U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEIC 3rd year

SDIS

June 2021

Distributed Backup Service for the Internet

Report

(Project 2)

Class 4 - Group 26

Nuno Castro Silva - up201404676

Rui Filipe Mendes Pinto - up201806441

Tiago Gonçalves Gomes – up201806658

Abstract

This report aims to explain all the implemented features in this project, namely the implemented protocols, JSSE, concurrency design, scalability, and fault-tolerance.

The project was developed within the scope of the Distributed Systems course.

1. Overview

In this report, we explain all important aspects of the project, in each specified section, regarding our implementation of the **Chord** protocol, a decentralized distributed system. We begin with the definition of all the different message types, naming all parameters passed on each and explaining whom they're sent by and to, and what their purpose is.

Secondly, we describe how our design deals with concurrency on the different services, making use of **thread pools** and **Java NIO**. The next section describes our use of **JSSE** on communications between nodes, making use of the **SSLEngine** class.

The Scalability section describes our implementation of the **Chord** protocol, such as how it keeps its nodes stabilized and updated, and what the process of adding a new node to the network is like.

Finally, in the fault tolerance section, we explain our implementation of an adequate response to nodes who suddenly stop working, in our decentralized design, for it to keep running in case of single-point failures.

2. Protocols

We implemented the backup, delete and restore protocols. In the section below, we will describe the interfaces and messages used by them.

2.1. RMI Interface

For the client using the *TestApp* to give the desired instructions to the node, namely the backup, delete and restore commands, we are using the Remote Method Invocation (RMI), a Java API that performs remote method invocation, providing remote communication between Java programs. For that, we have an interface *RMIService*, that extends Remote:

```
public interface RMIService extends Remote {
    void backup(String path, int replicationDeg) throws RemoteException;
    void delete(String path) throws RemoteException;
    void restore(String path) throws RemoteException;
}
```

(**RMIService.java, lines: 6-10**)

This interface is implemented by the *Node* class, forcing it to implement the backup, delete and restore methods. Furthermore, we always Serialize and Deserialize the Node's storage when it goes offline and online, respectively.

2.2. Messages

For the communication between the nodes, we are using TCP, with SSLEngine.

2.2.1. ADD_NODE

{Origin IP} {Origin Port} {Origin ID} ADD_NODE {To Add Node IP} {To Add Node Port} {To Add Node ID}

The goal of this message is to add a recently created node to the chord ring network. Firstly, it is sent to the gate node (the node that every node needs to contact to enter the network). Each node that receives this message starts the *Add Node* procedure (**AddNode.java**) in a new thread. During this procedure, it is checked (according to the node's IDs) whether the new node is between the current node and the node's successor, and if not, an *ADD_NODE* message is sent to the element that is nearest to the desired position (fetched from the finger table), which will start this procedure again, until the new node finds its correct position. When the correct position is found, the new node is informed of its successor and predecessor, and the successor of the new node is informed of its new predecessor.

2.2.2. ADD_NODE_SET_PRED

{Origin IP} {Origin Port} {Origin ID} ADD_NODE_SET_PRED {New Predecessor Node IP} {New Predecessor Node Port} {New Predecessor Node ID}

This message is used in the *Add Node* procedure, described above, as well as when there is a node that failed (fault-tolerance). Its goal is to inform a node of its new predecessor. Furthermore, this message triggers the start of the *FileRedistribution* procedure, to update the position of the current node's files, since there are files that could stay in the new predecessor.

2.2.3. ADD_NODE_SET_SUCC

{Origin IP} {Origin Port} {Origin ID} ADD_NODE_SET_SUCC {New Successor Node IP} {New Successor Node Port} {New Successor Node ID}

This message is used in the *Add Node* procedure. Its goal is to inform a node of its new successor.

2.2.4. ASK_RESTORED_FILE

{Origin IP} {Origin Port} {Origin ID} ASK_RESTORED_FILE {File ID}

This message is used in the *Restore* protocol. Its goal is to allow a node to ask for a file that it had previously backed up to the network. The node that receives this message checks if it has the desired file, and acts accordingly to that: if it has the file, it sends it to the node that started the restore protocol; if it does not have the file, it sends a message to the node that started the restore protocol, informing that it does not have the file.

2.2.5. RESTORED_CONNECTION

{Origin IP} {Origin Port} {Origin ID} RESTORED_CONNECTION

This message is used in the *SendRestoredFile* procedure. Its goal is to send the current node contact information, namely the IP and port, to the node that has the file that the node is looking for. That way, the two nodes can start a new connection to exchange the desired file.

2.2.6. FILE_CONNECTION

{Origin IP} {Origin Port} {Origin ID} FILE_CONNECTION

This message is used in the *Backup* protocol, as well as in the *FileRedistribution* procedure. Its goal is the same as the *RESTORED_CONNECTION* message, described above, but we used two different types of messages for the same purpose for the node that receives it to know the current protocol and to act accordingly to that.

2.2.7. DELETE_FILE

{Origin IP} {Origin Port} {Origin ID} DELETE_FILE {File ID}

This message is used in the *Delete* protocol. Its goal is to allow a node to delete from the network a file that it had previously backed up.

2.2.8. FILE

{Origin IP} {Origin Port} {Origin ID} FILE {File ID} {File Name} {File Replication Degree} {File Replication Number} {Remaining Chunks} {Chunk Encoded Content}

This message is used in the *Backup* protocol, as well as in the *FileRedistribution* procedure. Its goal is to allow a node to send a chunk of a file to another node. The first thing the node needs to do to be able to send the file is to encode its content in *Base64* format. By using this format, we can encode the binary content of the files to string, making the task of sending the file easier. Since the maximum size of the *ByteBuffer* used in the *SSLEngine* protocol is limited to approximately 16.7kB, there is a need to divide the encoded file content string into chunks of strings. We chose a chunk size of 16200 bytes so that the remaining message can have a maximum of 500 bytes. Having the file content divided into chunks of strings, the node starts a new connection with the node that will receive the file, and sends each chunk through a *FILE* message.

2.2.9. QUERY

{Origin IP} {Origin Port} {Origin ID} QUERY {Lookup ID}

This message is used in all the protocols. Its goal is to find the node that corresponds to a certain ID. In the *Backup* protocol, this message is used to find the node that should store a certain file, in the *Delete* protocol, to find the node that has a certain replication of a file, and in the *Restore* protocol, to find the node that is storing the desired file. When a node receives a *QUERY* message, it checks whether the lookup ID corresponds to a position in between the nodes of its successor. If it corresponds, then a *QueryResponse* message is sent to give the node that originally sent the *Query* all the needed information. If it does not correspond, then a new *Query* message is sent to the node of the current node's finger table that is nearest to the lookup ID.

2.2.10. QUERY_RESPONSE

{Origin IP} {Origin Port} {Origin ID} QUERY_RESPONSE {Lookup ID} {Looked up ID}

This message is used in all the protocols. Its goal is to respond to the *Query* message with all the information needed, namely the origin IP, port and ID, the lookup and looked-up IDs.

2.2.11. RESTORED_FILE

[1] {Origin IP} {Origin Port} {Origin ID} RESTORED_FILE {File ID}

or

[2] {Origin IP} {Origin Port} {Origin ID} RESTORED_FILE {File ID} {File name} {File Replication Degree} {File Replication Number} {File Content}

This message is used in the *Restore* protocol. Its goal is to allow the node that has the file that is being restored to send the file to the recipient node (message version [2]). In case the node that received this message does not store the desired file, it sends the version [1] to inform the node that started the *Restore* protocol. Version [2] is using a connection and procedure similar to the one used by the *File* message.

2.2.12. SET_PRED

{Origin IP} {Origin Port} {Origin ID} SET_PRED

This message is used in the *Stabilize* procedure. Its goal is to allow a node to check if its predecessor is the node who sent the message and if not, set the sender's successor accordingly.

2.2.13. SET_SUCC

{Origin IP} {Origin Port} {Origin ID} SET_SUCC

This message is used in the *Stabilize* procedure. Its goal is to allow a node to tell another who is its successor.

3. Concurrency Design

In this section, we explain how we deal with the concurrency of the different services present.

It should be noted that, for a Chord network to be completely stabilized and ready to execute any of its Backup/Delete/Restore protocols, the Client **must wait a few seconds** after the addition of nodes until all nodes update its successors/predecessors and finger tables correctly. This is due to the stabilization process being run periodically, and as such, requires some time until a Peer has sent and received all necessary messages between its neighbors to be correctly updated.

3.1. Thread Pools

The use of thread pools is highly taken advantage of in the implementation of our project. In order to achieve concurrency, all Peers make use of thread pools in order to execute different tasks.

The periodically occurring stabilization (Stabilize and Finger table construction) makes use of the thread pool schedule function, to permit the recurrent background task of updating a Peer's successors, predecessors, and finger table.

```
ThreadPool.getInstance().execute(Node.listener);
ThreadPool.getInstance().scheduleAtFixedRate(new BuildFingerTable(),  initialDelay: 0,  period: 750, TimeUnit.MILLISECONDS);
ThreadPool.getInstance().scheduleAtFixedRate(new Stabilize(),  initialDelay: 0,  period: 750, TimeUnit.MILLISECONDS);
```

Execution and scheduling of a Peer's Listener and Stabilization methods, respectively
(Node.java, lines: 69 - 71)

Also, whenever a Peer needs to prepare and send a message to other Peers, the thread pool execute function is used to allow for managing multiple asynchronous messages sending tasks so that we can have improved performance.

Each Peer also has a Listener class that, whenever it receives an incoming message, also calls for execution of a specific handler for each message.

```java
public void handleMessage(String msg) {
    String messageType = msg.trim().split( regex: "\\s+")[3];

    switch (messageType) {
        case "QUERY" -> ThreadPool.getInstance().execute(new ReceivedQuery(msg));
        case "QUERY_RESPONSE" -> ThreadPool.getInstance().execute(new ReceivedQueryResponse(msg));
        case "SET_PRED" -> ThreadPool.getInstance().execute(new ReceivedSetPredecessor(msg));
        case "SET_SUCC" -> ThreadPool.getInstance().execute(new ReceivedSetSuccessor(msg));
        case "ADD_NODE" -> ThreadPool.getInstance().execute(new ReceivedAddNode(msg));
        case "ADD_NODE_SET_SUCC" -> ThreadPool.getInstance().execute(new ReceivedAddNodeSetSuccessor(msg));
        case "ADD_NODE_SET_PRED" -> ThreadPool.getInstance().execute(new ReceivedAddNodeSetPredecessor(msg));
        case "DELETE_FILE" -> ThreadPool.getInstance().execute(new ReceivedDeleteFile(msg));
        case "ASK_RESTORED_FILE" -> ThreadPool.getInstance().execute(new ReceivedAskRestoredFile(msg));
        case "RESTORED_FILE" -> ThreadPool.getInstance().execute(new ReceivedRestoredFile(msg));
        case "FILE_CONNECTION" -> ThreadPool.getInstance().execute(new ReceivedFileConnection(msg, isRestoredFile: false));
        case "RESTORED_CONNECTION" -> ThreadPool.getInstance().execute(new ReceivedFileConnection(msg, isRestoredFile: true));
        default -> System.err.println("Unknown Message Type:" + messageType);
    }
}
```

Handler of a received message by a Peer's Listener **(Listener.java, lines: 45 - 63)**

3.2. Java NIO

The use of the Java NIO library for non-blocking I/O operations is achieved in two important aspects of our project.

The first being in the communications between sockets (**socketChannels**), allowing easier control over the communication between Peers. With its non-blocking mode, a thread can request reading data from a channel, and only get that which is available, or nothing if no data is available, rather than remain blocked waiting for a response, allowing the thread to go on with its work.

We use this when, for example, a Peer asks another Peer for a file to reclaim, or when a Peer transfers a file to backup. In this case, the Peers establish a connection through NIO sockets, and the transfer of chunks of the file in question is sequentially made through reading and writes implemented with the SSLEngine.

The second use of NIO functions is during a Peer's file reading protocol, where a Peer can read a file from a given path, and store this file within its storage, all while the Peer's thread can do other operations. Both backup and restore protocols take advantage of the non-blocking properties to perform operations on files without blocking the Peer's thread.

```java
public void readFile() {
    try {
        Path fPath = Paths.get(path);
        UserPrincipal fileOwner = Files.getOwner(fPath, LinkOption.NOFOLLOW_LINKS);
        String fileName = fPath.getFileName().toString();
        long lastModified = fPath.toFile().lastModified();

        this.fileSignature = fileName + fileOwner.getName() + lastModified;
        File file = new File(this.path);
        this.data = new byte[(int) file.length()];

        FileInputStream inputStream = new FileInputStream(file);
        inputStream.read(this.data);

        if(this.data.length == 0)
            System.err.println("Data is null sized");
        inputStream.close();

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

Reading of a file using NIO functions **(PeerFileBackedUp.java, lines: 39 - 60)**

4. JSSE

Our project uses Java Secure Socket Extension (JSSE), in every communication between network nodes to assure the security of the communication. We have three important classes: **SSLPeer**, **SSLClient,** and **SSLServer**.

The last two extend the first one and they are both used for the implementation of a Java SSL/TLS server and client, making use of the **SSLEngine** class and also Java NIO for the communication between sockets, as explained in the previous topic. The use of the SSLEngine class allows us to have more control in the low-level part of the communication. This means that we can configure, for example, the way the handshake is performed, the read and write operations to the sockets as well as parts regarding application buffers and network buffers. All of this leads us to implement parts of the protocol itself.

During the project, we always use these classes for communication between nodes. And in case a Backup or a Restore is requested, we set up a special connection for the file transfer. For “normal” node messages in operations like Stabilization or Queries regarding the construction of the finger table, we have a unique Listener, which is a dispatcher, that also uses this kind of communication delegating the respective messages to the corresponding handlers.

```java
public Listener(int port) throws Exception {
    String IP = InetAddress.getLocalHost().getHostAddress();
    this.port = port;
    this.connection = new SSLServer(Macros.cypherSuite, IP, port);
}

public int getPort() { return this.port; }

@Override
public void run() {
    while(true) {
        try {
            String receivedMessage = this.connection.start();

            if (receivedMessage != null)
                handleMessage(receivedMessage);
```

Listener connection and message receival based on SSLServer (**Listener.java**, **lines: 22 - 39**)

In the previous image, the invocation of the start function is done by the SSLServer class and it waits for a connection/read of a message, as demonstrated in the following picture.

```java
public String start() throws Exception {
    while(active) {
        this.selector.select();
        Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();
        while(selectedKeys.hasNext()) {
            SelectionKey key = selectedKeys.next();
            selectedKeys.remove();

            if (!key.isValid())
                continue;
            if (key.isAcceptable()) {
                accept(key);
            } else if (key.isReadable()) {
                this.key = key;
                return read((SocketChannel) key.channel(), (SSLEngine) key.attachment());
            }
        }
    }
    return null;
}
```

Connection startup (**SSLServer.java, lines: 43-62**)

The way messages are sent is similar, the only difference is that we set up an SSLClient instead of an SSLServer and perform a write after the connection is set.

```java
public Sender(NodeInfo contactNodeInfo, String msg) {
    try {
        this.contactNodeInfo = contactNodeInfo;
        this.connection = new SSLClient(Macros.cypherSuite, this.contactNodeInfo.getAddress().getHostAddress(), this.contactNodeInfo.getPort());
        this.msg = msg;
    } catch (Exception e) {
        e.printStackTrace();
    }

}

@Override
public void run() {
    try {
        this.connection.connect();
        this.connection.write(this.msg);
```

Sender connection using SSLClient (**Sender.java, lines: 14-29**)

```java
public boolean connect() throws Exception {
    socketChannel = SocketChannel.open();
    socketChannel.configureBlocking (false);
    socketChannel.connect(new InetSocketAddress(remoteAddress, port));
    while(!socketChannel.finishConnect()) {
        // ...
    }

    engine.beginHandshake();
    return doHandshake(socketChannel, engine);
}
```

Connection set up (**SSLClient.java, lines: 33-43**)

To conclude, we require client authentication and for the cipher-suites, we use *TLSv.12*. It's also important to mention that this part of the project was adapted and changed from a public implementation of a "Server and Client implementation with SSLEngine" that is referenced at the end of the project. It's covered by an MIT License that allows dealing with the Software without any restriction especially regarding the rights to "use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software".

5. Scalability

In Distributed Systems, one of the main issues we have to solve has to do with scalability. For this, we implemented the **Chord** protocol to create a decentralized distributed backup service. This guarantees every node in the network is equally important to the node right next to him. The class that represents each node is called **Node.java**.

Whenever a new node joins the network it will have to first compute its id, which is based on its IP and Port, and then run a Consistent Hashing function over that. It first tries to contact the so-called Gate (the only node known to the network's external environment) and then, in case a node with the same ID doesn't exist already in the system, it enters it successfully.

Otherwise, it won't enter the network and it will stay alone in a network only known to itself. After the node is created it executes two important threads every 750 ms, regarding the network stabilization, presented in: **BuildFingerTable.java** and **Stabilize.java**.

The **Stabilize** operation (**Stabilize.java)** is when Node N asks its successor for its predecessor P and decides whether P should be N's successor instead. We simplified this process by sending a SET_PRED message to the node's successor and upon receiving, that node checks if its predecessor is the same as that of the node who sent the message. If it's not, it answers with a SET_SUCC message to adjust the "adjacent" nodes in the right way.

```java
@Override
public void run() {
    NodeInfo senderNodeInfo = new NodeInfo(this.IP, this.port, this.ID);
    if (!senderNodeInfo.equals(Node.predecessor)) {
        try {
            new messages.SendMessages.SendSetSuccessor(Node.nodeInfo, Node.predecessor, senderNodeInfo);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

"Receiver" view upon receiving of the SET_PRED message (**ReceivedSetPredecessor.java, lines: 16-26**)

The operation commonly known as ***Fix Fingers*** presented in **BuildFingerTable.java**, is responsible for updating the node's finger table. If the key of the finger table is less or equal to the successor's ID it set's the entry as the successor of the node. If not, it searches for the closest node with that specific ID (or greater) through the sending of a QUERY message.

```java
@Override
public void run() {
    this.printFingerTable();

    for (BigInteger newCurrentId : Node.fingerTable.getKeysOrder()) {
        if(Node.successor.getId().compareTo(Node.nodeInfo.getId()) > 0 &&
                newCurrentId.compareTo(Node.nodeInfo.getId()) > 0 &&
                newCurrentId.compareTo(Node.successor.getId()) <= 0) { // Não dá a volta
                    Node.addToFingerTable(newCurrentId, Node.successor);
        } else if(Node.successor.getId().compareTo(Node.nodeInfo.getId()) < 0 &&
                (newCurrentId.compareTo(Node.nodeInfo.getId()) > 0 ||
                newCurrentId.compareTo(Node.successor.getId()) <= 0)) { // Dá a volta
            Node.addToFingerTable(newCurrentId, Node.successor);
        } else if(Node.successor.getId().compareTo(Node.nodeInfo.getId()) == 0) {// Successor it's himself
            Node.addToFingerTable(newCurrentId, Node.successor);
        } else { // successor does the same thing to its successor, and so on...
            try {
                if(Node.successor.getId().compareTo(Node.nodeInfo.getId()) != 0)
                    new SendQuery(Node.nodeInfo, Node.successor, newCurrentId);
            } catch (IOException e) {
                System.err.println("Successor down, reorganizing...");
            }
        }
    }
}
```

FixFingers operation (**BuildFingerTable.java, lines: 13-37**)

6. Fault-tolerance

The current implementation of this project provides a basic fault tolerance mechanism, allowing a node to go down without breaking the whole network, removing almost all the single points of failure. However, it is fundamental that the gate node (the one every node needs to contact to enter the network) never goes down, because without it it's not possible to add new nodes. This makes our network partially fault-tolerant.

To check if a node went down, we catch the exceptions thrown when a node is trying to contact another, and check if the node is contacting its successor. If it is, we know the successor of the current node is down, and set its new successor to its subsequent successor and the new predecessor of the new successor of the node to the current node.

```
try {
    this.connection.connect();
    this.connection.write(this.msg);
} catch (Exception e) {
    if (Node.successor.equals(this.contactNodeInfo)) {
        try {
            Node.semaphore.acquire();
            System.out.println("Successor node went down. Initiating fault-tolerance protocol.");
            Node.successor = Node.subsequentSuccessor;
            new SendAddNodeSetPredecessor(Node.nodeInfo, Node.nodeInfo, Node.successor);
        } catch (Exception exception) {
            exception.printStackTrace();
        }
        Node.semaphore.release();
    }
}
```

Fault-tolerance mechanism (**dispatchers.Sender.java, lines: 25-43**)

7. Conclusion

The development of this project was quite challenging because we had to think about many aspects that could happen if we did this or that. Although all the goals of the project were completed, there are still some things that can be improved in future work.

Firstly, we are using the same public and private keys for all the nodes in the network. To solve that, one possibility would be to use a centralized server to store the public keys, but that would break the decentralization of the network. Other possibilities would be too complex for the project's ambit.

Secondly, our delete protocol is not dealing with the peers that could be down during its execution. Once we already implemented that feature in the first project, we thought it didn't make sense to do that again in this project.

Lastly, the fault-tolerance of the network could be improved, by having a backup gate node, in case the gate node fails, and by having a list of successors rather than the subsequent successor of a node, in case the node and its successor fail at the same time.

That being said, we improved our knowledge in Distributed Systems and can now build complex systems that work in various computers that can or not be in the same network.

8. References

[1] *Alex Karnezis*, SSLEngine Example - https://github.com/alkarn/sslengine.example

[2] *James F. Kurose, Keith W. Ross*, Computer Networking, Sixth Edition - https://eclass.teicrete.gr/modules/document/file.php/TP326/%CE%98%CE%B5%CF%89%CF%81%CE%AF%CE%B1%20(Lectures)/Computer_Networking_A_Top-Down_Approach.pdf

[3] Vinh Trương, Testing implementations of Distributed Hash Tables - https://core.ac.uk/download/pdf/16312626.pdf

[4] *Ion Stoica , Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishna*, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, - https://courses.cs.duke.edu//cps212/spring15/15-744/S07/papers/chord.pdf

[5] *Sugih Jamin*, Computer Networks, Lecture 10, University of Michigan - https://web.eecs.umich.edu/~sugih/courses/eecs489/lectures/10-DHT.pdf

[6] *Pedro Alexandre Guimarães Lobo Ferreira do Souto* , Distributed Systems Lectures Material - https://web.fe.up.pt/~pfs/aulas/sd2021/