Faculdade de Engenharia da Universidade do Porto
Master in Informatics and Computing Engineering
Distributed Systems

Distributed Backup Service for the Internet

2nd Project

Bernardo Ramalho - up201704334
Catarina Fernandes - up201806610
Flávia Carvalhido - up201806857
João Rosário - up201806334

03MIEIC03 - Group 21

Wednesday, 2nd of June, 2021

Abstract

This project was developed for the Distributed Systems Course Unit using VSCode as a text editor tool and the machine's terminal to run the program, with the objective of developing a distributed system using Chord and JSSE. The problem simulates a distributed backup system, composed by peers that each represent a node in Chord. The developed program supports operations such as backup, restore, delete and reclaim. The problem was successfully solved and the solver's implementation and results are described throughout this report.

Contents

1. Overview

This project supports the same main operations as those implemented in project one (backup, delete, restore and reclaim).

In order to achieve secure communication, we chose to use the *javax.net.ssl.SSLSocket* and the *javax.net.ssl.SSLServerSocket* classes. Although our goal was to use the *javax.net.ssl.SSLEngine* abstract class, the lack of time to solve the implementation problems that emerged while developing didn't allow us to fully implement it.

To ensure concurrency, our design uses thread-pools, such as *java.util.concurrent.ScheduledThreadPoolExecutor*, and thread-safe data structures, such as *java.util.concurrent.ConcurrentHashMap<K,V>* and *java.util.concurrent.ConcurrentSkipListMap<K,V>*. We also used the Chord protocol to allow our decentralized system to be scalable and fault-tolerant.

In summary, in order to raise the ceiling of our grade, we have implemented:

- Thread-based concurrency;
- Communication using *javax.net.ssl.SSLSocket*/*javax.net.ssl.SSLServerSocket*;
- Improved scalability, at the design level, using the Chord Protocol and, at the implementation level, using *java.util.concurrent.ScheduledThreadPoolExecutor*;
- Improved Fault tolerance using Chord's fault-tolerance features which let the application avoid single-points of failure;

2. Protocols

2.1 RMI

We used *RMI* in order to establish communication from a client to a peer. That peer becomes the initiator peer of a certain protocol. Then, the initiator peer handles all communication and operations needed to complete the protocol.
The remaining communication is done using the *javax.net.ssl.SSLSocket* and the *javax.net.ssl.SSL ServerSocket* classes to provide secure communication. The details of the transport layer will be further described in the JSSE - chapter 4.

The remote interface has 5 methods that are implements by the peer. All of these implementations can be found in the *ChordPeer.java* file from line 141 to line 488, in the src directory.

- **Backup** method, which starts the backup protocol. For this, the client needs to send, as arguments, the file path and the desired replication degree;
- **Restore** method, which starts the restore protocol. In this one, the client only needs to send, as method arguments, the file path;
- **Delete** method, which starts the delete protocol. Just as the restore method, the client only needs to send, as method arguments, the file path of the file to be deleted;
- **Reclaim** method, which starts the space reclaim protocol. In this one, the client only needs to send, as method arguments, the space that the peer can use.
- **State** method, which starts the state protocol. In this one, the client does not need to specify any arguments.

2.2 Implemented Messages

2.2.1 FINDSUCCESSOR

`FINDSUCCESSOR <peerID> <CRLF>`

This message is sent by a peer who wants to know who the successor of *peerID* is. The peer who has received the message calls the *ChordLayer.findSuccesor(peerID)* method, which tries to find the successor of the *peerID*, possibly sending more *FINDSUCCESSOR* messages. The peer who received this message then responds with a *SUCCESSOR* message.

2.2.2 SUCCESSOR

`SUCCESSOR <successorID> <successorAddress> <successorPortNumber> <CRLF>`

This message is sent by a peer who has found the successor of *peerID* is. The peer who has received the message now knows who the successor of *peerID* is.

2.2.3 GETPREDECESSOR

`GETPREDECESSOR <CRLF>`

This message is sent by a peer who needs to know who is the predecessor of the node to which he has sent this message. The node who received the message then responds with a *PREDECESSOR* message.

2.2.4 PREDECESSOR

`PREDECESSOR NULL <CRLF>`

or

`PREDECESSOR <predecessorID> <predecessorAddress> <predecessorPortNumber> <CRLF>`

The *PREDECESSOR NULL* message is sent when the node who received *GETPREDECESSOR* doesn't have a predecessor. The *PREDECESSOR* message is sent so that the peer who receives it knows who the predecessor of the node who sent the message is.

2.2.5 NOTIFY

`NOTIFY <predecessorID> <predecessorAddress> <predecessorPortNumber> <CRLF>`

This message is sent by a peer who believes to be the predecessor of the peer who should receive this message. The peer who receives this message needs to check if the *ChordNode*, obtained with the information contained in the message, is indeed its predecessor.

2.2.6 CHECKCONNECTION

```
CHECKCONNECTION <CRLF>
```

This message is sent by a peer who wants to know if the peer that receives this message is alive. The peer who receives this message should respond with an *ALIVE* message.

2.2.7 ALIVE

```
ALIVE <CRLF>
```

This message a *CHECKCONNECTION* answer, sent to a peer who wants to know if the sender peer is alive.

2.2.8 INITIATOR

```
INITIATOR <file-path> <initiatorID> <initiatorAddress> <initiatorPortNumber> <CRLF>
```

This message is sent by the Initiator peer of a Backup protocol. This message is sent so that the peer who is gonna save the backed-up file knows which peer to message in case it needs to delete said file. This information is saved in the *FileData* of the file and is necessary for the space reclaim protocol.

2.2.9 SAVEFILE

```
SAVEFILE <file-id> <file-nr-chunks> <file-replication-degree> <file-size>
                                <file-path> <CRLF>
```

This message is sent by the Initiator peer of a Backup protocol. This message is sent so that the initiator peer knows who will save the requested backup file. Peers who receive this message will check if they can save this file: if they have enough free space or if they don't already have a file with the same path. If these conditions are both verified, they will respond with a *SAVED NULL* message. Otherwise, they will propagate the *SAVEFILE* message received to other nodes until they receive a *SAVE* message, containing the information about the peer who can save the file. If no peer can save the file, then all peers will respond with *NOTSAVED* to this message. Peers who already sent a *SAVEFILE* message have a `Boolean` attribute, which tells them if they have already sent the *SAVEFILE*, so that when they receive another *SAVEFILE* message, they automatically send a *NOTSAVED* message. For a better understanding we advise to see the code from the *Message.java* file, in the src directory, from line 123 to line 193.

2.2.10 NOTSAVE

`NOTSAVE <CRLF>`

This message is sent in response to a *SAVEFILE* message. This is sent when a peer can't save a file and also couldn't find a peer who could.

2.2.11 SAVE

`SAVED NULL <CRLF>`
or
`SAVED <nodeID> <nodeAddress> <nodePortNumber> <CRLF>`

This message is sent in response to a *SAVEFILE* message. The *SAVED NULL* message is sent when the peer who received the *SAVEFILE* message can save the file. The other SAVED message is sent when the peer, who previously received the *SAVEFILE* message, couldn't save the file, but found other peer who could save it. This message has all the information about the found node which can save the file.

2.2.12 PUTCHUNK

`PUTCHUNK <file-id> <chunk_number> <CRLF> <chunk-data> <CRLF>`

This message is sent by the Initiator peer of a Backup protocol. This message is sent to the peer who could save the file with *file-id*. Upon receiving this message, the peer creates a *Chunk* object with the received data and saves it into the *fileData* object, referring to the file that is currently backing up.

2.2.13 SAVECOMPLETED

`SAVECOMPLETED <file-path> <CRLF>`

This message is sent by the Initiator peer of a Backup protocol. The peer who receives it, knows that there are no more chunks left to receive and starts writing the file data into its physical directory.

2.2.14 GETCHUNK

`GETCHUNK <file-path> <chunk-number> <CRLF>`

This message is sent by the Initiator peer of a Restore protocol. The peer who received it needs to respond with a *RESTORECHUNK* message containing data from the chunk, with *chunk-number*, from the file, with *file-path*.

2.2.15 RESTORECHUNK

`RESTORECHUNK <file-id> <chunk_number> <CRLF> <chunk-data> <CRLF>`

This message is sent by a peer who previously received a *GETCHUNK* message. This message contains the data of the , with *chunk-number* from the file, with *file-id*. The peer who receives it adds the chunk to the *restoreFile* array.

2.2.16 DELETE

`DELETE <file-path> <CRLF>`

This message is sent by the Initiator peer of a Delete protocol. The peer who receives it needs to delete the file with *file-path*. It then answers with a *DELETED* message.

2.2.17 DELETED

`DELETED <CRLF>`

This message is sent by a peer to the Initiator peer of a delete protocol to confirm the deletion of a file.

2.2.18 REMOVED

`REMOVED <file-id> <CRLF>`

This message is sent by the Initiator peer of a Space Reclaim protocol to the Initiator peer of the Backup protocol of the removed file with *file-id*. The peer who receives this message restarts a Backup protocol for the file with *file-id*.

2.3 Backup

The *BACKUP* protocol can be initiated with the following command:

```
java TestApp <rmi-object-name> BACKUP <file-path> <replication-degree>
```

The *BACKUP* protocol has a series of steps that are programmed to happen sequentially, for each file. In each backup protocol there as many files as the necessary replication degree. The steps go as follows:

1. Initiator creates a new file and adds it to the *filesBackedUp java.util.concurrent.ConcurrentHashMap <String,FileData>* in the *PeerFolder* folder. The creation of the file also includes splitting the file into chunks of, at max, 10000 bytes in size;
2. Initiator checks which peer can save the file by sending a message that travels each node until it fins a suitable candidate that can store the file;
3. After finding a suitable peer, the initiator peer will send a *SAVEFILE* message so the peer who will save the file knows which peer to inform when it needs to delete the backed up file.
4. Sends *PUTCHUNK* message for every chunk of the file to the peer who will save the file;
5. Sends a *SAVECOMPLETED* message so the peer knows that there are no more chunks left to store.

The process behind finding a suitable peer is described in subsection 2.2.9 - *SAVEFILE* message. The main code of this protocol is located in the *ChordPeer.java* file, in the src directory, starting at line 163 and finishing at line 281 and in the *Message.java* file, also in the src directory, starting at line 123 and finishing at line 228.

2.4 Delete

The *DELETE* protocol can be initiated with the following command:

```
java TestApp DELETE <file-path>
```

Firstly, the Initiator Peer checks if the file is present in its local storage, verifying *filesBackedUp ConcurrentHashMap* in the *PeerFolder* folder. If saved, the Initiator Peer sends a *DELETE* message to all the nodes that contain a backup of the file.

DELETE message, as in subsection 2.2.16:

```
DELETE <file path> <CRLF>
```

After that, the initiator deletes all the references of that same file, after which it will delete the file location from the *fileLocation* map and the file reference from the *filesBackedUp* map.

When a Peer receives a *DELETE* message, upon parsing it, it will remove the file from the local storage folder and from the *storedFiles ConcurrentHashMap*.

The main code of this protocol is located in the *ChordPeer.java* file, in the src directory, starting at line 347 and finishing at line 387 and in the *PeerFolder.java* file, also in the src directory, starting at line 296 and finishing at line 336.

2.5 Restore

The *RESTORE* protocol can be initiated with the following command:

```
java TestApp <rmi-object-name> RESTORE <file-path>
```

The first step is to check if the current Peer has the file reference in its *backedUpFiles* map. If it doesn't, the operation is cancelled, as there is no way to obtain the file.

After the above condition is verified, the peer gets all the *ChordNodes* who have stored that file. This information is stored in the *ConcurrentHasMap fileLocation*, in the *PeerFolder* class. Consequently, it sends *GETCHUNK* messages for each chunk it needs to only one of those nodes. The peer needs to get all the *ChordNodes*, because if one of them fails while the initiator peer is trying to restore a chunk, the initiator peer resumes the operation by asking the following *ChordNode* for all the chunks, starting by the chunk that couldn't be restored by the previous *ChordNode*. This is shown in the *catch* instruction of the *restore* method in the *ChordPeer.java*, lines 315 to 331. That *catch* only occurs if the initiator peer can't send the GETCHUNK message to the *ChordNode*, assuming that it has shutdown).

When the peer has received all the chunks it needs, it will restore the file in its physical folder. The main code of this protocol is located in the *ChordPeer.java* file, in the src directory, starting at line 283 and finishing at line 345 and in the *PeerFolder.java* file, also in the src directory, starting at line 278 and finishing at line 294.

2.6 Reclaim

The *RECLAIM* protocol can be initiated with the following command:

```
java TestApp <rmi-object-name> RECLAIM <max-storage-size
```

If a client wishes to change the space a peer has available to store files, it can do so with this protocol. After receiving the command above, the Peer checks if the space that it is currently using is bigger then the new storage size that it can use. If so, the initiator Peer will iterate over the stored files and remove one by one until it frees up enough space. Every time the Peer removes one file, it has to send a *REMOVED* message to the initiator Peer of the backup relative to the file it has just deleted.

REMOVED message, as in subsection 2.2.18:

```
REMOVED <file-path> <CRLF>
```

If a certain Peer receives this message, it has to check whether or not the desired replication degree is still ensured. In case it is not, it initiates a new *BACKUP* protocol for that same file.

The main code of this protocol is located in the *ChordPeer.java* file, in the src directory, starting at line 389 and finishing at line 440 and in the *Message.java* file, also in the src directory, starting at line 230 and finishing at line 252.

2.7 State

The *STATE* protocol can be initiated with the following command:

```
java TestApp <rmi-object-name> State
```

The *STATE* protocol is used solely to compile information about the current state of the Peer and display it to the client. The information displayed is the following:

- For each file reference stored by the Peer:
  - File number
  - Pathname
  - File ID
  - Desired replication degree
  - Chunk information:
    - Chunk number
    - Chunk ID
    - Perceived replication degree
- For every chunk stored by the Peer:
  - File key that the chunk belongs to
  - Chunk ID
  - Chunk size
  - Desired replication degree
  - Perceived replication degree
- Maximum Peer storage capacity
- Used Peer storage capacity
- Peer finger table

The main code of this protocol is located in the *ChordPeer.java* file, in the src directory, starting at line 443 and finishing at line 488.

3. Concurrency design

The concurrency design in this project was achieved by using Threads, Thread-pools and Runnables. Two classes were created to help us achieve this:

Listener

- Responsible for listening to the *SSLServerSocket* and creating a thread that runs *ReceiveRequestTask*,
- This class implements the *Runnable* interface and is initialized in the *ChordPeer* class, executed by a *threadPoolExecutor*, which means there will be a certain amount of worker threads - currently 100 - allocated to receive messages from the socket.
- The code for this Class can be seen in the Listener.java file in the src directory.

ReceiveRequestTask

- This class is responsible for reading the request received from the *SSLServerSocket*. It creates a *Message* Class Object which will parse the request and solve it, returning the respective response needed to be sent. It then sends the message through the socket *ReceiveRequestTask*,
- This class also implements the *Runnable* interface.
- The code for this Class can be seen in the *ReceiveRequestTask.java* file in the src directory.

These two classes allow each peer to solve multiple requests at the same time, because after the *Listener* accepts a *Socket* from the *SSLServerSocket*, it creates a thread that runs the *ReceiveRequestTask*. This makes it so that the *Listener* is ready to accept other requests, while solving the previous one.

4. JSSE

The communication between the peers is all done using *JSSE*, so every protocol uses it. It was implemented because it is the most secure way of communicating and it also helps achieving better concurrency. Client authentication is required between every message and the default *cypher-suites* are also used. Unfortunately, the implementation of *SSLEngine* was not done due to lack of time. JSSE is mainly use in the *ReceiveRequestTask.java*, from lines 19 to 54, and in the *RequestSender.java*, from lines 38 to lines 82, class both in the src directory.

5. Scalability

To improve the scalability of the system, the Chord Protocol was implemented. This means that the peers don't know about every peer that exists in the system. To make the finger table, a *java.util.concurrent.ConcurrentSkipListMap<K,V>* is used. This table is responsible for storing information about 16 nodes, the successors of $peerID + 2^{k-1}$, where k goes from 1 to 16 (since our IDs are 16 bits). This way when the peer wants to find the successor of a key, it searches for the closest node in its finger table and asks the closest finger who is the successor of the key (see *ChordLayer.java* file, in the src directory, from lines 128 to 175).

This makes that each peer only knows about 16 peers, no matter how many peers exist in the *Chord*. This allows the system to have up to 2^{16} nodes, while not flooding peers with information about other peers. This also reduces the travel distance of each message, since it won't have to go from successor to successor.

The *java.util.concurrent.ScheduledThreadPoolExecutor* is also implemented, so that every task related to the maintenance of the *Chord* correctness is done in a separate thread (see *ChordPeer.java* file, in the src directory, from lines 119 to 126). Moreover, using this thread-pool to solve requests received by a peer, it can receive multiple requests and solve all of them at the same time. This is also possible due to the use of thread-safe data structures, such as *java.util.concurrent.ConcurrentHashMap<K,V>* (see *Listener.java* file, in the src directory, from lines 39 to 47).

6. Fault-tolerance

With fully implemented *Chord*, the system also makes use of its fault tolerance features. So when a node fails, a peer who noticed the failure, removes all entries that have the failed node from its finger table. It also updates its successor in case it was it who failed (see *ChordLayer.java* file, in the src directory, from lines 196 to 223). Each peer also runs a task every 5 seconds where it checks if its predecessor is still alive. If the predecessor is not alive, the peer sets its predecessor to *null* (see *CheckPredecessorTask.java* file, in the src directory, from line 15 to 56).

The is also an implemented additional feature to improve the system's fault-tolerance. All of the peers who have initiated a *BACKUP* protocol, run a task every 5 seconds to check if the nodes who have stored the files are alive. If they aren't, then the peer restarts the *BACKUP* protocol for that specific file, thus making sure that the replication degree is always respected (see *CheckBackupNode.java* file, in the src directory, from lines 10 to 36).

7. References

[1] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek,and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications.I *EEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.