sucPredecessor))
                return;

            if (Singleton.betweenPredecessor(sucPredecessor.getId(), currentId,
                Main.chordNode.getSuccessor().getId())) {
                Main.chordNode.setSuccessor(sucPredecessor);
            }

            MessageInfoNode message = new MessageInfoNode(Main.chordNode.getInfoNode(),
                MessageType.NOTIFY, Main.chordNode.getInfoNode());
            new SendMessage(Main.chordNode.getSuccessor().getIp(),
                Main.chordNode.getSuccessor().getPort(), message).call();

        } catch (Exception e) {
            e.printStackTrace();
            Logger.ERR(this.getClass().getName(), "Error on stabilizing.");
        }
```

The NIO operations for the server reading and writing messages were not implemented in this project. We chose not to due to the fact that despite the IO operations does not offer great scalability, it is more efficient in terms of reading and writing.

6 Fault Tolerance

This topic will approach on how the system implements fault tolerance, in other words, how it handles the event of one peer leaving the network or occurring errors on peers communication.

In the context of the chord protocol, when it's not possible to establish a communication between entries in the finger table, the `SSLSocket` will raise an error, since the communication has been refused. In this scenario, it's expected that the system continues to work and also recovers itself from the actual state.

Predecessor error recovery

By sending a message to the predecessor node at a fixed rate, it's verified that the predecessor is available and communicable. In case of error, the exception will be handled in the following way:

Listing 6.1: Stabilize

```
    try {
        InfoNode predecessor = Main.chordNode.getPredecessor();
        new SendMessage(predecessor.getIp(), predecessor.getPort(), new OK()).call();
    }catch(Exception e){
        Main.chordNode.setPredecessor(null);
    }
```

By setting the predecessor as null, sending messages to the successor will be avoided. Eventually, the stabilize algorithm will eventually change the successor to a new one.

Successor error recovery

When a peer's successor leaves the network, the procedure applied is more complex than just setting its value to null: it's immediately replaced by the first entry in the finger with the id different from the peer id that has recently left the network.

Listing 6.2: fixSuccessor

```
/*
 * Case it's not possible to communicate with the successor, the
 * it's necessary to find a successor in the finger table that is not
 * the actual successor.
 */
public void fixSuccessor() {
    BigInteger[] fingerTableOrderArray = fingerTableOrder.toArray(new BigInteger[0]);
    for (BigInteger key : fingerTableOrderArray) {
        if (!fingerTable.get(key).getId().equals(successor.getId()))
            setSuccessor(fingerTable.get(key));
    }
    // The only node in the network.
    setSuccessor(infoNode);
}
```

In case it's not possible to estimate a successor from the finger table, the own node is set as its successor.

We managed to create this approach by thinking in the scenario where all the nodes leave the network and remaining just one peer in it. In this case, the successor of the current peer must be itself.

Recovering the finger table

By applying the last two approaches for error recovering, the `fixFingers` function will eventually be fixed. In case a request is made to one node that isn't active anymore, a short message will be displayed in the terminal, the message will not be propagated to other peers, but the system will remain working.