SDIS 2019/2020

Project 2: Distributed Backup System for the Internet

André Filipe Magalhães Rocha - up201706462
David Luís Dias da Silva - up201705373
Manuel Monge dos Santos Pereira Coutinho - up201704211
Mário Gil Marinho Mesquita - up201705723

This report contains implementation details of the second SDIS project “Distributed Backup System for the Internet”. The first section contains an overview of the overall project, followed by a more in depth explanation of the implementation of the used protocols, the concurrency design and how we improved the scalability and assured fault-tolerance of our system.

1. Overview

Our backup system supports the same operations as the ones described in project 1 (backup, restore, delete, reclaim and state). We improved upon them, making it possible to perform **every task from every peer** (even if it isn’t the one who initiated the backup of the desired file).

The identification of the file passed as an argument in the restore and delete operations must be the file **UUID**, based on the filename and metadata (Backup.java: line 117). This allows us to promote a good accessibility, making it possible to access the file from any peer, and ensure that files with similar names do not create conflicts between themselves.

We allow the user to backup files with a **maximum replication degree** of 10 to make the system more robust to peer failure or other unexpected behaviours and to improve concurrency we make use of thread pools and asynchronous sockets channels to allow non blocking IO operations. This also deters peers from exhausting all available space of the system by issuing an unusually large replication degree.

The ceiling grade that we propose is **20**:

- TCP with thread-based concurrency - 14
- Scalability: Chord with asynchronous I/O from the Java NIO library - 14+4 = 18
- Fault-tolerance: Chord’s fault tolerant features with chunk replication and peer shutdown control (described in future chapters) - 18+2 = 20

2. Protocols

In this chapter we will start by briefly explaining how the overall system works and then we will detail each operation in its subchapter.

A Peer can be started using the following command:

```
java com.feup.sdis.peer.Peer <hostPort> <peerID> <peerAp> <peerSpace> [chordBoostrappingAddress]
```

- <hostPort> - Port where the backup service will operate;
- <peerID> - ID of the peer (must be unique);
- <peerAp> - Is the peer's access point in RMI registry;
- <peerSpace> - Maximum file system memory allocated to the peer (in megabytes);
- [chordBoostrappingAddress] - Address (in the format IP:port) of the peer that is used for bootstrapping the chord. These argument is only needed when a peer is joining an existing chord ring.

The **ID of a peer** is created from **hashing the <peerID> argument** using JavaSE's **UUID.** In a "real-world" scenario we could choose to use different "seeds" for the hashed ID, depending on the system's conditions. If every Peer were to be run on **different machines** we could hash the IP address (that would be different for each peer) and use a constant port for the Distributed Backup Service among all peers. However, if **different instances** of the backup system were to be ran, we would choose to hash the pair IP:port (where the port is constant for each instance of the system), this way there would be no collisions between instances of the system running on the same protocol. Since, in our case, every peer operated in the **same machine** and had the same IP address, a correct approach would be to hash the port that was passed by argument. To increase the testing flexibility and having all these approaches into consideration we chose to have an additional peer ID argument.

To allow the communication between the TestApp and the Peers we use RMI with a similar interface to the previous project:

```
java com.feup.sdis.TestApp <hostname[:port]> <peerAp> <operation> [opnd]*
```

- <hostname[:port]> - Location (IP and port) of the RMI registry;
- <peerAp> - Is the peer's access point in RMI registry;
- <operation> - Is the operation the peer must execute. It can either trigger a subprotocol, SHUTDOWN a peer or retrieve its state with the STATE argument;
- [opnd] - operands of the specific protocol, specified in the the corresponding subchapter.

The communication between peers is achieved using **TCP without JSSE**, but **with asynchronous channels** as described in chapter 4.

To uniquely identify a peer and to provide information about the network address where it can be reached we created a class called **SocketAddress**. This class is in the chord package and contains the peer's ID (in **UUID** format), it's IP address and port.

In this project the make use of Java's capabilities and write the **Message** object directly avoiding unnecessary conversions between String and Message, resulting in a cleaner code. We divide this class in two main subclasses: **Request** (package com.feup.sdis.messages.requests) which all have an handle that returns a Response and a method to get the SocketAddress of the corresponding request; **Response**, with the status indicating whether the request was successful or not and which error occurred (package com.feup.sdis.messages.responses). Each of these classes is subdivided in more specific messages that will be described in due time, implementing the strategy design pattern.

A **chunk-based** system similar to the one proposed in the first project is used in combination with the use of **redirects**: in order to avoid one peer saving two similar chunks, it will have to redirect one of them to its successor and save this information because the Chord system will think, and rightfully so, that it is the owner of those chunks, being this peer responsible for telling where it really is.

2.1 Backup Subprotocol

To initiate the backup of a file, the user must call the ***TestApp*** with the operation parameter as **BACKUP** and 2 operands: the first one being the **path to the file** they desire to backup, absolute or relative, and the second being the **replication degre**e that symbolizes the number of times that the peer will try to replicate the file in the system.

If the backup protocol fails, meaning not every chunk reached the specified replication degree due to no space left, a message will be returned to the user and all the chunks will be deleted from the system using the delete subprotocol to ensure the consistency of the peers.

The backup is made in **2 phases** (ignoring the delete part) for each pair chunk - replication:

2.1.1 Phase 1 - Backup Lookup Request

After subdividing the file into chunks, the backup subprotocol will **lookup** for the corresponding **owner** of the pair chunk-replication (ChunkBackup.java: line 40). Once found, it will send a message with the **chunk length** (*BackupLookupRequest*). If this peer was either the initiator, already has this chunk of the file with a different replication number or doesn't have sufficient space, it will **redirect** this request to its **successor**. This process is done **recursively** until we either reach the beginning of the circle again or a peer that meets this conditions is found.

In the first case a message will circulate back until the initiator peer with the status *NO_SPACE* and the last (that is the same as the first) Peer address. If all goes well, in the

way back each peer on the way saves a redirect to the final position and passes back the message with the status *SUCCESS* and the valid peer address.

2.1.2 Phase 2 - Backup Request

Now that we have the final destination (if phase one was successful), we can contact this peer directly avoiding polluting the system with heavy chunks being passed around (ChunkBackup.java: line 52).

In order to perform this, we use the *BackupRequest* that returns successfully if it all went according to plan and the chunk was saved or returns *ERROR* if there was a delete request for the same file meanwhile. Both this phases are abstracted via the *ChunkBackup* class, that is called on the Backup's process method once for every pair of chunkNumber-replicationDegree.

By pointing directly to the ending chunk we guarantee that we will only have to send **at most two messages** to find him.

2.2 Delete Subprotocol

This subprotocol has only one operand (the file *UUID*) and must be called using its uppercase name (***DELETE***). The *UUID* is used so that we can perform this operation in **every peer**, and not only the initiator. In order to being able to do this, this protocol is also divided into two phases. The first one is common to the restore subprotocol, and we will only describe it here once for the sake of brevity.

2.2.1 Phase 1 - Get Chunk Info

We don't have the information needed to delete or replicate one entire file (we don't have the replication number, the number of chunks, the correct final file name…). So we first iterate, up to the maximum replication degree (10), asking for the initial chunk in order to get all the information that we need.

We begin by sending a ChunkLookupRequest that resolves the redirects (gets the address where the chunk truly is, Delete.java: line 31), and just after that do we use the *GetChunkInfoRequest* to communicate directly with the final peer, getting all the information that we need.

This process is made with the *getChunkInfo* static function present in the *Restore* class, line 106.

2.2.2 Phase 2 - Chunk deletion

Now that we have all the information that we need, we begin by deleting the file from the backup ledger of the initiator peer with the *DeleteFileInfo* request and then we iterate

through all the chunks and replications deleting each one, using the *deleteChunk* static function in the *Delete* class, line 65.

This process begins in the owner of each chunk-replication pair and goes through the successor until the peer where the chunk really is, deleting all the redirects on the way in order to keep the system neat and clean. Once it reaches the final peer (that can also be the first), it removes it completely from the folder structure and sends back the status: *FILE_NOT_FOUND*, if the chunk was not present in the final address; *CONNECTION_ERROR*, if the connection failed during this process; or *SUCCESS*, if the operation was performed successfully for that chunk.

2.3 Restore Subprotocol

This protocol is very similar to the previous one: the *TestApp* is called the same way (one operand with **UUID of the desired file** and de **name of the operation** in uppercase - **RESTORE**); and the first phase is exactly the same one, so we recommend reading the previous subchapter first.

The second phase begins with the *ChunkLookupRequest* for each replication of each chunk (only the first one will matter but we do this so we can reassemble a file with chunks from different replication numbers) and then the peer that initiated this protocol sends a *GetChunkRequest* to the one that has the chunk saved, retrieving the necessary information to rebuild the file.

2.3 Reclaim Subprotocol

Similar to all the ones before, this operation is called via the TestApp indicating the final space (in megabytes) that the desired peer will have and the protocol name (***RECLAIM***).

This too has two different phases: first we pick the chunk with the biggest size (so that we only move as few chunks as possible) and then we initiate the second phase of the delete subprotocol (so that all the redirects are clean). Once this phase ends, we begin the *ChunkBackup* subprotocol in order to get a new home for this chunk. This way all the redirects will be correctly placed and we tend to minimize them, because we will try the owner, according to chord, once again and only then its successors.

This process is done simultaneously to every chunk that the peer has to get rid of in order to not exceed the new maximum size.

3. Concurrency Design

A distributed system would not be complete if it would not handle the concurrent execution of different requests. There can be identified two different modules to this

concurrent design: handling multiple peer messages received; handling the dispatching of several messages to different peers. These are used very differently in each one of them, but they both share small lifespans (even though dispatchers may take significantly more time). This makes them especially prompted to be used with a Cached Thread Pool from the *Executors.newCachedThreadPool*, that specializes in handling cases where threads are required frequently for short periods of time.

3.1 Handling Messages

TCP communication in this application heavily relies on Asynchronous Communication provided by the Java NIO.2. All peers run an instance of a *AsynchronousServerSocketChannel*, which receives an *AsynchronousChannelGroup* created specifically with a cachedThreadPool from the beginning. This assures the ability of the Channel to scale the number of messages being processed at the same time.

Messages are then accepted in a while loop, while given a *CompletionHandler*. These handlers are used as **callbacks** that are **asynchronously** called whenever a TCP connection is accepted. This channel and connection acceptance is handled in *com.feup.sdis.peer.MessageHandler.receive* After the connection is accepted, data is asynchronously read from that channel up until the message is fully restored. The **request** is then **deserialized** and properly handled, before **returning a response**. This is handled in *com.feup.sdis.peer.SerializationUtils.deserialize*.

The initiation of the communication with a given peer's server channel information, is also handled in *com.feup.sdis.peer.MessageHandler.sendMessage*, however, this will be further discussed with in the Message Dispatching topic.

3.2 Message Dispatching

As it has been mentioned before, every file is separated into several different chunks, which leads to an exchange of messages to happen for each chunk, for most protocols. Naturally this can be parallelized to achieve maximum efficiency. This relies heavily on the use of the the Cached Thread Pool mentioned above whenever it was needed to process different chunks. This thread pool use is evidenced throughout all the classes present in the package *com.feup.sdis.actions***,** both through the use of its *execute* and *submit* APIs, whenever the return result was not important, thus needing a *Runnable*, or when the respective result was vital for next operations, thus needing a *Callable* that returns a *Future.*

Whenever it came to sending a message, the *com.feup.sdis.peer.MessageHandler.sendMessage* method was called, which handled setting up the connection, sending a message and receiving the response in it. This method relied heavily on the use of *Future* API option present within *AsynchronousSocketChannel*, rather than the *CompletionHandler* API, which assures asynchronous communication .

4. Scalability

4.1 Chord

To tackle the problem of scalability of the system, we opted by using **Chord** peer-to-peer lookup protocol (package *com.feup.sdis.chord*). This decision allows us to **avoid having single-point failures** , that is, the failure of a peer, whichever may it be, does not cause the disruption of the entire system. Also, since the lookup algorithm is distributed among all peers on the backup-system, the **load** of said algorithm is **not focused** on any particular peer, as would happened with the “server” on the centralized approach.

By having Chord we also remove the network “address dependency” of the stored chunks. That is, to lookup a chunk, a peer only needs to know the Chord key corresponding to that file, since the key is always bound to the chord peer “responsible” for the chunk, as opposed to storing the concrete address of said peer. This allows for changes on the ring’s topology, without having to change information that is stored in the backup system itself, which contributes to the scalability of the system.

Our implementation is, nearly, **decoupled** from the rest of the backup system, which only interfaces with Chord through the *lookup(chunkID, repDegree)* and *getSuccessor()* methods.

4.1.1 Chord peer key specification

Our implementation followed the paper by *Stoica et al.*[1] and used a ***m* value of 128**, which corresponds to size-128 finger tables. This value is dictated by the utilized hash function, which, in our case, was provided by the ***UUID*** class of JavaSE. We chose to use this class because it provides a solid way for representing the peer’s identifiers and of comparing them. Also, the **arithmetic operations** that need to be done to the identifiers in order to implement the chord algorithm (addition and modulos) can be easily achieved by converting the UUIDs instances of JavaSE’s ***BigInteger*** class, and back (methods *initKeyLookupSteps, compareDistanceToKey, convertToBigInteger and convertFromBigInteger* of Chord). A peer’s **Chord identifier** is obtained by creating a UUID from the peerID string that is passed upon creation.

4.1.2 Chord initialization

In order to **create** the Chord instance, we provide two possibilities. The first is done by passing, as a constructor argument, the *SocketAddress* address of the current peer, which will create a new Chord ring. The second way is to **join** an existing Chord ring by bootstrapping through a Socket Address that contains the IP address and port of another peer.

4.1.3 Peer communication

The pseudo-code that is presented in the referenced paper [1] shows an interaction between peers using RPC. However, we decided to utilize the constructions we created for sending and receiving **messages** using **TCP** in a **scalable** manner (class *MessageHandler*). For example, when peer A needs to execute *findSuccessor* on peer B, he sends it (using *MessageListenter.sendMessage*) a *FindSuccessorRequest* that, when received, calls

findSuccessor on B and returns a *FindSuccessorResponse.* For this purpose we created packages containing the needed Request/Response classes needed to execute every inter-peer execution: *com.feup.sdis.messages.requests.chord* and *com.feup.sdis.messages.responses.chord*

4.1.3 Periodic methods

The **periodic functions** described in the paper (*fixFingers*, *checkPredecessor*, and *stabilize*) are run at a fixed rate, without using *Thread.sleep* calls (to **avoid blocking** and achieve maximum **scalability)** by using a *ScheduledThreadPoolExecutor.* These threads are started one after the other with a small delay between them in order to reduce the number of concurrent executions. The period between executions of these methods is controlled by three flags that exist in the *Chord* class (*STABILIZE_INTERVAL_MS*, *FIX_FINGERS_INTERVAL_MS* and *CHECK_PREDECESSOR_INTERVAL_MS*).

To make our solution compatible with the concurrent design of the rest of the system, we make use of JavaSE's ***AtomicReference*** and ***AtomicReferenceArray****,* these classes ensure consistency in the **concurrent accesses** to the finger table, successor list and predecessor/self references.

4.1.4 Debugging artifices

To streamline the development of the system and to simplify the understanding of the inner-workings of the system, we included calls to the method *normalizeToSize* upon chord creation and key lookups that "scale down" (using modulus operations) the specified key and table size, which allowed us to work with smaller and more understandable values while maintaining the *UUID* interface. The referred size is controlled by the *FINGER_TABLE_SIZE* value on the *Chord* class. By using the modulus operator to achieve small sizes, there is a **significantly increased risk of having collisions between peer IDs**, however, since this functionality was only introduced for testing purposes we chose do not address it directly but to be **aware of it while testing the system** (an example of this can be seen in Appendix A). The likelihood of these collisions decreases exponentially with increasing table sizes (according to the paper should be 128).

4.2 Java NIO

Concurrent execution has a high impact when it comes to truly scaling a system and therefore could not be forgotten. The use of Java NIO, more specifically, Asynchronous I/O was, therefore, one of the pillars to truly scale the system. How this was used has already been explained with extreme detail in section 3, where a summary of the impact of Asynchronous I/O had on the concurrency design of our system and consequently its scaling capability.

5. Fault-tolerance

5.1 Chunk Replication

In order to avoid having a single point of failure, we implemented the ability of replicating the chunks of a file with a maximum degree of 10 (preventing malicious users overloading the system). This way, even if a certain peer that contained a chunk shuts down unexpectedly, we can still restore the file if we so desired (and if the replication degree is sufficient).

The replication degree became an important part of the information and an integral part of the key used by the chord, so that two chunks of the same file would be scattered/spread through different peers.

In the unfortunate event that two keys of the same chunk collide, one of them is redirected to the successor (B) and the redirect reference is saved by the first peer (A). In this way, even if the peer A fails, B would become the owner of that key (according to chord logistics) and we maintain a consistent state.

5.2 Chord fault tolerance

Another aspect that contributes to the system's fault tolerance is the use of Chord as described in **section 4.1**. This avoids the single-point failures that would come with having every peer be dependant on a central server, as it **shares the algorithm of key/chunk lookup among all peers**.

If a peer fails, the system will only be affected for a short amount of time, since other peers will **balance their respective finger tables to account for the new ring topology,** which is mainly achieved through the periodic functions (*fixFingers*, *stabilize* and *checkPredecessor*). As described in the paper, even during this brief balancing of the fingertables, lookups will still succeed but may have worse performance than with a stabilized ring. In this project, we considered to types of peer absence, one that is transient (accidental) and one that is considered permanent (peer shutdown); the differences between these situations and of the way we handle them are explained in detail in **section 5.3.**

In addition to the inherent fault tolerance that Chord gives to the system, we also sought to improve our chord implementation resistance to peer failures. For that we included the **list of successors** that is mentioned in point 3 of the "Dynamic Operations and Failures" section of the paper [1]. To employ these changes we included an extension to the stabilize periodic method that queries the peer's successor for it's successor-list (using *ReconcileSuccessorListRequest*), from which the current peer will update it's own list. In case of an unresponsive successor during Chord operations, the system will try the next successor in the list. The peers in the list are also considered when searching for the closest preceding node.

5.3 Peer Shutdown

We divided a peer failure into two cases: when someone voluntarily shuts down a peer and when something unforeseen happens and we have to deal with its consequences (like internet failure).

Our philosophy was that in the first case the peer is prone to not coming back in the foreseeable future, because its owner intended for the shutdown to happen; in case of internet failure we assume that the peer will be back and running in no time so we must try to keep the system's consistency meanwhile.

These different approaches will lead to two ways of dealing with the occurrence of a shutdown.

5.3.1 Voluntary Shutdown

A peer can be shutdown voluntarily either with a **SIGINT**/**SIGTERM** signal or with the TestApp using the operation **SHUTDOWN** without any arguments. This option assumes that the peer will be gone for an **indefinite** time (maybe forever), which means that the system should replicate all the chunks that he has saved before it leaves.

To execute this in an orderly manner, we execute a Reclaim protocol with final space 0, making the peer **dispose of all the chunks** that he previously owned and backing them up in the new corresponding peers (ShutdownHandler.java, line 15). After all chunks have been deleted/transferred, the program will **shutdown the Chord** ring (by stopping the periodic threads), closing the message handler and **deleting persistent data**.

5.3.2 Involuntary Shutdown

If there is a network error, we assume that the peer will be back soon and that it didn't lose the files, so we mustn't replicate the chunks that he owned.

This assumption lead us to the conclusion that there is only one operation that we must have in account while the peer is down: the deletion of a file of which the peer currently has chunks (all the other operations can be performed without the presence of this peer).

We decided to implement a queue of failed delete operations that, at a periodically fixed rate of time, will pop the first element and try to perform it. If this protocol fails it goes back to the end of the queue. This routine is performed a maximum of amount times defined in *Constants.MAX_REQUEST_RETRIES*.

5.4 Peer Startup

This routine isn't really to increase fault-tolerance, but we decided to mention it here because it also helps to maintain the system's consistency.

When a new peer joins the system, after the chord is stabilized, he will trade a message with its successor and its predecessor. It will receive from both of them information regarding their redirects, which is relevant to make sure all chunks in the system are tracked.

Furthermore, it will ask its successor regarding the chunks whose keys are now attributed to this peer. This is done by the message *TransferChunksRequest*, responsible for both sending the chunks to transfer as well as the redirects.

These will then be transferred to this new peer (if he does have enough space for them), and redirects will be updated to point towards this new peer. With this, this chord node's information is now consistent with the requests it will get from now on. For each chunk that is actually taken from the other peer, a new *TakeChunkRequest* is sent, which takes the chunk body from the peer as well as fix the redirects.

6. References

[1] - Stoica, I.; et al. (2001). "Chord: A scalable peer-to-peer lookup service for internet applications". ACM SIGCOMM Computer Communication Review. 31 (4): 149.

Appendix A

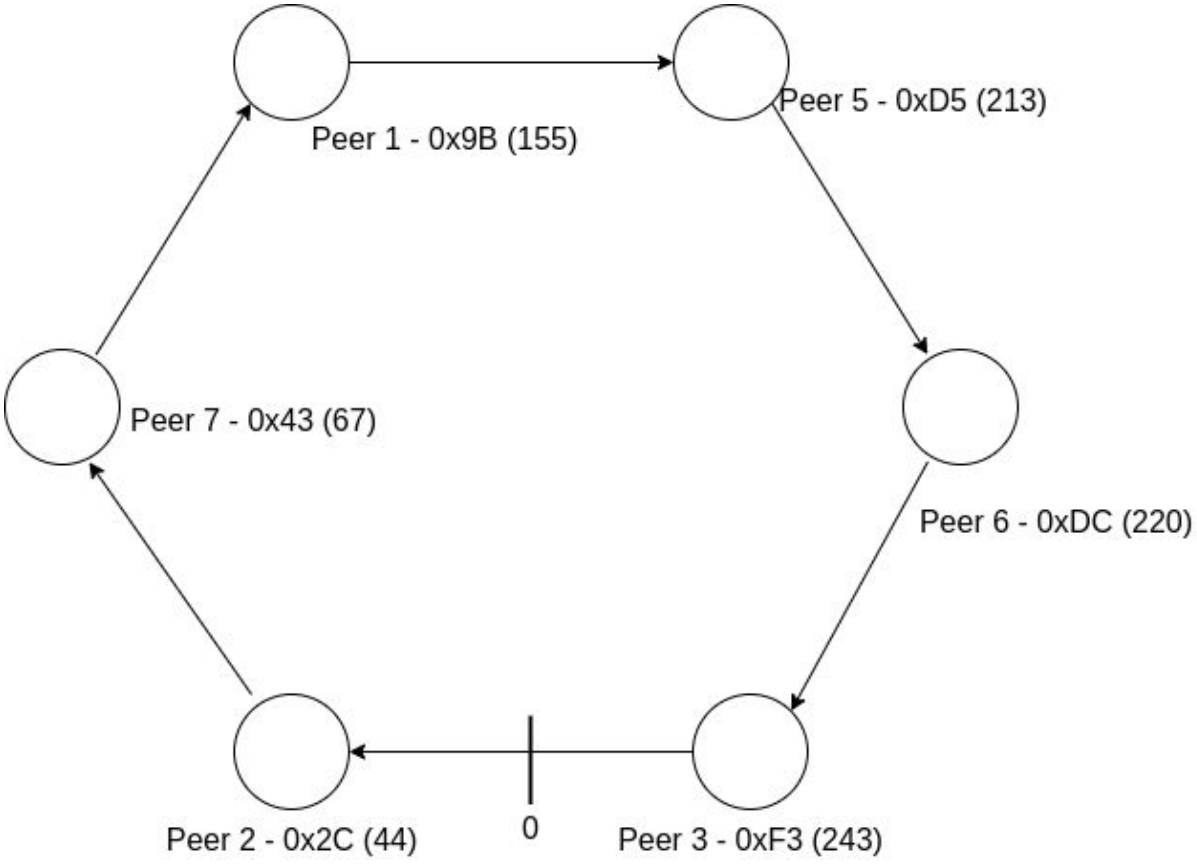

Fig 1.- This is the topology of a chord ring with six peers. In this example the table size is reduced to 8, as such, the keys are presented modulus 256. The caption of each node contains the peerID argument, the corresponding hex value after the hash and modulus and it's decimal value. Peer 4 was skipped because the debugging modulus operation for table reduction made its key collide with another key.