

2º Pratical Assignment - Report

Report – MIEIC

Course: Sistemas Distribuidos

Authors:

* Ana Rita Fonseca Santos –  [up201605240@fe.up.pt](mailto:up201707291@fe.up.pt)
* André Filipe da Silva Moutinho – [up201704889@fe.up.pt](mailto:up201704889@fe.up.pt)
* Eduardo João Santana Macedo – [up201703658@fe.up.pt](mailto:%75p%32%301%37%303%36%358@%66%65%2eu%70%2e%70t)
* Sérgio Bruno Rodrigues Dias – [up201704889@fe.up.pt](mailto:up201704889@fe.up.pt)

**Maio 2020**

**Academic Year 2019 / 2020**

**Índice**

[**1.** **Introduction** 3](#_Toc40824013)

[**2.** **Overview** 3](#_Toc40824014)

[**3.** **Protocols** 3](#_Toc40824015)

[**3.1.** **Backup** 4](#_Toc40824016)

[**3.2.** **Restore** 4](#_Toc40824017)

[**3.3.** **Delete** 4](#_Toc40824018)

[**4.** **Concurrency Implementation** 4](#_Toc40824019)

[**5.** **JSSE** 4](#_Toc40824020)

[**6.** **Scalability** 4](#_Toc40824021)

[**7.** **Fault-tolerance** 4](#_Toc40824022)

# **Introduction**

This report serves as a complement for the second practical assignment, having the purpose of specifying the design and implementation of the backup service. It also describes the extra features that were implemented to raise the project’s grade’s ceiling.

# **Overview**

The implemented solution allows for the backup, restore and deletion of files. There is also a state operation, which shows the client the storage status and the files that a give peer has initiated backup for.

JSSE was used for secure communication. Although the project members planned and attempted to use the more complex interface **SSLEngine**, the difficulties encountered in the process and shortage of time led to the decision of changing the communication mechanism halfway through the project development. Therefore, the final solution uses **SSLSocket** instead**.**

As for the scalability and fault-tolerance, those are ensured at the design level. The implemented design is decentralized and conforms with the Chord protocol/algorithm, which was incorporated to satisfy both requirements.

.

# **Protocols**

# **Backup**

The protocol is initialized with the following command:

$ java src.service.TestApp <*peer access point*> **BACKUP** <*filename*> <*replication* *degree*>

The file to backup should be placed in the folder ***build/files\_to\_backup.***

1. Initialy, the class TestApp connects to the right peer via RMI, and then executes the following function *backup(file\_path, replication\_degree*.
2. A new thread is initialized in the correct ChordNode, which creates the key file, stores the key, the file path and its desired replication degree.
3. Secondly, a FIND\_BACKUP\_NODE message is created with the necessary information, and the thread calls find\_successor\_addr(key\_file, msg). This last function checks if the current node successor is the correct node to backup the file and from here two things can happen:

* If the previous is true in the first iteration, it will return the successor’s IP and from there, since we still are on the node that started the request, we send directly the file to the correct node with a BACKUP\_FILE message. If it is false, the node will forward the query (with the FIND\_BACKUP\_NODE message) using full advantage of the chord implementation and each node’s finger table.
* On the following iterations when a node receives a FIND\_BACKUP\_NODE message, it once again checks if its successor is the correct node. If this is the case, this time it will send a message to the peer that first originated the query with the IP address of the node to back up the file. Once the peer receives this message it will then send the file to the node with the IP address it received.

4. Once a Chord Node receives a BACKUP\_FILE message containing the file it will store the file in the correct place and update its *files\_backed\_up* hashmap so that the information is up to date. Secondly, it will handle the replicatin degree:

- When the Replication Degree received in the message is higher than one, the node will subtract this value by one and ask its successor to backup the file. This will go on until the value of the replication degree is one, meaning that we have achieved the replication degree requested by the user. If in this cycle a node asks a successor to backup a file and it already has it, it means that we went full circle and thus there aren’t enough nodes on the chord circle to achieved the desired replication degree.

# **Restore**

The protocol to restore a file is initialized with the following command:

*$ java src.service.TestApp <peer access point>* ***RESTORE*** *<filename>*

1. Once the TestApp is connected to the specific peer via RMI, the restore function is called and executed in this ChordNode.
2. In the ChordNode a new *Restore* thread is created, with the *node* who invoked the restore protocol and the *file\_name* as arguments.
3. This thread is responsable for making three important tasks, depending of the constructor:
   1. **SEND\_RESTORE\_MSG:** The first task, handles the process of finding the key for the file. With this key, it stores the path of the file in the node’s *files\_restored* HashMap.

Following that, it creates a message of type RESTORE\_FILE and tries to find in the chord circle, the node that has that specific key file.

To reach the node as quickly as possible, the first approach is to find the node that is suposed to have the file (the successor of key file) always using the search algorithm O(log N)). However, if along the way it finds a node with the key file we are looking for (possible due to replication degree) the query stops here.

The search starts in the node itself, and the message created is to communicate between the nodes within the circle when needed. When the message is received, they continue the search.

When a node returns true when asked if he has the file, a new *Restore* thread is created with the information of the node, the key file and the message responsible for this search. The second task EXECUTE\_RESTORE is initiated.

* 1. **EXECUTE\_RESTORE:** If a node creates this type of *Restore* thread, that means he has a file that he needs to send to the node who requested restore (the RESTORE\_FILE message sent in the constructor contains in the header the IP address of the node who started the restore request in the first place).

The file is converted to bytes and stored in the body of a new message of type RETRIEVE\_FILE and sent to the specific peer who requested the restore.

Once this message reaches the destination node, a new Restore thread is created with the node and the message received as arguments, performing the third and the last task, SAVE\_FILE.

* 1. **SAVE\_FILE:** When a node starts this thread, the message sent in the constructor is parsed, and the file is created in the *peer\_key/restore* directory, with the name corresponded to the key file in the files\_restored HashMap. Once the file is created, the key is removed of the files\_restored HashMap.

# **Delete**

The protocol to delete a file is initialized with the following command:

*$ java src.service.TestApp <peer access point>* ***DELETE*** *<filename>*

1. Once the TestApp connects to the specific peer via RMI, the delete function is called and implemented in the correct ChordNode.
2. Here a new *Delete* thread is created, with the *node* that invoked the restore protocol and the *file\_name* as arguments.
3. This thread is responsible for making two important tasks, depending on the constructor:
   1. **SEND\_DELETE\_MSG:** Handles the process of finding the key for the file and checks if the node is responsible for that key file, in other words if it is the successor of the key (the first node that tried to backed up the file).

If the query returns true it tries to delete the file automatically, calling *delete\_file()*.

Otherwise if the node isn’t the successor key, it creates a message of type FIND\_DELETE\_FILE\_NODE, and starts a search for that specific node in the circle using the Chord’s optimized search algorithm to jump between nodes and the message referenced before to communicate between them. When the message is received, the node verifies if its responsible for that key as described above, if not, they continue the search. The search ends when the message reaches the successor of the key file. When that happens, a new *Delete* thread is created with the information of the node and the key file as arguments. The second task EXECUTE\_DELETE is initiated.

Since the file no longer exists, in the end of this task, the *Restore* thread removes the key file from the node’s *files\_list* HashMap, responsible for storing information about the files the node requested to backup.

* 1. **EXECUTE\_DELETE:** This thread executes the function *delete\_file()*, and as the name indicates, the function is responsible for deleting the file.

Since the backup was done basically going from node to node sequentially until the desired replication degree is fulfilled, if the file doesn’t exist in this node (and not because space was reclaimed), this means the file is gone from the circle. If this is verified the delete protocol ends, if not a new message is created of type DELETE\_FILE and sent to the successor’s node.

When a node receives this message a new *Delete* thread is created to do the same task again.

# **Concurrency Implementation**

# **JSSE**

As mentioned earlier, this project does make use of JSSE for a safer communication. The security is ensured through the usage of Secure Sockets by the classes **SSLServerSocket**, **SSLServerSocketFactory**, **SSLSocket** and **SSLSocketFactory**, all from the javax.net.ssl package. Usages of this implementation can be seen in the file **Server.java**, which initializes the server and has loop which runs in the background waiting for new connections. Also, the file **MessageSender.java** has the functions used to connect and send messages to a peer.

# **Scalability**

Scalability of the application was ensured by implementing the Chord protocol/algorithm. This protocol revolves around a distributed hash table (DHT) to help in the resolution of unstructured names in the network. A DHT stores pairs (key, value) by assigning keys to different peers/computers in the network. A node only stores the values of the keys it is responsible for. Both values and peers are assigned an *m* bits identifier using consistent hashing, where 2m = maximum number of peers in the network.

Through the usage of this protocol, each peer only needs to keep track of around **log2 m**other peers, being **m** the amount of peers currently in the network. These peers’ information is stored in what’s called a **finger table**. The finger table helps in a way that querying for a key that is stored in the network is fast, since the number of nodes that must be contacted to find a successor in an *N*-node network is O(log N).

As mentioned earlier, implementing this protocol ensures scalability, since adding nodes to the network will have a low impact on the memory usage and the number of nodes required to query a key from the network.

Reference to important functions that implement chord (all in ChordNode.java):

* **join(InetSocketAddress)** : line;
* **notify\_successor()** : line;
* **notified(InetSocketAddress)** : line;
* **notified(InetSocketAddress)** : line;
* **find\_successor\_addr(long, Message)** : line

# **Fault-tolerance**

The design chosen was decentralized, which means that fault-tolerance was ensured by Chord’s fault-tolerance features. There are three threads running in the background that ensure the stability of the network:

* **FixFingersThread**: Every three seconds checks whether the peers in the finger table are all alive and updates the finger table in case of departure and join of nodes;
* **PredecessorThread**: Pings the predecessor and in case it doesn’t reply, sets it as null so that it can be updated later through the usage of a notify;
* **StabilizeThread**: Asks the successor for its predecessor and decides whether that predecessor should be the new successor, then notifies its successor, so that it can update its predecessor.

These threads are respectively implemented in the files: **FixFingersThread.java**, **PredecessorThread.java**, **StabilizeThread.java**, and only start running when the peer joins the network.