Distributed Backup Service for the Internet

Final Report

Integrated Masters in Informatics and Computing Engineering

Distributed Systems

Grupo T6G22:
César Alves Nogueira - up201706828@fe.up.pt
Diogo Machado - up201706832@fe.up.pt
Gonçalo Marantes - up201706917@fe.up.pt
José Pedro Baptista - up201705255@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

May 29, 2020

Abstract

This project consists of a distributed backup service for the internet and was developed using communication protocols and secure channels, as well as concurrency and synchronization methods learned in the Distributed Systems Course. This application was developed using Java 11 and Open JDK 11.

Contents

1 Introduction

In this project we developed a distributed backup service for the internet. The main goal of this report is to clarify the key aspects of our implementation, such as:

- **Project Overview:** Overall project features.
- **Protocols:** Detailed description of methodologies, synchronization methods and data structures used which allow the concurrent execution of different protocols.
- **Concurrency:** Concurrent service request handling, using threads and thread pools.
- **Java Secure Socket Extension:** Description of *JSSE* use cases.
- **Scalability:** Design and implementation characteristics for scalability.
- **Fault-tolerance:** Design and implementation characteristics for fault-tolerance.

2 Project Overview

2.1 Features

This project was developed mainly using *Chord*[1] as its basis for data lookup protocols, scalabilty, fault-tolerance and synchronization. With the help of *JSSE*[2] we can guarantee secure communication channels between nodes in the Chord ring. And finally the use of *Threads* for concurrency help with optimizing implemented protocols.

2.2 Supported Operations

2.2.1 Backup

The *Backup* operation allows one of the peers of our system to call for the other peers to backup a selected file, with a specified desired replication degree (RD). For this effect, it divides the file into chunks, with a maximum size of 15kB. Then, each chunk is assigned an ID within the Chord network, and it will be stored in the *backup/* directory of the peer which is the successor of that ID (excluding the initiator peer), and in the RD following peers, for redundancy.

2.2.2 Restore

The *Restore* operation uses the chunks created by the *backup* operation to reconstruct a file. The initiator specifies the file name, and from there the program will compute the IDs of the file chunks, and send the request to retrieve them. After obtaining all of the chunks, it will reunite them into the original file, which is stored in the *restored/* directory of the initiator peer.

2.2.3 Delete

The *Delete* operation is requested by a peer to delete all occurrences of chunks from a specified file from the chord network. For this, the initiator will send a delete request to its successor, which will be propagated in the same way until it reaches the initiator once again.

2.2.4 Reclaim

The *Reclaim* operation is used to manage the occupied space in a given peer. The argument passed is the maximum space that the peer is allowed to occupy. While the maximum space of the peer will remain the same, this control is achieved by deleting chunks, one by one, until the occupied space becomes lower or equal than the specified amount (e.g. if the value is 0, then all of the chunks in this peer will be deleted). When deleting a chunk, if this deletion makes it fall under the desired replication degree for its backup, then a protocol is initiated to store the chunk in another peer which is neither the one that is deleting it nor one that has initiated the backup that originated that chunk.

2.2.5 State

The *State* operation can be called for a given peer to obtain its current storage situation. Taking no further arguments, this operation prints on the terminal the following items:

- *Peer Storage Section:* In this section, we have the ID of the peer in the chord network, the total capacity of the peer, and the amount of used and free space (in kB and in percentage)
- *Backed up files Section:* In this section, we can see the files that have been backed up by the peer, including the file path, the file ID (for the backup service), the desired replication degree for that backup and the perceived replication degree for each of the chunks that make up the file.
- *Stored Chunks Section:* In this section, we have the information for each of the chunks that were stored by this peer, from backups called by other peers. This information includes the number of the chunk within the original file, the id of the file (for the backup service), the size of the chunk, and the IP address and port for each of the peers that have initiated a backup for this chunk.

2.3 Compilation

In order to run this project there is a provided *Makefile* that takes care of compilation, cleaning, *RMI* execution and script execution for testing purposes.

```
# compilation
JC = javac
OUT_DIR := build/

all: mkdir
  $(JC) -d $(OUT_DIR) src/*.java src/*/*.java src/*/*/*.java

mkdir:
  @mkdir -p $(OUT_DIR)

clean:
  @rm -rf build/

rmi:
  rmiregistry -J-Djava.class.path=$(OUT_DIR) &

#execution
ADDRESS = "localhost"
PORT = 30001
KNOWN_ADDR = "localhost"
KNOWN_PORT = 30001
AP = "ap14661"
FILE = "text.txt"
RD = 1
SIZE = 100

start:
  @sh scripts/start_chord.sh $(ADDRESS) $(PORT)

join:
  @sh scripts/join_chord.sh $(ADDRESS) $(PORT) $(KNOWN_ADDR) $(
    KNOWN_PORT)

backup:
  @sh scripts/backup.sh $(AP) ../test/$(FILE) $(RD)

restore:
  @sh scripts/restore.sh $(AP) ../test/$(FILE)

delete:
  @sh scripts/delete.sh $(AP) ../test/$(FILE)
```

```
41
42 reclaim:
43   @sh scripts/reclaim.sh $(AP) $(SIZE)
44
45 state:
46   @sh scripts/state.sh $(AP)
```

Listing 1: *Makefile*

Simply execute *make* in the project's root folder to compile, creating a *build/* folder including all compiled Java classes. Run *make rmi* to execute the *RMI registry* inside the *build/* folder. And finally *make clean* to delete all the compiled classes and *build/* folder.

2.4 Execution

After having compiled the project and started the *RMI Registry*, there are scripts provided to help with the execution of the project. You can easily access these script with the use of our *Makefile*.

2.4.1 Node Initialization

To initialize the chord ring with the first node, simply execute *make start*, this will run a node in *localhost* using *port* 30001 as default. After to add more nodes to the chord network that we can simply execute *make join PORT=<port>*, where *<port>* is replaceable by the wanted port where the newly joined node will listen on. This will also make the new node run on our local network.

```
# create chord ring with the first node
make start
# it is possible to specify the port by executing
make start PORT=<port>
# please remember that changing the default initial port will also
# affect the way new peers are joined
```

Listing 2: Executing starting node

```
# join the chord ring with a new node
# remember that <port> must be different from 30001
make join PORT=<port>
# if the default 30001 was changed for the initial node or we want
# to connect to a different one on the same local network
make join PORT=<port> KNOWN_PORT=<known_port>
# however this only work for our local network
# if we want to connect to a different one we need to execute
make join PORT=<port> KNOWN_ADDRESS=<known_address> KNOWN_PORT=<
    known_port>
```

Listing 3: Executing new nodes

2.4.2 Operations

To execute the protocols, you just need to execute *make <protocol>*, where the options are *backup*, *restore*. *delete*, *reclaim* and *state*. You can specify the access point to be used for the protocol calls, using *make <protocol> AP=<access_point>*, where the access point has the format *ap<node_id>* (e.g. *ap64568*). In addition, you can specify the arguments for the specific protocols in the same way: you can use *FILE=<file_name>* in the *backup*, *restore* and *delete* protocols, to specify which file from our *test/* directory is used for the

execution of the protocol, you can use *RD=<replication_degree>* to specify the desired replication degree for the *backup* protocol, and finally you can specify the maximum space for the *reclaim* protocol using *SIZE=<maximum_size>*.

```
# Perform the backup of a file
make backup AP=<access_point> FILE=<file_name> RD=<rep_degree>

# Restore a previously backed up file
make restore AP=<access_point> FILE=<file_name>

# Delete the chunks of a backed up file
make delete AP=<access_point> FILE=<file_name>

# Reclaim disk space, by limiting the space occupied by a node
make reclaim AP=<access_point> SIZE=<maximum_size>

# Request the current state information of a node
make state AP=<access_point>

# Default values for the various arguments, if not specified:
#   > AP   :  ap64568  (the default chord starting node)
#   > FILE :  text.txt
#   > RD   :  1
#   > SIZE :  100

# Note (FILE): only the file name is required, as the script uses
#              the file with the specified name in our test/ folder
```

Listing 4: Running the various sub-protocols

3 Protocols

In this section we explain the logic and messages used in each of the implemented protocols.

3.1 Backup

After a backup request is received from the application, the node that received it, proceeds to split the file into chunks, and, for each chunk of the given file, calculates the id of the file in the chord ring, and finds its successor (by calling the *findSuccessor* method). Once it has the successor, it sends a *PUTCHUNK* message to it. This message has the following structure:

```
PUTCHUNK <file-id> <chunk-number> <replication-degree>
         <initiator-ip> <initiator-port> <first-successor-ip>
         <first-successor-port> <data>
```

Listing 5: *PUTCHUNK* message

In this case, the initiator is the node where the backup was requested, and the first-successor, the successor of the current chunk, in the chord ring. Once the initiator is done sending all of these messages and making the perceived replication degree for each chunk of the file 0, it returns. When a node receives a *PUTCHUNK* message, if the node is the initiator of the backup, it simply sends the received message to its successor. If that is not the case, it tries to store the chunk and, if it has sufficient space, stores it. If the replication-degree received in the message was greater than 1, it sends the message with replication-degree - 1 to its successor, and it's successor does the same until it the replication degree received is 0. If this message reaches the first-successor again, it realizes that it wasn't possible to reach the desired replication degree, and it sends the following message to the initiator peer:

```
UPDATERD <file-id> <chunk-number> <missing-replicas>
```

Listing 6: *UPDATERD* message

Whenever the initiator receives this message, it makes the occurrences of the chunk equal to the desired replication degree - missing replicas.

3.2 Restore

The restore protocol starts when the initiator node receives the request from the application. After receiving this request, the node finds the successor of each chunk of the requested file in the chord, and sends to it the following message:

```
GETCHUNK <initiator-ip> <initiator-port> <first-successor-ip>
         <first-successor-port> <file-id> <chunk-number> <hash>
```

Listing 7: *GETCHUNK* message

Once this request is made for all chunks, it waits until all the chunks are received (through a *CHUNK* message), or until it times out. For this, we are using a *CountDownLatch*, with size equal to the number of the chunks requested. The timeout period is 8 seconds without receiving any *CHUNK* message (of the requested file). The *CHUNK* message has the following format:

```
CHUNK <file-id> <chunk-number> <data>
```

Listing 8: *CHUNK* message

3.3 Delete

Once this protocol is started, the initiator node (the one that received the request from the application) deletes all stored chunks related to the file, as well as all occurrences. Proceeds to send the following message to its successor:

```
DELETE <origin-ip> <origin-port> <file-id>
```

Listing 9: *DELETE* message

After receiving this message, a peer does the same, and keeps the chain by sending it to its successor. Since the origin is the initiator peer, once the message arrives again, the initiator breaks the chain, ending the protocol.

3.4 Reclaim

When a node executes a reclaim, it deletes stored chunks up until the used space for chunk storage is less or equal to the specified value in the request. When a chunk is deleted, the node has to let the node that initiated the backup of the file know that it removed its copy of the chunk. It can be done by sending the following message:

```
REMOVED <file-id> <chunk-number>
```

Listing 10: *REMOVED* message

After receiving this message, the node decrements the stored replication degree for that file. If the new replication degree gets lower than the desired replication degree, it initializes a dedicated (sub)protocol to try to ensure desired replication degree, by sending a *ENSURERD* message to the successor of the file in the chord ring:

```
ENSURERD <initiator-ip> <initiator-port> <first-successor-ip>
         <first-successor-port> <hash> <file-id> <chunk-number>
```

Listing 11: *ENSURERD* message

Upon receiving this message, the node checks if it has the requested chunk stored. If it doesn't have the chunk stored, sends the same message to its successor. If there isn't any copy of the requested chunk stored in any of the nodes of the chord (this message arrived again at the first-successor node), the replication degree of the file won't be met, and a error message is displayed. If it has the requested chunk stored, it proceeds to send a *SAVECHUNK* message to its successor:

```
SAVECHUNK <file-id> <chunk-number> <hash> <initiator-ip> <initiator
    -port> <first-successor-ip> <first-successor-port> <data>
```

Listing 12: *SAVECHUNK* message

When a node receives this message, it checks if it has the conditions to store the received chunk (<data>). Similar to the *ENSURERD* message, if those conditions are not met, it sends it to its successor, until some node of the chord stores it, or until the message reaches the node that started this message chain, in which case, the replication degree of the file won't be met and a error message will be displayed. If the chunk gets accepted and stored by a node, it sends the following message to the initiator node (the node that started the *ENSURERD* protocol and the one with the occurrences stored):

```
CHUNKSAVED <file-id> <chunk-number>
```

Listing 13: *CHUNKSAVED* message

If a node receives this message, it increments the number of occurrences of the chunk. This usually that the *ENSURERD* protocol was successful in keeping the desired replication degree after a RECLAIM.

4 Concurrency

When implementing our project we took in consideration the vast number of messages that would need to be exchanged for synchronization between peers. For this reason we use *Threads* for node communication.

There are two main *Threads* running alongside our *ChordNode* process:

- *ChordMaintainer:* This *Runnable* class is responsible for maintaining chord node synchronization, making sure that each node is aware of other nodes.
- *ChordChannel:* This *Runnable* class is responsible for listening on the TCP Socket, and dispatching working *Threads* to handle message requests. This approach also allows for multiple messages to be handled concurrently.

```
private void startMaintainer() {
    // perform maintenance every half second 1.5 seconds after
    starting
    this.executor.scheduleWithFixedDelay(new ChordMaintainer(this),
     1500, 500, TimeUnit.MILLISECONDS);
}
```

Listing 14: *ChodeNode* Class *startMaintainer* Method

```
private void startChannel() {
    this.channel = new ChordChannel(this);
    this.executor.execute(this.channel);
}
```

Listing 15: *ChodeNode* Class *startMaintainer* Method

```
@Override
public void run() {
    while (true) {
        try {
            SSLSocket socket = (SSLSocket) serverSocket.accept();
            ObjectInputStream ois = new ObjectInputStream(socket.
    getInputStream());

            String message = (String) ois.readObject();
            ois.close();
            handleMessage(socket, message);
            socket.close();
        } catch (Exception e) {
            System.err.println("Node disconnected while reading
    from socket");
        }
    }
}
```

Listing 16: *ChordChannel* Class *run* Method

5 Java Secure Socket Extension

Our project also makes use of *JSSE*. In fact, it is used in all types of communication between nodes in the chord network to ensure that messages are exchanged securely, from requests and replies to file transfers. However the communication between the *Application* and its *Peer* is being done by *RMI*.

From the resources made available by *JSSE* we opted for *SSLSocket*. This class is instantiated every time it is necessary to send or receive a message.

From a sender's perspective its implementation is rather straightforward and very similar to a simple SSL devoid socket. We simply need to instantiate an *SSLSocket* object with the help of *SSLSocketFactory* class, after creating our SSL Socket we can use the *connect* method to connect to an *InetSocketAddress* object. This object contains information about an *IP* and *port* to which we must connect to.

After this, we create a *DataOutputStream* object with our socket's *OutputStream* attribute, and simply send our message in form of byte array.

```
SSLSocket socket = (SSLSocket) SSLSocketFactory.getDefault().
    createSocket();
socket.connect(address, timeout);
OutputStream os = socket.getOutputStream();
DataOutputStream dos = new DataOutputStream(os);
byte[] messageBytes = MyUtils.convertStringToByteArray(message);
dos.writeInt(messageBytes.length);
dos.write(messageBytes, 0, messageBytes.length);
dos.flush();
```

Listing 17: *ChordChannel* Class *sendMessage* Method snippet

Finally, our receiver uses the *SSLServerSocketFactory* class to create a server socket stored in a variable names *serverSocket*. This is done right at the start of our program. After that, our receiver listens for an incoming message and creates a new *SSLSocket* instance for each message it receives, which will be responsible for this particular message. At last, data is read from the socket and the message is dispatched to a working thread by calling the *handleMessage* method.

```
socket = (SSLSocket) serverSocket.accept();
InputStream is = socket.getInputStream();

DataInputStream dis = new DataInputStream(is);
int length = dis.readInt();

byte[] messageBytes = new byte[length];
dis.readFully(messageBytes, 0, messageBytes.length);
String message = MyUtils.convertByteArrayToString(messageBytes);

handleMessage(socket, message);

dis.close();
is.close();
```

Listing 18: *ChordChannel* Class *run* Method

6 Scalability

To ensure that our system is scalable we have implemented a Chord Network, a scalable Peer-to-peer lookup service for internet applications. This service guarantees efficiency in a dynamic network. It provides simply one operation: given a key, it maps that key to a node. This node can be responsible for storing a value associated with that key.

The main advantage of using chord is that the complexity of a simple lookup is equal to *log(n)*, where n is equal to the number of nodes in the network.

This section focuses on the chore principals of the Chord Network and also our changes to improve testing and simplicity.

6.1 Keys

One major aspect of chord is consistent hashing since all information is encrypted. For that we use a hashing algorithm based on the original *SHA-1*. This algorithm generates 16-bit keys. We have decided to use 16-bit keys instead of the original 160-bit keys to simplify our testing and chord operations.

However, 16-bit allows for a maximum of 65536 nodes on the network, which we deemed enough for our implementation. Because of this, each node stores 16 entries on its Finger Table.

6.2 Chord Ring

Inside the chord ring, for a single node to reach any other node in the network it is necessary for each node to store information regarding its successor, its predecessor and a finger table. Since each node stores information regarding its successor, the set of nodes make a ring.

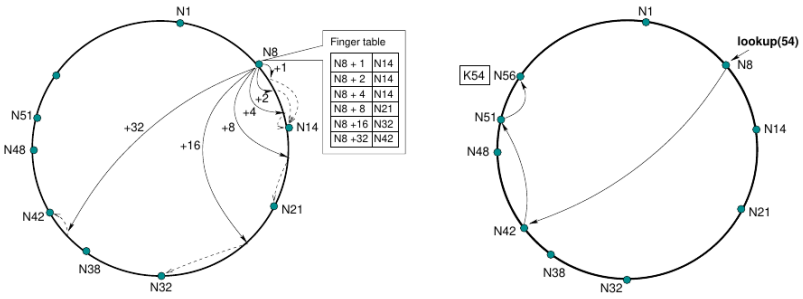

Figure 1: Lookup process of node 54 staring at node 8 in a Chord ring with 64 available nodes and 10 live nodes.

6.3 Finger Table

In order to avoid linear search and make it logarithmic, each node has its own finger table. This table contains information regarding m other nodes in the network. Note that m is equal to the number of bits used for hashing keys, in our implementation m is equal to 16. The first entry in a node's finger table is also its successor.

6.4 Lookup Key

Every time a node needs to lookup a key, it first checks if that key is between itself and its immediate successor. If it is not it then uses its finger table to find the closest preceding node of that key. For example, in Figure 1 when node 8 wants to lookup key 54, it first confirms that 54 is not between 8 and 14, then it moves on to find key 54's closest preceding node, which is 42. This process is repeated for node 42 until key 54 is between a node and its successor.

```
public synchronized String[] findSuccessor(InetSocketAddress
    requestOrigin, int id) {
    int successorId = this.getSuccessorId();

    if (successorId == this.getId()) {
        return this.channel.createSuccessorFoundMessage(id, this.
    getId(), this.getAddress()).split(" ");
    }
    else if (Utils.inBetween(id, this.getId(), successorId, this.m)
    ) {
        if (!requestOrigin.equals(this.getAddress())) {
            this.channel.sendSuccessorFound(requestOrigin, id, this
    .getSuccessorId(), this.getSuccessorAddress());
            return null;
        }
        else {
            return this.channel.createSuccessorFoundMessage(id,
    successorId, this.getSuccessor().getValue()).split(" ");
        }
    }
    else {
        InetSocketAddress closestPrecedingNode = this.
    getClosestPreceding(id);
        return this.channel.sendFindSuccessorMessage(requestOrigin,
     id, closestPrecedingNode);
    }
}
```

Listing 19: *ChordNode* Class *findSuccessor* Method

6.5 Stabilization

In order for our solution to be scalable, it must support dynamic node operations, i.e when nodes join or leave the network. For this reason a stabilization protocol is performed periodically that updates a node's successor and finger tables.

6.5.1 Stabilize

Given a node N, it asks its successor for its predecessor P and decides if P should be N's successor. This is the case when P is a newly joined node.

```
private void stabilize() {
    // send message to node's successor asking for its predecessor
    // message handler will take care of the rest
    NodePair<Integer, InetSocketAddress> successor = this.node.
    getSuccessor();
    this.node.getChannel().sendGetPredecessorMessage(this.node.
    getAddress(), successor.getValue());
}
```

Listing 20: *ChordMaintainer* Class *stabilize* Method

After node N receives a reply from its successor it also notifies its current successor, which may or may not be P about N's existence. This allows N's successor to update its current predecessor to N.

6.5.2 Fix Fingers

For each key on N's finger table it performs a lookup of those keys, and after receiving a reply it updates its finger table. This is a necessary operation since newly joined nodes can change N's finger table.

```
private void fixFingers() {
    // get, update and set finger
    int finger = this.node.getFinger();
    finger = (finger + 1) % this.node.getM();
    node.setFinger(finger);
    // calculate new node ID
    Integer nodeID = (node.getId() + (int) Math.pow(2, finger)) % (
    int) Math.pow(2, this.node.getM());
    // find node's successor
    String[] reply = this.node.findSuccessor(nodeID);
    // if successor fails let the handler deal with it
    if (reply == null)
        return;
    // build reply node
    int replyNodeId = Integer.parseInt(reply[2]);
    InetSocketAddress replyNodeInfo = new InetSocketAddress(reply
    [3], Integer.parseInt(reply[4]));
    NodePair<Integer, InetSocketAddress> replyNode = new NodePair
    <>(replyNodeId, replyNodeInfo);
    // update entry in finger table with index 'finger'
    node.setFingerTableEntry(finger, replyNode);
}
```

Listing 21: *ChordMaintainer* Class *fixFingers* Method

6.5.3 Check Predecessor

Node N checks if its predecessor is still alive, if not than it sets is predecessor as null.

```
private void checkPredecessor() {
    // node may not have a predecessor yet
    if (node.getPredecessor().getKey() == null)
        return;
    // send message to predecessor
    boolean online = node.getChannel().sendPingMessage(this.node.
    getAddress(), this.node.getPredecessor().getValue());
    // if he does not respond set it as failed (null)
    if (!online)
        node.setPredecessor(new NodePair<>(null, null));
}
```

Listing 22: *ChordMaintainer* Class *fixFingers* Method

7 Fault-tolerance

7.1 Chord Ring

The correctness of the Chord protocol relies on the fact that each node knows its successor. However, this invariant can be compromised if nodes fail. For instance, in Figure 1, if nodes 14, 21 and 32 fail simultaneously, node 8 will not know that node 38 is now its new successor, since it has no finger pointing to 38. This causes problems due to the fact that an incorrect successor will lead to incorrect lookups. Consider a query for key 38 initiated by node 8. Node 8 will return node 42, the first node it knows about from its finger table, instead of the correct successor, node 38.

To increase robustness, each Chord node maintains a successor list of size r instead of a single successor. Now each node contains information regarding its first r successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list.

In order to disrupt the Chord ring, all r successors would have to fail simultaneously, and event that can be highly improbable with the increase of r.

8 Conclusion

Having completed this project we have learned a lot about the Chord protocol, including its usefulness and fault-tolerance capabilities as a distributed backup service.

In addition, the use of secure communication channels also made us realize the importance of security in data exchange over the internet.

References

[1] dl.acm.org. (2020). Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. [online] Available at: https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407

[2] docs.oracle.com. (2020). CopyOnWriteArrayList (Java SE 11 & JDK 11). [online] https://docs.oracle.com/en/java/javase/11/security/java-secure-socket-extension-jsse-reference-guide.html

[3] docs.oracle.com. (2020). Java Secure Socket Extension (JSSE) Reference Guide. [online] Available at: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CopyOnWriteArrayList.html

[4] docs.oracle.com. (2020). ScheduledThreadPoolExecutor (Java SE 11 & JDK 11) [online] Available at: https://docs.oracle.com/en/java/javase/ 11/docs/api/java.base/java/util/concurrent/ScheduledThreadPoolExecutor.html

[5] baeldung.com. (2020). SHA-256 Hashing in Java — Baeldung [online] Available at: https://www.baeldung.com/sha-256-hashing-java