U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

May 2020

Distributed Systems

Distributed Backup Service for the Internet

Alexandre Miguel de Araújo Carqueja 201705049

Ana Margarida Ruivo Loureiro 201705749

João Miguel Ribeiro de Castro Silva Martins 201707311

Leonor Martins de Sousa 201705377

3MIEIC06 - Group 21

Prof. Pedro Alexandre Guimarães Lobo Ferreira Souto

Prof. Hélder Fernandes Castro

Summary

Overview

The project implements the main operations of the backup service as specified in the first assignment. Those are: **backup**, **delete**, **restore,** and **reclaim**. We implemented features that brought concurrency to our design, as well as security, scalability, and fault-tolerance. In this assignment, we also implemented thread-pools to provide concurrency, secure sockets to improve security, Chord, and asynchronous file channels to improve scalability, Chord's fault-tolerance features, and chunk replication to improve fault-tolerance.

Protocols

The base communication protocol used in the project was TCP, except for the communication between the client application (TestApp) and a peer, which reused the implementation from the previous assignment, that was done using RMI.

The exchange of information itself was done using *ObjectStreams*, which allowed us to have a more high-level interface of the messages exchanged and eased processing. Nevertheless, each subprotocol has its set of messages, which are described below.

Backup

Since we implemented Chord and are using TCP to communicate between peers, we had to rethink the logic of the backup protocol.

Having Chord functionality in mind, the main idea is that each chunk we wish to backup will be assigned to a given peer in the ring and it's responsible for storing it. Then there are particular cases that we also explain how we deal with, such as the allocated peer being the initiator, or the allocated peer doesn't have enough free space to store a chunk, etc.

To make some concepts clearer, we identify the peer that starts a protocol as the **initiator**, the peer responsible for a given chunk as the **ideal,** and the peer that will store a chunk in case that the **ideal** can't as the **ideal's successor**. Summing it up, the **initiator** starts a backup, send a message to the **ideal** peer for that chunk. Then, this one either stores it or sends it over to the first **successor** that is able to store it.

Now, shortly this is how we deal with particular situations:

- Since we implemented chunk replication, the desired replication degree is bigger than 1, the message is sent to the **ideal** and this one is responsible to send the chunk to its **successors** until the replication degree is met and keep references of those who stored the chunk. In the end, answers back to the **initiator**, report the success in backing up the chunk and the replication degree achieved.
- Whether the **ideal** doesn't have enough space or is the initiator, the behavior is to send it to the first **successor** available and store a reference of the one who stores it.

The messages used in this protocol are:

- ➔ *PUTCHUNK*:
  - *type*: “PUTCHUNK” - identifies the message and lets the receiver know what contents of the message are expected.
  - *senderID*: peer ID or -1 - if the sender is the **initiator** this field contains its ID or if it’s the **ideal**, -1.
  - *fileID*: the ID of the file whose the chunk belongs.
  - *chunkNo*: [0;n-1] - n is the total number of chunks the file is comprised of
  - *repDeg*: >0 - the desired replication degree (the backup may fail if the value is higher than the number of peers active)
  - *nChunks*: >0 - the total of chunks the file is comprised of
  - *body*: the chunk’s data
- ➔ *STORED*:
  - *type*: “STORED”
  - *senderID*: >0 or -1 - when the **ideal** sends a *STORED* to the **initiator**, this contains the replication degree achieved in this backup, when is the **successor** answering to the **ideal**, the value is -1.
  - *fileID*
  - *chunkNo*
  - *origin*: *InetSocketAddress* - is the sender’s address information, like a sort of a “business card”, so if the receiver needs to contact the peer directly it may, instead of going through the lookup first. It’s used in a few cases where having this information is more useful and the specific peer wanted is known.
- ➔ *DECLINED:*
  - *senderID*: -1 - this specific message is used only to notify the **ideal** that a **successor** that was contacted to store a chunk wasn’t able to.
  - *fileID*
  - *chunkNo*

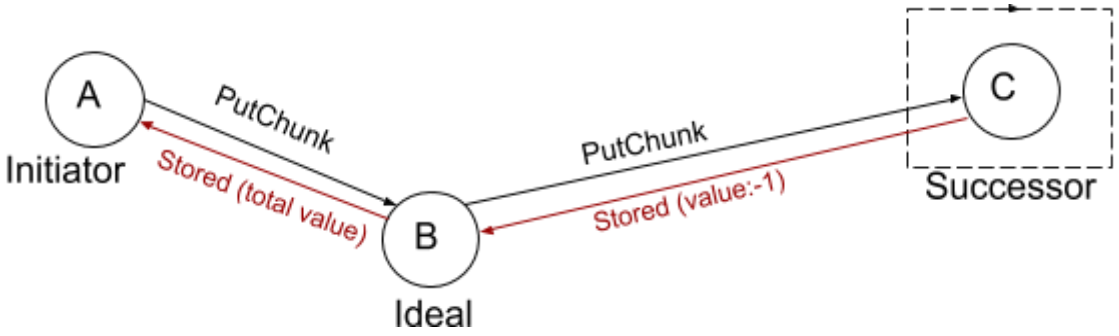

Fig1. A scenario where initiator sends a backup request with a replication degree bigger than 1 to the ideal; ideal forwards to successors so they can store the chunk as well (the number of peers to send to are in agreement with the desired replication degree); successors answer back to the ideal with a stored message with -1; the ideal collects the stored messages from the successors and then sends back a stored message with the number of replications archived.

Delete

The **delete** protocol has the purpose of removing a file from the backup service, all of its chunks, and replicas. The goal of this protocol's usability is for any peer to be able to remove the file. Therefore, there are some limitations on what information an **initiator** has to begin the protocol. The peer only has access to the file id and not the total number of chunks the file was divided into. To allow the full delete of every chunk, and since each file has at least one chunk, the logic of this protocol is to send the delete information to the first **ideal** peer, which will cause for the peer to remove that chunk (and notify the **successors** who have replicas (based on the **reference table**) to delete them as well), and reply with the total number of chunks that that chunk's file is made of (this information is obtained back in the **backup** protocol). This way, the initiator will know for certain how many more delete messages it has to send and be able to go through with the deletion of that file.

The messages used in this protocol are:

- ➔ *DELETE:*
  - ◆ *type*: "DELETE"
  - ◆ *senderID:* peer ID or -1 - it will be the ID of the **initiator** or -1 if it's the **ideal** sending to its **successors**
  - ◆ *fileID*
  - ◆ *chunkNo*
- ➔ *NUMREPLY*:
  - ◆ *type*: "NUMREPLY"
  - ◆ *senderID*: ID of the sender of *DELETE* message
  - ◆ *fileID*
  - ◆ *chunkNo*
  - ◆ *nChunks*

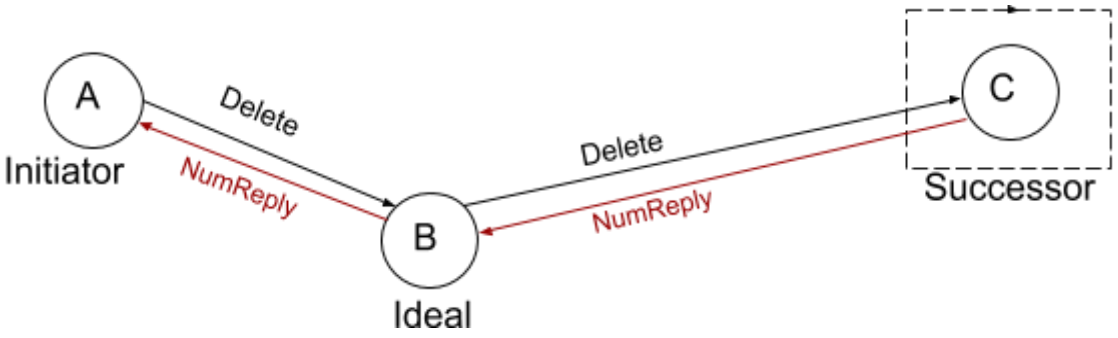

Fig2. A scenario where initiator sends the request to the ideal; ideal forwards to successors in the reference table (peers that also have the chunk); these can process the delete as well; in the first chunk of the file the initiator receives an answer with the number of chunks associated to the file it wants to restore

Restore

The **restore** is quite similar to the **delete** protocol in the way of obtaining information. Same as before, any peer can initiate this protocol on any file backed up in the service. And the way of knowing how many chunks the file is made of is the same, getting the first chunk and the reply will also contain the total number of chunks of the file.

The difference remains in the way of obtaining what we want. In the first scenario, a single message is sent to every **ideal** peer who is responsible for storing a chunk, and from there the other peers also storing are notified by it. However, we want now to receive the chunk's information and it might not be present on the **ideal**. The chunk content that is not present on the **ideal** will be present on one of its **successors**, which we have a **reference table** about. So the sequential process of the protocol will be as follows: the **initiator** requests a chunk to the **ideal** peer. if it has it, will reply with the wanted chunk. If not, the **ideal** will query the **successors** who also stored the chunk, and confirm that indeed the **successor** has definitely the chunk. Then one will answer back, and the **ideal** peer will know which one and replies to the **initiator** with the address of the **successor** who holds the chunk. Finally, the **initiator** contacts the **successor** and retrieves the chunk wanted.

The messages used in this protocol are:

- ➔ *GETCHUNK:*
  - ◆ *type*: "GETCHUNK"
  - ◆ *senderID*: ID of the **initiator**
  - ◆ *fileID*
  - ◆ *chunkNo*
- ➔ *CHUNK:*
  - ◆ *type*: "CHUNK"
  - ◆ *senderID*: ID of the peer sending the message
  - ◆ *fileID*
  - ◆ *chunkNo*
  - ◆ *nChunks*
  - ◆ *body*: byte[ ] - the content of the chunk
- ➔ *FORWARDGET:*

- *type*: “FORWARDGET”
- *senderID*: ID of the **ideal** sending the message to one of the **successors**
- *fileID*
- *chunkNo*

➔ *GETTOIDEAL:*

- *type*: “FORWARDGET”
- *senderID*: ID of the **successor** replying to the **ideal**
- *fileID*
- *chunkNo*
- *flag*: boolean - **true** if it has the chunk, **false** otherwise

➔ *REPLYINFOINITIATOR:*

- *type*: “REPLYINFOINITIATOR
- *senderID*: ID of the **ideal** who’s sending back the info to the **initiator**
- *fileID*
- *chunkNo*
- *address*: address of the **successor** who holds the chunk

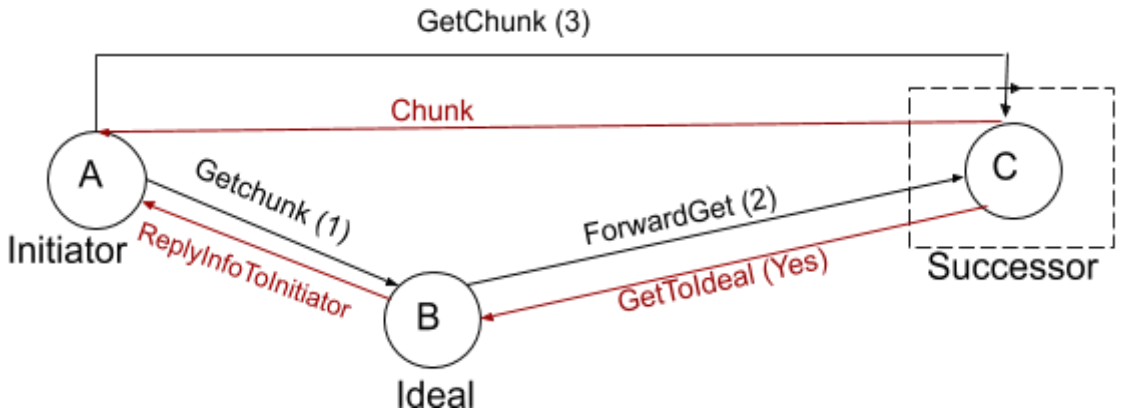

Fig3. A scenario where initiator sends the request to ideal; ideal doesn’t have the chunks so it forwards the request to successors referenced in the reference table has having the chunk; the successor answer to the ideal confirming its availability; the ideal answers to the initiator with the information related to the successor; the initiator is capable of asking directly to the successor for the chunk; the successor sends the chunk to the initiator

Reclaim

As specified in the handout, one peer must be able to control its own storage space. In case it is reduced and some chunks that have stored need to be freed we need to make sure that the replication of those chunks is assured by the rest of the peers of the service. Following the same logic as in the previous assignment, when a chunk is removed the peer removing will check the **ideal** of that chunk. If it is the same as the **initiator**, it will begin a backup for that chunk for another **successor** to store. If the **ideal** is another peer, it will send a message to that peer. Once the message is received by the **ideal** it will check if it has the removed chunk in its own storage. If it has, it will then initiate its backup. If it is not, it will

communicate with one of its **successors** to retrieve the chunk and use it to initiate the backup (because we want to have the reference stored in the **ideal** peer).

The messages used in this protocol are:

- ➔ *REMOVED:*
  - *type*: “REMOVED”
  - *senderID*: ID of the **initiator** of the reclaim protocol
  - *fileID*
  - *chunkNo*
- ➔ *SUCCGETCHUNK:*
  - *type*: “SUCCGETCHUNK”
  - *senderID*: ID of the **ideal** who is querying one of its **successors**
  - *fileID*
  - *chunkNo*
- ➔ *DECLINED:*
  - *type*: “DECLINED”
  - *senderID*: ID of the **ideal** who is replying to the **initiator** with the address of the **successor**
  - *fileID*
  - *chunkNo*
  - *origin*: address of the **successor** who has the chunk

Concurrency Design

As mentioned before, we have implemented a thread-based solution. This is done using a thread pool and we break the execution into tasks that are run separately. Some tasks are cyclic events, such as saving state or listening to the requests, therefore having a dedicated thread to them. The rest of the tasks are “on-demand”, meaning that whenever a request reaches the peer, it’s processed and it’s handled concurrently to make sure that each peer can deal with multiple requests at the same time. Because some requests are linked, for example, a request of a chunk and its receival, there needs to be some way of the threads communicating with one another. For that purpose, we make use of synchronized data structure such as the *ConcurrentHashMap*.

We have also used java.nio to deal with file input and output. We use *AsynchronousFileChannel* to write and read from every file related to the service itself (backup requested files, chunks, restored files) but the files that store the state of each peer when it’s closed correctly. *AsynchronousFileChannel* allows us to run all I/O operations in separate threads contributing to the concurrency design.

JSSE

In order to improve the security of the communication between peers, we make use of secure channels, which we implemented using *SSLSocket*.

We implemented mutual authentication and each peer alternates between the role of server or client whether it is receiving or sending a message. Thus each peer has a folder with the three keys available: *server.keys*, *client.keys*, and the *truststore*. These *keystores* and *truststore* used are the ones made available in the Lab5 (the contents of each of the files are encrypted using the password 123456).
The cipher suits were chosen based on its compatibility with 256-bit keys with RSA encryption: TLS_DHE_RSA_WITH_AES_256_CBC_SHA and TLS_RSA_WITH_AES_256_CBC_SHA.

In order to run each peer we added the following properties to the java command that initiates the peers in the command line:

```
-Djavax.net.ssl.keyStore=keys/server.keys
-Djavax.net.ssl.keyStorePassword=123456 -Djavax.net.ssl.keyStore=keys/client.keys
-Djavax.net.ssl.keyStorePassword=123456 -Djavax.net.ssl.trustStore=keys/truststore
-Djavax.net.ssl.trustStorePassword=123456
```

Scalability

In our design, we chose to implement *Chord* and make it totally distributed. This is already a way to improve scalability as we allow our network to grow easily and not cause disruption to the peers already in the network. The usage of threads also makes it easier to have more requests being processed at the same time, but some aspects could be improved, related to I/O operations. To help mitigate this issue we made use of the Java NIO API to make some tasks asynchronous. In particular, we used *AsynchronousFileChannel* for the writing and reading of files in the Peers.

Fault-tolerance

Since failures are inevitable and eventually occur, there is the need of having some fail-safes for when it happens and the service integrity isn't compromised. We analyzed two aspects that could be improved and have implemented solutions to mitigate them.

The first, regarding the failure of a peer, since we are using Chord to sustain our distributed application, there needs to be a way of preventing the ring to be broken when a node fails. Using the Chord's paper[1] as inspiration, we implemented the fault-tolerance feature mentioned in it, and add a **successor**'s list to each node, alongside the finger table, which will aid the recovery of the ring when a node fails. More details about this implementation can be found in Chapter IV, Section E, Subheading 3 of the paper.

The second, also regarding the failure of a peer, but in this case the availability of the chunks data. If we had chunk's data only stored in one peer at the same time, we would have an entire file compromised in case of one peer failure, so each peer could be a single point of failure of a file's backup, which we could not tolerate. For that reason, we reuse the same logic as the first assignment and replicate the chunks throughout more peers, as many as requested by the client (although limited by the number of peers currently available in the network). In our particular scenario, each **ideal** peer is responsible for assuring the desired replication degree of a chunk, requesting its immediate successors to store them and collect

the **references** of those who do. This way this information is centralized in the **ideal** peer and accessible to any peer who makes a request for that chunk. In the end, the **ideal** replies to the **initiator** with the replication degree achieved.

In addition to this, and taking advantage of Chord fault tolerance features we implemented a backup mechanism to the reference tables of each peer. Since when one peers leaves/fails the other peers who store the chunks are still available, it makes no sense to lose this information. So, we back up each chunk references tables in its successor to be accessed in case of failure of peer exit. Periodically the peer sends the tables to the successor to minimize the amount of information lost. When a peer leaves voluntarily it anticipates this backup and it makes sure that the successor has the most recent version.

References

[1] - Stoica, I., Morris, R., Liben-Nowell, D., R. Karger, D., Kaashoek, M., Dabek, F. and Balakrishnan, H., 2001. *Chord: A Scalable Peer-To-Peer Lookup Protocol For Internet Applications*. [online] Dl.acm.org. Available at: <https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407?download=true> [Accessed 21 May 2020].