U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Distributed Backup System for the Internet

SISTEMAS DISTRIBUÍDOS
MIEIC 3º ANO

TURMA 4 - GRUPO 23

Carlos Daniel Rodrigues Lousada - 201806302
João Carlos Fonseca Pina de Lemos - 201000660
José David Souto Rocha - 201806371
Tomas Afonso Alves Louro Mendes - 201806522

# Table of Contents

# 1 Overview

This project was developed during the course of Distributed Systems. Its aim is to implement a Distributed Backup Service for the Internet capable of uploading and downloading files in a Peer-to-Peer context.

The project has 5 distinct protocols, which enables it to backup, delete and restore multiple files, while also being able to reclaim occupied space and show the user the state of the Peer.

Concerning the grading ceiling, our project reaches the maximum available ceiling, due to the implementation of TCP (14 values), JSSE excluding SSLEngine (1 value), and Chord which relates to scalability and fault tolerance (2 values each).

# 2 Protocols

Our project has 5 implemented protocols: backup, delete, state, reclaim and restore. In order to run any protocol, a request must be sent by the Sender Peer (*Sender.java*) and received by the other Peers (*Receiver.java*).

Instead of using a string (previously done in the First Project), the messages in our project are objects of the *Message* class. This means that every protocol has a distinct type of *Message*, which fits the needs of each protocol (*BackupMessage* for *Backup*, *DeleteMessage* for *Delete*, *RestoreMessage* for *Restore*, etc).

```
public interface Message {
    InetSocketAddress PeerAddress = null;
    String protocol = null;
}
```

Listing 1: Message.java

After the *Message* is sent through the respective Socket Address, the Receiving Peer processes the *Message* through the use of the *Message* handling process (*MessageHandler.java*).

```
    ...
    if (received instanceof BackupMessage) {
        this.handleBackup();
    }
    else if (received instanceof StoredMessage) {
        this.handleStored();
    }
    else if (received instanceof RestoreMessage) {
        this.handleRestore();
    }
    else if (received instanceof UpdateFileMessage) {
        this.handleUpdateFile();
    }
    else if(received instanceof DeleteMessage){
        this.handleDelete();
    }
    else if(received instanceof DeletedMessage){
        this.handleDeleted();
    ...
```

Listing 2: Handling Message received - MessageHandler.java

## 2.1 Backup

When a Peer receives a backup request (*Listing 2*) it checks whether the file contained in the *Message* is one of its own (*Listing 3*). We do this due to the fact that a *Backup Message* will end up coming back to itself, since this *Message* is re-transmitted throughout the several Peers. Additionally, the Peer also checks whether the file contained in the received *Message* has already been stored (*Listing 4*).

```
if (rcvd_file.getOriginalPeer().equals(ChordNode.self))
```

Listing 3: Comparison to original Peer - MessageHandler.java

```
if (Peer.getStorage().getStoredFiles().contains(rcvd_file))
```

Listing 4: Check if file is stored - MessageHandler.java

If this premise turns out to be false, a copy of that file is saved on the Peer's disk, and the information concerning that file is stored for later use. If the file has reached its desired replication degree, a *StoredMessage* is sent. If the replication degree still hasn't reached the desired value a new *Backup Message* is sent to the Peer's next Successor (*Listing 5*).

```
if (rcvd_file.getCurrentRepDegree() >= rcvd_file.getRepDegree()) {
    StoredMessage msg =
        new StoredMessage(ChordNode.PeerAddress, rcvd_file);
...
else {
    BackupMessage msg =
        new BackupMessage(ChordNode.PeerAddress, rcvd_file);
    Peer.getExec().execute(
        new SendMessageThread(msg, clientSocket,
        ChordNode.getSuccessor().getSocketAddress()));
```

Listing 5: After Peer Backup - MessageHandler.java

Upon receiving a *StoredMessage* the Sender Peer is able to display a series of *Messages* with information to the user.

## 2.2 Delete

In order to initiate *Delete* protocol, the Sender Peer needs to send a *DeleteMessage* with a file to the Peers that had previously performed a backup of that file. Upon receiving a *DeleteMessage*, a Peer checks if it contains the file present in the *Message*.

```
FileC fileReceived = received.getFile();
    for(int i = 0;
    i < Peer.getStorage().getStoredFiles().size();
    i++){
        if(Peer.getStorage().getStoredFiles()
        .get(i).equals(fileReceived)){
```

Listing 6: Contains file - MessageHandler.java

It roams its files until it finds the correct one, deleting it if so. After the deletion is complete the Peer replies with a *DeletedMessage*.

## 2.3 Restore

To initiate a *Restore protocol*, the Parent Peer sends a *RestoreMessage* to the Peers that had previously performed a *Backup* of the file to restore.

The receiving Peers then send the file back to the initiating Peer, which will then save that file to its disk.

```
FileC fileReceived = received.getFile();
...
RestoreMessage msg = new
    RestoreMessage(ChordNode.PeerAddress, rcvd_file);
ObjectOutputStream out = new
    ObjectOutputStream(clientSocket.getOutputStream());
out.writeObject(msg);
```

Listing 7: Adding file to message - MessageHandler.java

### 2.4 State

*State* is a simple single Peer protocol that displays the information of the Peer to the frontend. This includes the selected Peer's data (name, path, backup id, ...), a list of all of its backed up and stored files, as well as information regarding failed file deletion attempts. This protocol can be seen on file *Peer.java* from lines 143 to 226.

### 2.5 Reclaim

Similar to *State*, *Reclaim* is a single Peer protocol that limits the space provided to the service by the Peer. This protocol is only carried out when the new space limit is lower than the space currently being occupied in the Peer's storage. On the other side, if files need to be deleted in order to guarantee the desired space, the Peer queues a list of stored files by descending order of size, removing them one-by-one until the new space limitations are satisfied. It then goes file by file (of the files included in this list) and starts new backup processes (creates and sends new *BackupMessages*) with a replication degree of 1 for each of the files in the removal queue (*Listings 8*). On the whole, the aim is to find a Peer that can store the file about to be deleted by the initiator Peer.

```
 fileToRemove.setRepDegree(1);
    fileToRemove.clearPeers();
    fileToRemove.setParentPeer(Peer.self);
    BackupMessage msg =
        new BackupMessage(ChordNode.PeerAddress, fileToRemove);
...
for (FileC f: filesToRemove) {
    storage.removeFile(f);
```

Listing 8: Backing up before deleting - MessageHandler.java

After the backup of these files is completed the Peer then removes the files in the list from its storage, which leads to the completion of this protocol.

## 3 Concurrency design

Our project can handle concurrent service requests. This is done by using the ExecutorService with ThreadPools. ExecutorService is a JDK API that

simplifies running tasks in an asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and an API for assigning tasks to it.

```
public static ExecutorService exec =
    Executors.newFixedThreadPool(256);
```

Listing 9: ExecutorService in ChordNode.java

Each *Message* is sent through a new thread, which is done through the execution of the *ExecutorService*.

```
exec.execute(new SendMessageThread(msg, null,
    ChordNode.getSuccessor().getSocketAddress()));
```

Listing 10: Sending Backup Message - Peer.java

# 4 JSSE

In order to guarantee secure Internet communications our project used Java Secure Socket Extension (JSSE).

The *SSLSocket* class is a subclass of the standard Java java.net.Socket class. It supports all of the standard socket methods and adds methods specific to the secure sockets.

The creation of the sockets is handled by *SSLSocketFactory*. This creates secure sockets to be used by the service. The *SSLServerSocketFactory* class is analogous to the *SSLSocketFactory* class, but is used specifically for creating server sockets (*ServerSocket.java*).

```
 SSLSocketFactory factory = (SSLSocketFactory)
    SSLSocketFactory.getDefault();
SSLSocket clientSocket = (SSLSocket)
    factory.createSocket(socketAddress.getAddress(),
        socketAddress.getPort());
```

Listing 11: Creating SSLSockets - Sender.java

# 5 Scalability

In our implementation each Peer extends the *ChordNode* class. When a Peer is initiated, it starts by joining the *Chord* (this.join();- Peer.java l.116).

```
public void join() throws IOException {
    if (isBootPeer) {
        this.create();
        return;
        }
    System.out.println("Joining Chord on: "
        + bootPeerAddress);
    ChordMessage msg = new
        JoinChord(PeerAddress);
```

```
    sendMessage(bootPeerAddress, msg);
}
```

Listing 12: Join - ChordNode.java

By having every single Peer automatically join the *Chord* upon starting, the system can handle a large number of Peers, depending on the size of each Peer's finger table. In this project, we used a finger table size of 8, allowing up to 256 Peers to simultaneously be part of the same *Chord*.

# 6 Fault-tolerance

In order to guarantee the stability and good performance of the system in case of errors, a fault-tolerance mechanism was implemented. If, at any given instant, a Peer is disconnected from the *Chord*, it informs both the *Predecessor* and *Successor*. These Peers will then update their registers in order to connect to one another, and maintain the *Chord's* integrity.

```
ChordMessage msg = new
    ChordDisconnect(PeerAddress,
        self.getChordID(),
        predecessorID,
        predecessorSocketAddress,
        successorID,
        successorSocketAddress);
    Sender senderSuccessor = new
        Sender(successor.getSocketAddress(), msg);
    senderSuccessor.sendMessage();
```

Listing 13: Sending message to successor - ChordNoce.class

Before leaving, the Peer reclaims all of its space, by initiating a new *Backup Protocol* for each file. The replication degree for each Protocol is 1, since we want to preserve the file's original replication degree. Once the Protocol is completed, the files are deleted from its storage. By doing this, even if a Peer leaves the system, the *Chord's File System* integrity is maintained.

```
public static void leave() {
...
 if (0 >= spaceUsed) {
...
fileToRemove.setRepDegree(1);
    fileToRemove.clearPeers();
    fileToRemove.setParentPeer(Peer.self);
    BackupMessage msg = new
    BackupMessage(ChordNode.PeerAddress, fileToRemove);

    exec.execute(new SendMessageThread(msg, null,
        ChordNode.getSuccessor().getSocketAddress()));
...
 for (FileC f: filesToRemove) {
        storage.removeFile(f);
    }
```

```
    Peer.getStorage().setCapacity(0);
    Peer.getStorage().setSpaceAvailable(0);
}
ChordNode.leave();
```

Listing 14: Peer Backing Up before leaving - Peer.java

As previously said during the *Delete protocol* (which starts with the reception of a *DeleteMessage*), at the end of a delete order, the receiver Peer sends a *DeletedMessage* back to the Sender. This *DeletedMessage* informs the Sender of the correct deletion of the file from the system. In case this message isn't received, the system informs the user of an error, which is displayed everytime the *State* protocol is run.

```
List<FileC> aux = new ArrayList<>();
for (FileC file: storage.getBackedFiles()){
    if(file.getRepDegree() == 0) aux.add(file);
...
if(aux.isEmpty()){
...
    System.out.println("All requested file deletions were
    successful!");
...
else{
...
    System.out.println("Unsuccessful file deletions request:\n" );
...
```

Listing 15: Displaying unsuccessfully deletion in state - Peer.java