Distributed Backup Service

João Diogo Martins Romão[1] Tiago Ferreia Alves[2], Nuno Filipe Ferreira de Sousa Resende[3] and João Miguel Gomes Gonçalves[4]

SDIS, 3MIEIC06, G26, Distributed Backup Service with Chord

1 Overview

The purpose of this second assignment is to complement the first one where we developed a backup service that allows for the **backup** of files divided in chunks and distributed in multiple peers, the **restoration** of a previously backed up file and the **deletion** of the previously stored chunks. The application also allows total control of the peer's storage usage, having the possibility of **reclaiming** the space it wants by deleting chunks it stores. Additionally to this, it can also verify the activity of the peer using the **State** protocol.

One of the major differences of this project, when compared to the first one, is the implementation of **TCP** and **Chord**, which is a distributed lookup protocol which allows an efficient location of the nodes that stores the desired data item, and even adapts efficiently as nodes join and leave the system, being able to keep the lookup time via **O(log N)** messages to other nodes.

In order to achieve a non-blocking **SSL** solution, the group decided to use **SSLEngine** for secure communication, allowing to encrypt and decry-pt messages sent and received from peers. **Scalability** was also a major concern in in our project, having used **thread pools** and **asynchronous reading and writing** to files. Each channel runs in a different thread, and so all the messages which are received at the same time in the same channel.

Finally, in order to address **fault tolerance**, each file can be backed up with a certain **replication degree**, each chunk being saved n times, since a file would be lost in case of a single peer failure otherwise.

TODO FILE STORAGE?

All of these considerations allow for the ceiling of our project to be of **20 grade points**.

2 Protocols

Since this project required the same protocols as the previous one developed in this curricular unit, we implemented the same five protocols present in the first project, the **Backup**, **Delete**, **Restore**, **Reclaim** and **State**. In this next section, it will be presented how we implemented these protocols, as will as the underlying transport protocols and the format of all the messages. The **RMI** interface will also be thoughtfully explained.

2.1 File system Protocols

– **Backup** The backup protocol allows the user to backup a given file in the distributed network. Firstly, the initiator peer verifies if the backup of the file has already been realized by the peer, by checking if the file is already present in the *"Hosting Metadata"*. This metadata has only simple file information such as the id, rep Degree and the number of chunks of the hosting file. Secondly, it properly adds the new backup information to the hosting metadata, and divides the file in chunks of **15Kb**, giving each one a Chunk file id, which takes into account the hash of the fileId, the chunk number and the respective replication degree of the file, so each copy of the chunk is saved in a different peer. After this, it is created a **Putchunk** message for each chunk, which is sent to the *Backup Channel* of the peer decided by Chord, based on the ChunkId.
The peer which received the **Putchunk** message can have different behaviours. If the *shouldSaveFile* function returns true, the peer will promptly save the chunk and calculate a new file id, with the same information, but with a replication degree reduced by one, saving each copy of the chunk in a different peer. On the other hand, if the function returns false, the peer will resend the message to its successor, until it can save the file. It was also necessary to create a *shouldResend* function, which verifies if the chord id of the chunk is not in between the current node id and the successor's id, which would mean it had already tried to backup the file in all the chord nodes. The shouldSaveFile, considers if the file has already been saved in that peer, if it has enough space to save the chunk and if it is not the same peer which initiated the protocol or the one which sent the previous message.
Since **TCP** is reliable, we have decided to eliminate the **STORED** messages, which were useless, in order to reduce the amount of messages involved in the backup of a file.

– **Delete**
Since we are using Chord to its full capability, not saving any information of which peers backed up our files, the delete protocol consists of updating the metadata, and calculating all the chunk Ids generated in the backup by doing two nested for-loops, one for each chunk number, and one for each replication degree, sending the **DELETE** message to the corresponding Chord nodes. After receiving the DELETE message in the ***Control Channel***, the receiver peer will then delete the file if it previously backed it up, or send the delete message to its successor, always verifying if this message enters an infinite loop, analysing the interval as explained in the Backup protocol.

– **Restore**
In the restore protocol, firstly it is verified if the peer has hosted a backup for the file, and if it did, it creates a chunk file id for each chunk, only considering in the id generation the original replication degree of the backup. It then sends a **GetChunk** message for each of the chunks, giving as arguments the

ip address and port of the peer's Restore channel, so the receiver can know where to send the saved chunk. This message is then sent to the Control Channel of the node calculated by Chord. When the other peer receives the **GetChunk** message, it will verify if it has the given chunk, sending back to the initiator peer all the information in a *Chunk* message. Otherwise, if the peer does not have the given chunk, it will then forward the **GetChunk** message to it's successor. If the message passed through all the peers, a new **Getchunk** message is created, with the same information as the previous one but with the replication degree decremented by one in the chunk Id generation. This process is repeated until the pretended chunk was found, or until the replication degree is 0.

- **Reclaim**
  In the beginning of the implementation of the **Reclaim** protocol, we initially tried to follow the same logic of the first project, in which the peer which initiated the reclaim would firstly delete the file, send a **REMOVED** message to a peer which had the same chunk backed up, and that last peer would initiate a Backup protocol, which would save the file in another peer. The problem with this implementation, is that the peer would send the REMOVED message to a peer with the saved chunk, and this peer would send the file to the same peer which initiated the reclaim, only to realize this peer did not have enough space and send it to its successor. Therefore, we implemented a much **simpler solution**, in which the peer, before **deleting the file, initiated a backup protocol in its successor**, which would be the final destination in the other implementation. After doing the backup it can then delete the file, being possible to prevent many messages and unnecessary look-ups.

- **State**
  The state protocol is very simple, not having to communicate with other peers and with the only objective of presenting the *"Metadata"*. This state is divided in two parts, the *"Hosting Metadata"*, which contains all the information of the files which were initiated by the peer, and the *"Stored Chunks Metadata"* which stores all the necessary information of the files that the peer is currently backing up. The Hosting metadata, has for each file it hosts the id, path, replication degree and size, and the stored metadata has information about the file id, the chunk id as well as the size and the replication degree (perceived by the peer) in which it saved each chunk.

2.2 Messages

In our implementation of the distributed system, all the peers communicate with each other by messages, using **JSSE** secure communication which uses **SSLEngine**. All the messages have a *getBytes()* method, which allows the function present in the ***"MessageSender"*** class to send the message to the

calculated chord node. The format of all the messages, excluding the ones used in chord which are explained in the chord section, are the following:

- **PUTCHUNK** - Message sent by the peer, when it wants to backup a given chunk with a given **replication degree**, it includes the necessary information to save the chunk, like the **File id** and the **chunk number**, as well as the **IpAddress** and the **Port**, with the only objective of verifying if the peer that is receiving the message is not the peer which initiated the backup protocol. The Ip address and port were also initially used to inform the peer which was backing up the file, of where to send the stored messages, but we decided that in this implementation of the project this type of message was not necessary, since the initiator peer does not save information of who saved it's files. The **content of the chunk** is also sent in this message, as a byte array, "protected by the *CRLF*". Additionally to these parameters, we had to add an extra one, *selfRcvd* in order to verify if it does not enter the infinite while loop when sending to its immediate successor, which if the message passed two times through the initiator peer, we would increment this variable, and if its value were 2, then the message would have already passed by the peer two times.
  Format: $PUTCHUNK < FileId >< ChunkNo >< IPAddress >< Port >< ReplicationDeg >< CRLF >< CRLF >< Body >$

- **GETCHUNK** - After backing up a file in the distributed system, the peer can send this type of message in order to **recover a certain chunk** of the given file. This message includes the **file id** and the **chunk number**, so the receiver can verify if it contains the file, as well as the **replication degree**, which is used in order to prevent, in the case, that the receiver peer does not contain the file and send to its successor, an infinite loop in the case of none of the peers have the file stored. The replication degree allows to recreate the chord file id. The **Ip address**and the **port** are the indication of where to send the chunk.
  Format: $GETCHUNK < FileID >< ChunkNo >< IPAddress >< Port >< REP\_DGR >< CRLF >< CRLF >$

- **CHUNK** - Is the response to the **GETCHUNK** message, includes the **content of the requested chunk**, as well as the **file id** and corresponding **chunk number**, in order for the receiver to know which chunk is being recovered.
  Format: $CHUNK < FileId >< ChunkNo >< CRLF >< CRLF >< Body >$

- **DELETE** - When a user wants to delete a file, it can send the **Delete** message to all the peers which have any of the file chunks stored. It contains the **file Id**, so the user knows which file to delete, and the **ChunkFileId**, which is the chunk file id used to create the chord hash. This last parameter is sent in order to prevent the infinite while loop, which could be caused if

the receiver did not have the given file and propagated to its successor and so on. Using this parameter it is possible to verify if this message had already been passed through all the nodes.
Format: $DELETE < FileID >< ChunkFileId >< CRLF >< CRLF >$

2.3 RMI

In a fashion similar to the first assignment, each peer implements a ***"RemoteObject"*** so it can use **RMI**, which contains the following methods:

- void **backup**(File file, int repDegree) - Initializes the backup procedure of the given file with an intended replication degree
- void **restore**(String path) - Initializes the restore procedure of the using the file path given
- void **delete**(String path) - Initializes the delete procedure of chunks in other peers using the file path
- String **state**() - Gets the information of a peer stored in the metadata like stored chunks, files that it has backed up and the remaining reclaim space
- void **reclaim**(double maxDiskSpace) - Initializes the reclaim procedure, so that the peer can reclaim the desired disk space, deleting some chunks that it is storing

3 Concurrency design

3.1 Implementation structure

Each peer starts up by registering creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name received as a command line parameter in a **Java RMI** registry. This allows the peer to communicate with the client, the developed **TestApp**, and execute the requested protocol.

The peer also creates four receiver threads, one for each of the channels described on the first project (**MC**, **MDR**, **MDB**) and one thread to receive chord messages, that handle received messages trough SSLEngine and allow the peer to receive and process the messages.

3.2 Concurrent data structures

Since we have used **RMI**, the multicaster thread is the thread created by the **RMI** run-time to handle any protocol operation invoked by the **TestApp**.

In order to keep consistent information of each peer we needed thread-safe data structures. The information relative to a peer is kept in a **Serializable** class called Metadata, that keeps information relative to protocol executions hosted by the peer and information relative to chunks the peer has saved. Most

of this information is kept in several **ConcurrentHashMap's** since it allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So **ConcurrentHashMap** allows concurrent threads to read the value without locking at all.

3.3 Use of synchronized blocks

In order to keep the chord node finger table up to date a **ScheduledThreadPoolExecutor** is periodcally checking for possible updates on the finger table. Futhermore the finger table also needs to be updated when a peer cannot establish connection with the successor, this peer might not be running anymore and the finger table entries where it is present must be deleted. To do this we make use of a thread safe synchronized block.

```
private void deleteDataFromFingerTable(List<ChordNodeData> fingerTable, ChordNodeData toRemove) {
    synchronized (this) {
        for (int i = 0; i < fingerTable.size(); i++) {
            if (toRemove.getId() == fingerTable.get(i).getId()) {
                fingerTable.set(i, this.data);
            }
        }
    }
}
```

Fig. 1. Auxiliary Variables

3.4 Chord Scheduled Executors

As it was mentioned a **ScheduledThreadPoolExecutor** is used to keep the chord node finger table up to date.

ScheduledThreadPoolExecutors are also used for all of the remaining chord operations that need to be done periodically, such as the stabilize and notify actions, keeping the information of a node that might become one peer's successor if the current one fails and periodic calls trying to connect to the predecessor and the successor to check if they are still running.

These actions are executed concurrently by making use of this structure.

3.5 Elimination of blocking calls

In some implemented protocols, it was necessary to wait between 0 and 400 ms before sending a message to the control channel, in order to prevent the flooding of the channel, or for example in the enhanced backup, to decide whether to store the chunk or not depending on the stored messages received during that time.

In the beginning, we used the *thread.sleep()* calls to generate those waiting times. However, this calls can lead to a large number of threads running at the

same time, consequently limiting the scalability of the service. In order to reach a better scalability, instead of the *thread.sleep()* we used the *ScheduledPoolExecutor.schedule()*, a non blocking behaviour that is run in another thread.

The last step was to remove all the blocking calls that would be accessed by multiple threads, such as in File reading and writing, being possible to reach an even higher scalability. This was possible by using the **AsynchronousFileChannel** Class, which allows concurrent access to files and spawns a thread in different ways:

- Callback functions are used to execute a task as soon as the file operation was finished
- The operation returned a Future object, that allowed to execute other operations before having to use the respective value, taking maximum advantage of the asynchronous capabilities.

4 Chord

In this section, we proceed to the explanation of the implementation of the Chord structure, how it's connected to the peers and how the messages are being sent.

Our main data structures that maintain the required information about the nodes in the chord are contained in the classes ***"ChordNode"*** and ***"ChordNodeData"***. All of these are contained in the ***"chord"***

The **ChordNodeData** class holds the following information:

- **id** - Unique ID of the node in the chord
- **addressPortList** - Data structure that contains the ports and addresses of each channel

We decided to keep the separation of the channels similarly to the first project, so that each one has its purpose. For example, the backup channel is only responsible for the messages that involve the backing process of a file chunk (in this case, the **PUTCHUNK** message) and nothing else. Furthermore, we added a channel dedicated to the messages related to the Chord, which handles messages like **GET_SUCCESSOR**, **NOTIFY**, etc.

The second important class, **ChordNode**, contains most of the important information, like the predecessor and successor of that peer, the finger table that holds information on the other nodes that this node knows and methods to accommodate all the required tasks.

It holds the following important information:

- **fingerTable** - List of **ChordNodeData** that contains the information of the peer's known successors
- **data** - The peer's own information
- **predecessor** - Peer's predecessor information
- **successor** - Peer's successor information

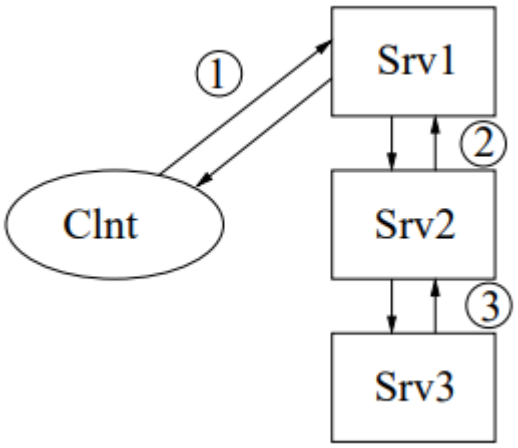
– safeSuccessor - The current successor's successor. Updated periodically and may be util when the successor disconnects.

Now we present the useful functions and their purpose of this class:

– **void create()** - Initializes the predecessor and successor information
– **void join(String chordAddress, int chordPort)** - Sends the appropriate message to get the successor information, so that node can join the chord by contacting the node with address chordAddress and chordPort
– **void stabilize()** - Runs periodically and checks that the successor and predecessor information is up to date. It asks for the predecessor of the successor to check for new nodes joined the Chord ring, telling the successor about itself
– **void fixFingers()** - Runs periodically and checks that a node's successors' information is correct and that those nodes are contactable
– **ChordNodeData findSuccessor(Integer id)** - Finds the node's successor for a given id, possibly needing to send an appropriate message
– **ChordNodeData closestPrecedingNode(int id)** - Gets the information on the highest predecessor of the id in the finger table.
– **int generateHash(String ipAddr, int port)** - Generates the **SHA-1** hash to create an identifier of a node in the Chord by using the given ipAddress and port, with values up to 2^m and for the files.
– **int generateHash(String name)** - Function overload, similar to the one described above with
– **void checkPredecessor()** - Contacts the predecessor to check if it is still up and running, setting it to null otherwise.
– **void checkSuccessor()** - Contacts the successor to check if it is still up and running, setting it to null otherwise.

The most essential operation of the Chord is the lookup of a successor of a given key, which is used, for example, to determine which peer should store a chunk.

In the execution of this procedure, first we check that the given key is between the peer's own id and its successor's id, and in case that's true, the peer's successor is found. Otherwise, the peer will check in its fingerTable using the **closestPrecedingNode()** function if any of them is the successor. If all of these fail, the message **GET_SUCCESSOR** is sent to find more information about the peers "further away" to the closest preceding node. If a node ever receives a message of this kind, it tries the execution described previously, forwarding the message further if the needed information is not found locally. As soon as this information is found, the response is sent to the node that directly asked for the information, making this a recursive approach as the following image illustrates:



– **JOIN** - Message containing the id of the peer that wishes to join. This message is sent to one node already in the chord it wishes to join. This message will be replied with the successor of the node that just joined serialized.

  Format: $JOIN < Id >< CRLF >< CRLF >$

– **NOTIFY** - Message containing the serialized data of the current peer. This message is sent in the *stabilize()* function to his successor, informing him that he is now his new predecessor. His successor will then update his predecessor. The response to this message is essentially discarded.

  Format: $NOTIFY < SerializedData >< CRLF >< CRLF >$

– **GET_PREDECESSOR** - Message sent to the peer's successor in the *stabilize()* function to update the predecessor of a peer. This message will be replied with the serialized predecessor of the peer that it is sent to.

  Format: $GET\_PREDECESSOR < CRLF >< CRLF >$

– **GET_SUCCESSOR** - This message is handled in the exact same way as the **JOIN** message, only existing for better readability purposes and improved comprehension.

  Format: $GET\_SUCCESSOR < Id >< CRLF >< CRLF >$

We will now proceed to describe some of these interactions in further detail:

When a peer connects to the chord, it shall send a **JOIN** message containing his id to one node of the chord and it shall reply with a message containing the successor of the peer that just joined serialized. This successor is calculated based on his id and the id of the peer that just joined. The id is calculated by encrypting the address and the port and casting it to an integer. Like this, when a new peer joins the chord, it joins with a successor initialized, however it still doesn't have a predecessor initialized yet. This is fixed by invoking the method **stabilize()** periodically. This method will, for every node of the chord, send a **NOTIFY** message containing himself serialized to its successor, informing that it is its new predecessor. Its successor will then update its predecessor.

Another detail that had to be treated when a peer joins, was the transfer of chunks that should be backed up on the peer that just joined. This whole process is based on the id of the chunks. This is calculated similarly to the way of the peer id, by using the file id, chunk number and replication degree and encrypting it using the same hash as before. This way we determine if a chunk should be backed up on a node that just joined, which happens if its id is smaller than the id of the node which is storing the chunk and higher than the chunk id. If this happens the peer with the wrongly stored chunks shall send a **PUTCHUNK** message with the body of the chunk and deletes it.

The function **fixFingers()**, which is called periodically, updates the information in every finger table of every peer. This finger table has m entries and each entry (from 0 to m-1) has the successor of $currentPeer.id + 2^{entry}$. This successor is discovered using the function **findSuccessor()**.

5 JSSE / SSL

The **JSSE** is used in this project to safely send messages between peers, being the standard way Java provides to implement **SSL/TLS** communication. In order to achieve a non-blocking SSL/TLS solution, **JSSE** provides the **SSLEngine** class, which is used when sending data through **TCP** connections which are established throughout all of the protocols. In order to use **SSL Engine**, it was necessary to decide the way the transport link was going to be implemented, as well as some parts of the protocol himself, which will be explained in the following section. The **SSLEngine** is created using the **SSLContext**.

In our project there are four main classes which implement the **SSL** connections:

- ***"SSL"*** - The **SSL** is an abstract class, with general **SSl** functions which can be used either by the sender or by the receiver peer. It implements functions such as the initialization of the **SSl** Context, which also creates the **TrustManager** Factory as will as the **KeyManager** factory for the given information present in the **SSLInformation** class. The handshake protocol is also implemented in this class, involves the exchange of configuration messages in order for the connection to be established. It also has a generic

read and write function, which the receiver and the sender will call in their own read and write functions, as well as the discconect function and other util ones like the enlargeBuffer.

- ***"SSLReceiver"*** - This class extends the SSL one to run as a server. In takes as arguments the Ipaddress and the port in wich it will listen to all the messages, as well as the Handler Channel, an argument of type *"Channel"* which will parse and handle all the read messages. After the instantiating, the thread will wait for new connections and handle the requests which arrive to it. When the thread is executed, it calls the method *start* which is a loop that will run as long as the server is active.

- ***"SSLSender"*** . This class is used whenever a peer wants to send a message to another, and consists of two phases: The connection, where it opens the socket channel to communicate wit the configured server and tries to complete the handshake protocol, and the write function, which is called if the connection was set successfully and sends the message to the connected server.

- ***"SSLInformation"*** - This is just a model class, in which we specifed the predefined location of all the important variables of the **SSL** connection as it can be seen in the picture below:

```
public class SSLInformation {
    public static final String protocol = "TLSv1.2";
    public static final String serverKeys = "../ssl/resources/server.keys";
    public static final String clientKeys = "../ssl/resources/client.keys";
    public static final String trustStore = "../ssl/resources/truststore";
    public static final String password = "123456";
}
```

Fig. 2. Auxiliary Variables

6 Scalability

Our project includes some features, already described previously in more detail, when it comes to scalability: the **Chord** structure, **thread-pools** and **java NIO**.

6.1 Chord Protocol

The Chord protocol contributes heavily to the scalability of any peer-to-peer service for Internet applications. Even though we may have added some extra interactions to ensure the backup service's perfect operation, the structure is, in its essence, the original concept. The fact that a given node only maintains

information on a few nodes near it on the Chord contributes for a large reduction on the memory spent, that would otherwise be needed if we decided to keep centralized server, where every node would keep information on all other nodes. By keeping information on about only $log(n)$ nodes, with the finger table update mechanism and the usage of messages on the Chord channel, we can ensure good performance and reliability, even with a large number of peers in the system.

6.2 Thread-pools

Thread-pools, already used in the first project, are used to send and receive message, to run the Chord protocols described previously, periodically, and to initialize protocols. Thread-pools are a good solution for juggling the concurrency in a system, while keeping track on the amount of threads used and, as such, contributing to the scalability.

6.3 Java NIO

Java NIO is also useful since it allows multiple messages to be exchanged without much overlay, contributing therefore to the scalability.

7 Fault Tolerance

In this last section we describe the features that we implemented to avoid some possible failures:

One of the features, already considered in the first project, is the implementation of a replication degree. Without this feature, if a node were holding a copy of a chunk and suddenly went offline, that file would be lost with no way of recovery until it went back on, destroying the purpose of a backing service. To reduce this chance, a replication degree associated is requested on a backup service, which makes it so that there is redundancy on that chunk.

Furthermore, a metadata file that keeps a peer's state is kept, so that some information can be preserved if a peer shuts off. This information contains the chunks stored in that peer, and the chunks that peer has already backed up and the remaining reclaim space.

TODO: Escrever as mesmas cenas sempre da mesma forma, por exemplo, ipAddress vs ip address vs IPAddress, etc. TODO: Ver o que fica a itálico, bold, etc.