Faculdade de Engenharia da Universidade do Porto
Mestrado de Engenharia Informática e Computação

Universidade do Porto

Faculdade de Engenharia

FEUP

Sistemas Distribuídos - Second Assignment

Distributed Backup Service for the Internet

Turma 5 - Grupo 22

up201704748	João Alberto Preto Rodrigues Praça
up201705227	Lucas Tomás Martins Ribeiro
up201604686	Ricardo França Domingues Cardoso
up201704684	Silvia Jorge Moreira da Rocha

2019/2020

Overview

In this project, as the name indicates, we developed a distributed backup service for the internet. The project revolves around the concept of using the free space in other computers to backup files of your own, despite this we must ensure that every individual machine that takes part in the process can completely take control of its storage, being able to delete other people's backup files without any restraints and in a way so that those backed up files won't be lost.

The first substantial decision we had to make was about whether we would use a chunk based system or if we'd work with entire files this time around, to which we opted to keep using chunks. Then we faced the decision of using a centralized server or having a fully distributed system, we decided to take the second and use Chord to achieve that result as it helped to improve the scalability of the system.

We also aimed to improve scalability by the use of thread pools in each of the nodes of our Chord system, being that these perform such tasks as listening for and sensing messages. We also tried to implement fault-tolerant features, although they ended up not working completely.

When it comes to the security of the communication between computers, we also had to decide whether we would implement JSSE, and we decided we would implement SSLEngine, but due to a small implementation error we did not manage to integrate SSLEngine into the project, so we went for SSLSockets instead.

Our project's backup service supports four different operations: the main one being the backup of files to other computers, the restoration of a backup file, the deletion of all chunks of a file from the backup service and finally the ability to reclaim disk space that is occupied by other computer's backed up files.

Protocols

Our backup service has four protocols: Backup, Restore, Reclaim and Delete.

Backup - This protocol consists in making a copy of all chunks of file, a certain number of times (replication degree), through the peers of the system. After running the command:

```
java TestApp [address] [port] BACKUP [file] [replication degree]
```

The specified peer receives a BACKUP message indicating to backup that file and the desired replication degree. Then, the peer splits the file in chunks and sends them to N peers in its finger table (PUTCHUNK message), to which they respond with a STORED message if they successfully saved the chunk they were sent. If the number of distinct peers in the finger table is less than the desired replication degree, then the peer sends the chunk to its successor. Then, if the chunk can be stored, the replication degree is decremented, and if it is not zero, it also sends the message to its successor. This process is repeated until the replication degree is zero, or until the predecessor of the initial peer is reached (all peers available received the message).

Restore - This protocol consists in recovering all the chunks of a file and joining them to create a copy of the original file. After running the command:

```
java TestApp [address] [port] RESTORE [file]
```

The specified peer receives a RESTORE PROTOCOL message indicating to restore that file. Then, it sends to the peers that have stored the chunks of the file a GETCHUNK message, and they answer with a CHUNK message containing a chunk. After that, the initial peer joins all the chunks it has received, and making an exact copy of the original file.

Delete - This protocol consists in deleting all the chunks of a file that were previously stored in the backup system. After running the command:

```
java TestApp [address] [port] DELETE [file]
```

The specified peer receives a DELETE_PROTOCOL message indicating to delete that file. Then, it sends to the peers that have stored the chunks of the file a DELETE message, and then they delete the specified chunk.

Reclaim - This protocol consists in changing the space that the system is allowed to use on a specific peer. After running the command:

```
java TestApp [address] [port] RECLAIM [max_space]
```

The specified peer receives a RECLAIM_PROTOCOL message indicating to set the allowed space to the new value. If the value is less than the space occupied by the files of the

backup service, then the peer sends the necessary chunks to other peers through a BACKUP_CHUNK message and then deletes them.

Concurrency Design

To ensure the concurrency of our design we used thread pools to support concurrent service requests. A thread pool is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. We decided to implement concurrency because, in our perspective, a distributed system that will have multiple messages behind sent across the structure has to be able to handle them asynchronously to improve performance.

We also understand the importance of using *Non Blocking IO*, our original design was meant to support them but, due to time related issues, we were not able to implement it.

```
this.executor.execute((Runnable) new SendMessage(message, finger.getAddress(), finger.getPort(), this));

Line 228 in ChordNode.java
```

In the picture below you can see an example of the creation of a new *Thread* using a *ThreadPoolExecutor*. We also use a *ScheduledExecutorService* in order to run a stabilizer periodically that assures the consistency of all peers.

JSSE

To assure the security of the communication between peers, we decided to use JSSE. As mentioned before, we initially decided to implement SSLEngine. This communication was fully working on Windows operative system but not in Linux based ones. For this reason, and because we weren't able to find a solution either in our research or through moodle, we decided to switch to SSLSockets.

```
System.setProperty("javax.net.ssl.keyStore", "server.keys");
System.setProperty("javax.net.ssl.keyStorePassword", "123456");
System.setProperty("javax.net.ssl.trustStore", "truststore");
System.setProperty("javax.net.ssl.trustStorePassword", "123456");

Line 48 in ChordNode.java
```

```
SSLSocket socket = (SSLSocket) this.serverSocket.accept();
```

```
private SSLServerSocket serverSocket;
this.serverSocket = (SSLServerSocket) ssf.createServerSocket(chordNode.getKey().getPort(), 0, address);
```

Given that we require user authentication by using a keystore and trustore, we assure that no one outside the distributed system can access the communication channels and intercept the messages.

Scalability

For this assignment we decided to implement a distributed system. We, as a group, considered that a centralized solution was not scalable enough for the design we wanted to achieve. For this reason, we decided to follow the suggestion given and implement Chord.

Chord is a scalable peer to peer lookup protocol for internet applications. In this protocol, each of its nodes holds a successor, predecessor and finger table. This three registers are what makes chord an optimized lookup system. Each time we, for example, want to know where a specific node is we just look to our finger table and try to identify in which slot it can be found and ask that peer for the information making the search logarithmic.

There is also a stabilizer that assures the consistency of the distributed system. This stabilizer is a thread that from time to time (in our case, every 10 seconds) runs all the necessary functions.

This functions includes methods to verify if our successor is still active and if we are still its predecessor (that is, if no other node has joined the ring and taken a place between them). In case any of this cases turns out to be true, the system will run a function to correct the finger table and update the successor.

```
public class StabilizeChord implements Runnable {

    private ChordNode chordNode;

    public StabilizeChord(ChordNode node) {
        this.chordNode = node;
    }

    @Override
    public void run() {
        chordNode.fixFingers();
    }
}
```

Fault Tolerance

To add fault-tolerance to our system, we decided that when a peer is shut down (Ctrl-C command), it should send to other available peers the chunks it has stored before deleting them. This way, we would try to avoid to decrement the replication degree, maximizing the probability of a chunk being found when trying to be restored. However, there were some issues during the implementation that we were not able to fix in time, so when terminating a Peer process, some exceptions are thrown.