Distributed Backup Service*

Creating a Scalable and Distributed Backup System using Chord, JSSE, Java NIO, RMI calls and Thread Pools in Java

Ângelo Teixeira[up201606516], Henrique Lima[up201606525], Mariana Aguiar[up201605904], and Tiago Fragoso[up201606040]

Group 07, Class 05 Distributed Systems Course, Integrated Masters in Informatics and Computing Engineering, Faculty of Engineering of the University of Porto

Abstract. In this document we will demonstrate how we implemented the proposed **Distributed Backup System** as well as some technical decisions we took in order to accommodate some enhancements. The Service relies on **Chord** for the scalability property, **JSSE** for the secure messaging between Network Nodes, **Java NIO** and **Thread Pools** for the non-blocking procedures, maintaining asynchronicity. **RMI (Remote Method Invocation)** is used by a client application to trigger the protocols in the network (*Backup, Restore, Delete, Reclaim and State*), communicating with a Node via its *Access Point Name*. The used language was **Java**.

Keywords: Distributed Systems · Chord · JSSE · Java NIO · RMI · Java.

1 Introduction

This document is divided into sections, for an easier comprehension. In section 2, we explain the architecture behind the communication between Nodes. In section 3, we talk about how we tackled the scalability problem, namely by using the Chord [1] protocol, explaining our implementation of the said algorithm. In section 4 we mention the implementation of the asynchronous procedures and how the system can handle multiple requests at any given time, without blocking each other. In section 5 we present the implementation of secure communication between Nodes, namely by using SSL on top of the TCP connection. In section 6 , we explain the different protocols of the service, available to the client application. Finally, in section 7, we review our work, while also stating what we would like to have done additionally in this project, in case we had the time.

The Project is divided into multiple packages. The main package - the application - is in com.dbs, standing for distributed backup system. Inside it there are five packages: *chord, filemanager, network, protocols and utils* which are closely related to the sections in this report. Each section will reference the package and the code when necessary.

* Project made as part of the Distributed Systems Unit Course - FEUP

2 Communication Design

In order to have the possibility of having simultaneous procedures happening on a Node at the same time, it was necessary to communicate asynchronously, that way we created a package for this purpose: the *network* package.

2.1 Communicator

The main class is the ***Communicator*** which allows a node to send and receive messages. On the Node startup, it creates a Communicator instance, and creates a new thread which will call $Communicator :: listen$ (ln. 35) on loop. As new messages come for that node, they are handled on a new thread accordingly (each message type has a handle function which is called on arrival). This way, the Node never loses messages due to blocking, because when it receives a message, it deals with it in a new thread, while the first one is already waiting for more messages. These threads are acquired from the Node's Thread Pool.

The Communicator has two more functions for listening for messages, which have a different purpose than the first one. The $Communicator :: listenOnSocket$ (ln. 63) and $Communicator :: async\_listenOnSocket$ (ln. 86) are made so that they wait for a message on a given Server Socket, differing on the return type and the places used. While the first one is used more on the Nodes communication for the Chord Protocol, the second is used more for the Backup Protocols, where the *CompletableFuture API* is more helpful. These functions are used to listen for responses to a certain request. Instead of listening to every message on to main *listen* loop, we chose to separate it, due to the fact that we don't have to create so many handlers.

With this architecture, we send a request (to find a successor or to ask for a file backup, for example) and in this request we attach the socket info to where the receiving Node must answer, which will be a temporary socket created for this purpose. We then call one of the *listenOnSocket* functions with the temporary socket and receive the answer, returning it to the calling function, maintaining the program flow more understandable, as it *feels* synchronous.

Finally, the Communicator has a function to send messages $Communicator :: send$ (ln. 113) which sends a given message to a given socket.

2.2 Messages

As for the messages themselves, we use a Java Class for each message, so that we can simply read them and write them with *readObject* and *writeObject*, without worrying with more low-level concerns. The used messages are available in the *network.messages* package.

All the messages extend a superclass ***ChordMessage*** which is an abstract class with the un-implemented method $ChordMessage :: handle$ (ln. 86) which called upon reading the messages. This way, when a message is read, it is handled correctly, depending on the message type.

Some of the messages needed to have some info about a Node attached. For example, the *Successor Message* has the successor attached to the message and is sent directly to the node which requested the successor initially, without recursively responding to every node from which the request hopped. That way, when the requester receives the message, by calling $message.getNode()$ it learns about the successor it was looking for. For this reason, we created another abstract class, which extends *ChordMessage* which is *NodeInfoMessage* and is required to have a Node info attached.

3 Scalability

The scalability of the application relies on the fact that we use Chord [1], to maintain the network of nodes connected and able to communicate with each other despite the fact that Nodes can enter the network at any time, being part of it from then on. A detailed explanation of the Chord protocol is available at [1].

In our project, the Chord implementation is available on the *chord* package. It has a ***Chord*** interface, which specifies the required functions to implement the protocol and the ***Node*** class implements that interface, and represents a Node of the system, which will connect to the network (or create one if there is none - it will be the starter node) and be responsible to backup the files assigned to it (not always the case, in the Protocols section - section 6 we get into more detail).

As is stated on the Chord paper [1], the lifecycle of a Node is as follows:

1. Join or create the network ($Node :: join$ (ln. 233) or $Node :: create$ (ln. 223))
2. Stabilize Network (called periodically) ($Node :: stabilize$ (ln. 278))
3. Fix Fingers of the Finger Table (called periodically) ($Node :: fixFingers$ (ln. 348))
4. Check if Predecessor is online (called periodically) ($Node :: checkPredecessor$ (ln. 321))

Some other useful Chord methods are $findSuccessor$ ($Node :: findSuccessor$ (ln. 128)), which finds the successor for a given key and $closestPrecedingNode$ ($Node :: closestPrecedingNode$ (ln. 159)), which returns the closest preceding node of a given key, based on the Finger Table entries.

The fact that we use Chord, makes the application scalable, because one of the properties of chord is that it has *Consistent Hashing*, meaning that each Node entering or leaving has minimal disruption on the hash tables of other nodes, making the network very stable.

4 Concurrency design

In order to provide an always accessible service, the application needs to deal with requests asynchronously and without blocking more resources than necessary.

To allow this, each Node has a Thread Pool, which the procedures use to handle the requests in different threads. One example is the Communicator, mentioned above, which listens in a separated thread, blocking only that thread, and handles each request in a new thread, so it can get back to listening right away.

Another part of the *Concurrency Design* is made in the *FileManager* (*filemanager* package) where the static utility functions to manipulate the file system use the Java NIO API so that it processes files (read/write) without blocking, allowing for concurrent requests regarding the same resources.

5 Security

Through the use of **JSSE** (Java Secure Socket Extension), it is possible to guarantee that the data travelling across the network is only accessible by the intended recipient. Specifically, this can be achieved through the javax.net.ssl.**SSLSocket** class, subclass of java.net.**Socket** and the javax.net.ssl.**SSLServerSocket** class, subclass of java.net.**ServerSocket**. In our project, all sockets are **SSLSockets**. The SSLServerSockets are created through a wrapper function ***createServerSocket***(*utils.Network* : 76) which is responsible creating the ServerSocket with some custom parameters. A specific *SSLParameter*(*SSLParameters.setUseCipherSuitesOrder*(*true*)) is used to ensure that server cipher suites are the preferred ones, thus allowing us to alter the cipher suites on ServerSockets only (*utils.Network* : 84).

Regarding the cipher suites, some research was made to reduce the set of available to a subset with the strongest. Firstly, the key exchange algorithm used is **ECDHE**. Although **ECDHE** does not provide authentication, this was enabled through the following method: *socket.setNeedClientAuth*(*true*) (*utils.Network* : 82). Next, two certificate authorities are allowed: **ECDSA** and **RSA**, for compatibility. The preferred message encryption mechanism preferred is AES 256-bit with GCM, which provides encryption, as well as the CBC does, but also integrity checking, which the latter does not. Finally, the preferred hashing algorithm is SHA-384, which provides the strongest hashes. For the sake of compatibility with legacy systems (and some recent macOS systems), some weaker ciphers were included, namely using AES 128-bit and CBC. Since the available cipher suites are implementation and JDK dependant, this check is done programatically on the first execution of the aforementioned function (*utils.Network* :: *createSeverSocket*), through the use of the **filterCipherSuites** function (*utils.Network* : 64). Therefore, the cipher suites in preferred order are as follows:

1. TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
2. TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
3. TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
4. TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256

6 Protocols

A big part of this project was to build a backup service, which is able to store and retrieve files across a distributed network of Nodes on top of the mentioned Chord protocol.

Every *Manager* relies on a structure we created to store the current Node's state, which is the ***State*** class (*utils.State*) which stores, for the current Node, the stored replicas (*localReplicas*), the replicas for which the Node is responsible and their location (*replicasLocation*). If they are stored locally, the location will be the current Node. The State also has the current max allocation space to know if a Node can store any more replicas.

To simplify, one can say that the *localReplicas* stores the information about the replicas the node has, even though it may not be responsible for them (not their successor), and the *replicasLocation* stores the location of every replica of which the Node is responsible for.

The *localReplicas ConcurrentHashMap* maps a *FileIdentifier* to a Set of *ReplicaIdentifier*s, so that we can only account for the size of a file, even though we may have multiple replicas.

To implement the service, we provide the following functionalities to a client that wishes to communicate with the network:

- Backup - Store a file in the network, with a specified replication degree (i.e. the number of replicas of the file)
- Restore - Fetch a file from the network
- Delete - Delete a file (including all replicas) from the network
- Reclaim - Change the available space in a Node, which can lead to some redistribution of the stored replicas in order to keep the occupied space below the limit, in case the limit is lowered.
- State - Show the state of a Node, regarding the saved replicas, the replicas it is responsible for (even if it does not have them stored), and the occupied and total space

The implementation of these protocols is done in the *protocols* package, each protocol having its own sub package.

As the protocols are triggered via an external client application, through an RMI call to the desired node, we saw fit to have an Interface (***IDistributedBackupService***) with all the methods necessary, one for each protocol. Then, we created an adapter (***DistributedBackupServiceAdapter***) which implements the said interface, implementing the methods by calling the respective Manager for the protocol.

The following subsections dive into the implementation details of each protocol for a better understanding.

6.1 Backup

The ***Backup Protocol*** is ensured by the *BackupManager* (*protocols.backup* package).

This protocol is initiated by the ***backup*** method (*protocols.backup.BackupManager* : 46) and its procedure consists of generating n replicas of a given file, in order to have n replica keys. The hash is formed with the seed $filename + creationDate + replicaNumber$, computed by the ***generateReplicaIds*** ($filemanager.FileManager$: 128). So for 3 replicas of a file, there would be three hashes (keys) where the replica number varies from 0 to 2, leading to different hashes. This is made so that the replicas are stored in different nodes, so that, in the event that one replica is lost/corrupted, the others can be recovered.

Having the replicas' keys, the successors of those keys are found, through the ***findSuccessor*** method (*chord.Node* : 128). From then, a *BackupRequestMessage* is sent by the ***requestBackup*** method (*protocols.backup.BackupManager* : 228), to find out if the successor can store the replica. It will then respond with either a *BackupACKMessage*, meaning it wants to store the replica, *BackupNACKMessage*, meaning that it has no space to store the replica [1], or a *BackupConfirmMessage*, in case that it already had that replica stored, or any replica of the same file, in that case, it will simply store the fact that it ha the new replica, without actually storing a duplicate file, to save space.

In case the answer is a *BackupACKMessage*, the initial requester will then send a *BackupPayloadMessage* with the file content, to relieve the network from sending big files when they can't actually be stored. The receiver node, upon receiving the file content checks again if it still can store the file, which may not be possible due to simultaneous requests, and sends back a *BackupConfirmMessage* in case of success or *BackupNACKMessage* in case of failure.

On the special case of the successor being the node itself, the messages are not required, and it verifies locally if the conditions are good to store the replica.

Additionally, the *Backup Manager* stores the number of desired replicas of each file it was requested to backup (this is the reason why only the initiator node of the backup can restore the file it backed up) in a *ConcurrentHashMap* called *desiredFileRepDegree*, which maps each file to its desired replication degree, i.e, the number of replicas there are in the network for that file, in optimal conditions. This will be useful for the *Restore Manager* to know how many replicas there may be for it to try to fetch.

[1] The backup protocol actually tries to save the file wherever it can, meaning if there is no space, there was no Node with enough space to store the replica. For example, if the responsible node has no space to store the replica, it will delegate the task to its successor, and if also has no space, it will propagate, until it comes back to the responsible node. Once a node which can store is found, it answers back to the responsible node, which will store the actual location of the replica for which it is responsible. This is necessary, because of the way Chord works, as we can only find a key (file) through its successor, which is the responsible node. This node will now know where to fetch the file when it is asked to.

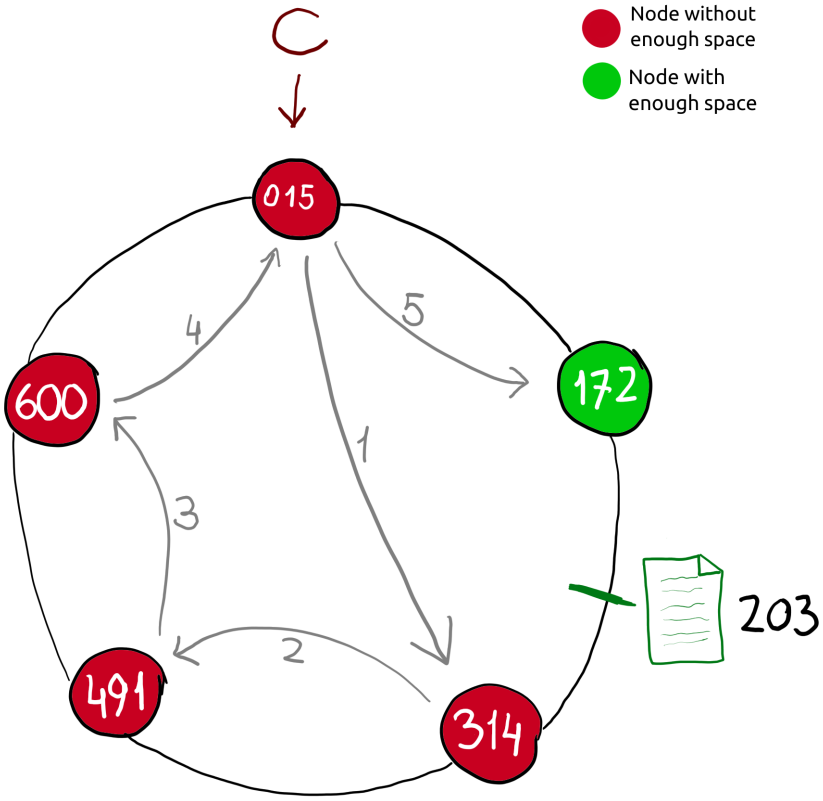

Fig. 1. Backup Protocol - Saving replica with key 203 on the network

An example of a backup can be seen in Fig. 1 where the client asks Node 015 to backup a replica with an id of 203. Node 015 then finds out that the successor of the replica, and therefore the responsible node for its storage is Node 314, requesting the backup to that node instead. As Node 314 has no space to store, it will propagate the request to Node 491, its successor, which has no space as well, and therefore will also propagate the request to its successor, Node 600. The cycle continues until the request reaches Node 172, which has enough space to store the replica. It will then send a confirmation to Node 314, stating that it has stored the file, and Node 314 will notify Node 015 so that it can respond and conclude the client's request.

If the Node 172 had no space, it would propagate the request to Node 314, which would conclude the cycle, and therefore Node 314 would realize that there was no space in any node to store the replica, answering back to Node 015, which will respond with an error to the client.

6.2 Restore

The ***Restore Protocol*** is ensured by the *RestoreManager* (*protocols.restore* package).

This protocol consists of retrieving any replica of a given file and is initiated by the ***restore*** method (*protocols.restore.RestoreManager* : 39). If the initiator node has stored the file, then there is no need to go around the network trying to find it and it can be instantaneously saved into the *restored/* directory of the node. However, most times the initiator node has not stored the given file, thus a search must be performed. Using the aforementioned *ConcurrentHashMap desiredFileRepDegree* of the *Backup Manager*, it is possible to retrieve the desired replication degree of the file and thus, the number of replicas. Following the same procedure as mentioned in the Backup protocol, the keys for each replica are generated and are used to find the successors of those keys. Then, a message *RestoreRequestMessage* asking for the file in question is sent by the ***requestRestore*** method (*protocols.restore.RestoreManager* : 70), to which the node can respond with a *RestorePayloadMessage*, in case it has the file. If this is not the case, two scenarios can occur: the node knows where the replica is and relays the *RestoreRequestMessage* to that node; or the node does not have the file nor does it know here it is and responds with a *NotFoundMessage*. Handling the restore request is performed by the ***handleRestoreRequest*** method (*protocols.restore.RestoreManager* : 119), which is called upon reception of a *RestoreRequestMessage*.

Once a node receives the *RestorePayloadMessage* containing the file data, it saves it to the *restored/* directory, using the ***storeRestorePayload*** method (*protocols.restore.RestoreManager* : 107). In case the request is not satisfied, the next replica is fetched from the network.

An example of a restore can be seen in Fig. 2 where the client asks Node 015 to retrieve a replica with an id of 203. Node 015 then finds out that the successor of the replica, and therefore the responsible node for its storage is Node 314, requesting the restore from that node. As Node 314 has not stored

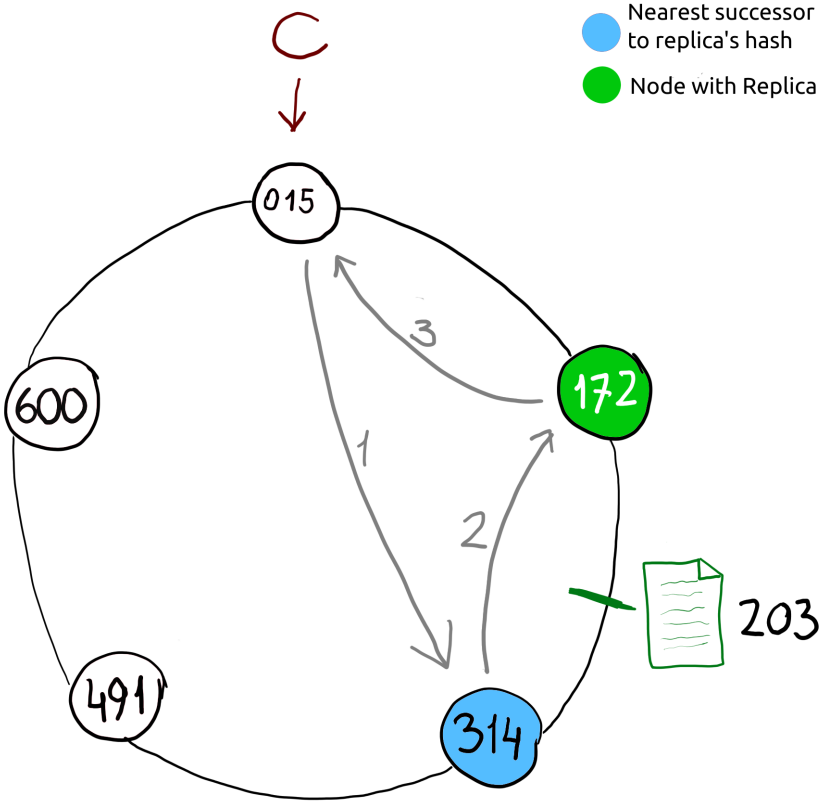

Fig. 2. Restore Protocol - Retrieving replica 203 from the network

it but knows its location, it will propagate the request to Node 172, where the replica is stored. This Node will then respond with the payload to the original Node 015.

6.3 Delete

The ***Delete Protocol*** is ensured by the *DeleteManager* (*protocols.delete* package).

This protocol consists of deleting all replicas of a given file in the network and is initiated by the ***delete*** method (*protocols.delete.DeleteManager* : 36) To do so, it starts by generating the replica keys for all replicas of the file and finding the successor of the keys. Then, a *DeleteReplicaMessage* is sent to each and every one of the nodes found in the ***deleteReplica*** method (*protocols.delete.DeleteManager* : 77). Upon reception of this message, each node checks for the replica's existence and sends a confirmation *DeleteConfirmationMessage* if it has deleted the replica. If this is not the case, two scenarios can occur: the node knows where the replica is and relays the *DeleteReplicaMessage* to that node; or the node does not have the file nor does it know here it is and responds with a *NotFoundMessage*. The node also deletes any record related to the replica. Handling the delete request is performed by the ***deleteReplica*** method (*protocols.delete.DeleteManager* : 111), which is called upon reception of a *DeleteReplicaMessage*.

The initiator node also collects all the delete confirmations and logs the results.

6.4 Reclaim

The ***Reclaim Protocol*** is ensured by the *ReclaimManager* (*protocols.reclaim* package).

This protocol consists of limiting the storage space of a node and, consequently, find a new "home" for the replicas it must delete. It is initiated by the ***reclaim*** method (*protocols.reclaim.ReclaimManager* : 31). This poses a problem as the *Chord* algorithm computes the key location given only the key itself, thus a change in location would make it impossible to find the location of a replica. To tackle this problem, we kept the node as responsible for the replica but stored it in another node, registering the new location.

Firstly, the node chooses which replicas it must delete to comply with new maximum storage space, using the ***freeSpace*** method (*utils.State* : 175). Then, it initiates a backup protocol for each of the replicas it must delete, updating the replicas location in its *State* as the backup requests are satisfied.

6.5 State

The ***State Protocol*** is the most simple of all protocols, as it does not require any communication between nodes.

Due to being so simple, we decided to include it in the *BackupManager* (*protocols.backup.BackupManager* : 185). It will simply access the Node's state and print the relevant information such as the stored replicas, the replicas for which the node is responsible, and their location in the network. It also shows the space occupied with the backed up replicas, as well as the total available space.

6.6 Extra - Redistribution Protocol

As this system deals with files, we must integrate the Chord's behaviour with the backup service so, for example, when a new Node joins the network, it notifies its new successor about being its new predecessor. Upon this, some files previously to the responsibility to the existing node, must be transferred to the new node, as it is now the successor of those files. This is done by applying the ***Redistribution Protocol***.

An example of this is shown in Fig. 3. In this case, no files are transferred. As Node 314 didn't have the replica on it already (it was actually stored in Node 172), it simply removes the replica from its *replicasLocation* and informs Node 259 that the replica is stored in Node 172 through a *UpdateReplicaLocationMessage* and when some node looks for that replica, the new successor, the Node 259, will know about the location of replica 203.

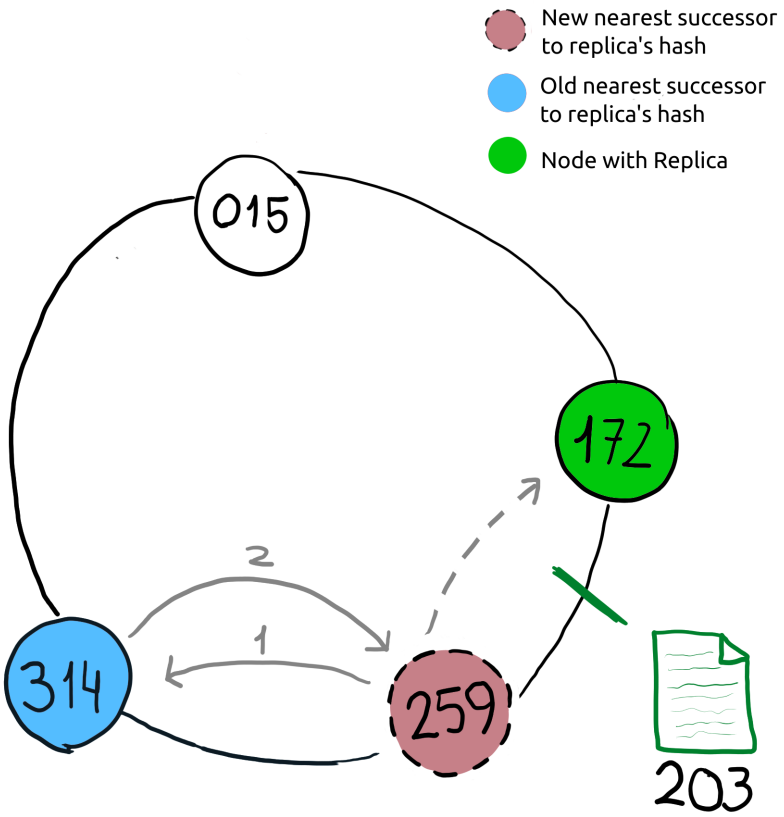

Fig. 3. Redistribution Protocol - Node 259 joins the network, making Node 314 transfer the responsibility of replica 203

7 Conclusions and Future Work

Reviewing our project, we think we have created a robust solution for what it aims to be: a distributed backup service, which allows for multiple nodes to store multiple and a variable number of replicas of given files. It allows for nodes to enter the system and adapts to it, however we have a bittersweet feeling because the node leavings, despite being handled, do not make the network adapt, as Fault Tolerance is not implemented fully. Thus a very important feature to be added in a future iteration would be to allow and handle fault tolerance in the network, following the protocol described in [1] for Fault Tolerance in Chord. On the other hand, we made possible for a file to be replicated in a node that is not its successor, making the most out of the available network, which we think is a good enhancement.

Another thing to improve could be the selection of replicas to release when reclaiming space. At the moment, the program simply iterates over the files and marks them to release until it has released enough space. However we could release them according to a better heuristic by selecting, for example, the smaller ones, which might be more prone to be saved in another place of the network, as they are small and can fit better in another node. On the other hand, releasing the big files first would imply releasing less files, leading to less messages and a lesser congestion of the network.

To sum up, this project had a little bit of everything: Network Communication, Security, Architectural Design, Data Structures and Algorithms Conceptualization, Design Patterns, Multi-threading, Distributed Systems theory, so we think it was very enriching and rewarding to be able to implement it and we vouch for its continuation for the years to come.

References

1. Ion Stoica et al.: Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications (2003)