SDIS P1: Serverless Distributed Backup Service

Eduardo Ribeiro - up201705421
Luís Ramos - up201706253
Miguel Pinto - up201706156

14th of April, 2020

1 Assumptions

In this section, it will be explained and detailed some assumptions that were made while developing this project.

Firstly, a situation that could generate some problems was cases when two peers request backup for the same file (after there was/wasn't a delete for it).

This scenario was thought about, but not addressed by the protocol, given the fact that we should assume that peers run in different machines, therefore the file system is different to every peer. Because the combination (filepath + fileModificationDate) was used to generate the file ID, if the path of the file and modification date was the same, the same file ID would be generated, so this situation could occur even with isolated file systems for each peer. However, this scenario is highly unlikely to happen, and the professor confirmed that we could discard this possibility.

Another situation that caught the group's attention was when after a reclaim occurred, and the perceived replication degree of a chunk got lower than the desired replication degree, another peer had to initiate the backup subprotocol for that chunk, as described in the project's specification. If there was another peer that had joined the system after the original backup had occurred (therefore, he did not know who had stored the chunks), then, in the case he would be prompted to store the chunks after the space reclaiming, he would assume that the backup-sending peer does not store the chunks. To fix this, the backup-sending peer sends a STORED message for the chunks he is backing up.

Another aspect we should refer in this section is the naming of the protocols the peers can implement: throughout this report, the regular protocol, with no enhancements, will be called the protocol 1.0. The protocol with the enhancements will be called protocol 1.1.

2 Explanation of the implemented enhancements

2.1 Backup Enhancement

For the enhancement of the backup subprotocol, we needed to find a solution that avoided the rapid depletion the backup space on the peers, because in the base protocol every peer that has enough space will save that chunk. Therefore, the enhancement needed to make sure that a peer would not store a chunk unnecessarily, given the wanted replication degree for a file. The enhancement also needed to be fully compatible with peers running the protocol 1.0.

The implemented backup protocol enhancement can be split in two different sections. The first part is a random backoff time (between 0 and 600ms) that enables the peer to receive all incoming messages and process them. This is done in order to give the peer a chance to register all peers that have stored the chunk, whose back up was requested, in a table (*perceivedReplicationTable*) that keeps track of all peers that have already backed up a chunk. After the backoff time is over, the peer checks to see if the perceived replication degree for the chunk is equal or higher than the desired replication degree for the file, registered in another table (*desiredReplicationTable*), if this condition is true then the peer only needs to check if the chunk is already stored and if so send the STORED message, otherwise the peer saves the chunk (if it can) and immediately sends the STORED message afterwards (if it was able to save the chunk).

This solution was found to be very effective because the random backoff enables some peers that get a lower backoff time to start saving the chunk and then proceed to send the STORED message, this way peers that end the backoff time later have a higher chance of already having a replication degree higher or equal to the desired one, and so, because of this, most of the times, the replication degree of a chunk and its desired replication end up being the exact same. It has a high success rate.

2.2 Restore Enhancement

For the enhancement of the restore subprotocol, we needed to find a way to transfer the chunk only between the two peers, instead of being through a multicast channel. For this, the use of a TCP connection was encouraged.

The implemented restore protocol enhancement was done using TCP as proposed in the project description. The model defined for the use of TCP was the server-client but in this implementation the server only handles one request and then is shutdown (therefore we have one "server"/connection per chunk transfer). The server side was done in the peer that is sending the chunk and the client side was done in the initiator peer. The protocol remained the same in the sense that the initiator sends a GETCHUNK message and one peer sends a CHUNK message (if it has the wanted chunk) after a random backoff time, however this last one is a bit modified. A new header line was introduced to this message, in order to pass on the information regarding the port used by the TCP socket of the server; the body of the message was removed because the objective of the enhancement is to not have to send the chunk to every peer. The new syntax for the CHUNK message is:

```
<Version> CHUNK <SenderId> <FileId> <ChunkNo> <CRLF>
<PortNumber> <CRLF><CRLF>
```

This allowed the other peers to have the knowledge that the chunk was already sent by the peer that sent the CHUNK message (and therefore avoid multiple CHUNK messages, like in protocol 1.0), and allowed the connection between the initiator and the sending peer to be established. After the TCP connection is made, the information transmitted in the TCP socket is only the chunk content, because for each chunk there is a new TCP connection being established, allowing multiple chunks to be received at the same time from possibly multiple peers.

2.3 Delete Enhancement

For the enhancement of the delete subprotocol, the objective was to find a way that peers that did not receive the DELETE messages (because they were offline) would still delete their chunks and reclaim that space.

The delete enhancement that is present in our implementation was done with the help of two new types of messages: the DELETED message and the GREETINGS message (because these messages are not from the base protocol, any peer running protocol 1.0 is going to ignore them). The new message types have the following syntax:

```
<Version> DELETED <SenderId> <FileId> <CRLF><CRLF>

<Version> GREETINGS <SenderId> <CRLF><CRLF>
```

The explanation of the GREETINGS message is straightforward: every time a peer boots up (with the protocol 1.1), it will send a GREETINGS message through multicast, for every peer to acknowledge.

In a regular peer that is implementing no enhancements, upon receiving a DELETE message indicating it to delete all kept chunks of a specific file, the peer will not responding with any message; this way, the initiator peer does not know if the other really deleted its chunks or not. Now, a peer running the enhanced protocol will respond with a DELETED message through the MC channel (after a random backoff of 0-400ms), indicating that it indeed removed the chunks related to that file.

The initiator peer, that knows which peers have the chunks of the file, will now keep specific data structures to store information about the peers that have at least 1 of those chunks, but have not responded with the DELETED message. The information for each peer is kept in instances of the class *FileDeleter*, which keeps all the DELETE messages that the peer was supposed to respond to in the *fileToDeletes* hashmap. It is assumed that these peers did not delete the chunks, which can happen for a number of reasons, one of them being their unavailability. A hashmap denominated *fileDeletionList*, connecting each *FileDeleter* to a peer, is kept in the *ChunkManager*.

In order for the peers to delete their chunks, even if they are not running at the time the initiator peer sends a DELETE message for that file, each time the initiator peer receives a message, and its sender has a *FileDeleter* instance associated to them in the *fileDeletionList* hashmap, all the DELETE messages contained its *FileDeleter* are re-sent to the peer, knowing this time it is online and ready to receive messages.

Lastly, and in order to avoid conflicts between the use of different subprotocols, if a backup request for a specific file is made (it can be made by any peer), all DELETE messages contained in all *FileDeleters* relative to that file are erased.

2.4 Ideas for Service Enhancement

We opted to include this section in the report to mention and explain some ideas that the group had for the enhancements, but were later discarded and not implemented, either because of the available time for the project or because we came to the conclusion that the possible drawbacks of the idea weighted more than the advantages.

The main idea that we ended up not implementing regarded the backup subprotocol enhancement. A change to the enhancement that could ensure that the perceived replication degree of each chunk would be equal to the desired degree for the file would be: after the peer sends the STORED message, it could have then proceeded to verify if the perceived replication was higher than the desired one. If that was the case, it meant that some peer(s) was/were storing the chunks unnecessarily, and could free that space in their storage. The peer would then wait a random backoff, let's say between 0-400ms, in order to hear any REMOVED messages from other peers regarding those chunks. If it received any, it meant that another peer already deleted the chunks. If the perceived degree was still higher than the desired one, the peer would then delete those chunks (or a portion of them), and send a REMOVED message for each one of them, to warn the other peers.

This solution is rather complex and has some failing points. For example, even with the backoff before deleting the chunks and sending the REMOVED message, two (or more) peers can delete the chunks without knowing that the others did it, which can lead to the perceived degree of the chunks being lower than the wanted one. Moreover, because our solution had good results, we decided that there was no need to implement this strategy into the system.

3 Concurrency Design

In this section, it will be explained in full detail the measures taken to ensure as much concurrency and scalability, as well as keeping the integrity and well functioning of the system.

3.1 ExecutorService, ThreadPoolExecutor and thread pools

For concurrency, the classes *ExecutorService* was used frequently. The class allowed to created a thread pool, with a maximum number of threads. If a task was dispatched to the thread pool, it would be assigned to one of its idle threads (if any; if not, it would wait for one to be idle). Using this class, we can immediately execute a task in a separate thread, or schedule it for the future (with a timeout), or even schedule it to be done at a fixed rate. This solution is more scalable than manually creating threads, because their creation and termination does have some overhead. The class *ScheduledThreadPoolExecutor* is also used in some cases, but it essentially behaves the same. The initialization code for these classes are:

```
ExecutorService service = Executors.newFixedThreadPool(nThreads);

ScheduledThreadPoolExecutor executor = new
    ScheduledThreadPoolExecutor(numberOfThreads);
```

3.2 General structure of the program

Each peer/instance of the program can be divided into parts. Each peer needs to receive incoming demands from a client (through RMI, in our case) and launch the corresponding subprotocol instance, with the right arguments, but it also needs to receive messages comming from other peers, through the 3 different multicast channels (MC, MDB and MDR) and process them.

For receiving messages from the multicast channels, a new class *Receiver-Thread* was created. It has as its main attributes a multicast socket to receive the messages from, a thread pool, and a *MessageHandler*. This last class was also created by us, and upon receiving a message, checks its type and calls the corresponding protocol method for processing it.

The job of the *ReceiverThread* is to always try to receive new messages from the multicast socket, and when that does happen, it dispatches it to the thread pool to be processed, which then calls the *MessageHandler* for the message to be correctly processed. The *ReceiverThread* class implements the *Runnable* interface, so it can be run in a separate thread.

This brief introduction was given to better explain the general structure and main functioning of the program. When a peer is initiated, it creates 3 instances of the *ReceiverThread* class: one for the MC channel, one for the MDB channel and another for the MDR channel. This way, there is one thread pool per channel, and every message received by another peer is correctly processed. The peer also initiates a fourth thread pool, to manage incoming requests by a client (made through RMI as previously mentioned).

3.3 Data integrity and concurrent data structures used

Because there are multiple threads and thread pools being used, and multiple subprotocol instances being launched, there is a high probability that the peer's data structures and attributes are going to be accessed simultaneously by those threads. Therefore, it is of major importance to guarantee that the data of the peer is kept consistent, being accessed and modified in a controlled way throughout the different threads, using appropriated data structures.

The main classes that contain data and information about the peer are the *ChunkManager* and the *FileManager*. They keep information about the chunks stored, the backed up files, the size of the chunks, desired and perceived replications, among other things. Most of this information is stored using the *ConcurrentHashMap* data structure, that allows to map a certain key to a value, while supporting fully concurrent retrievals and high expected concurrency for updates. All operations on objects of this class are thread-safe. Update operations on a *ConcurrentHashMap* generally do not block the whole map, so its performance is quite good. The system also uses *ConcurrentSkipListSet* sometimes, being an implementation of a thread-safe set: insertion, removal, and access operations are safely executed concurrently by multiple threads.

Finally, there are some variables and attributes, like the peer's *availableStorageSpace* and *maximumStorageSpace* (stored in the *FileManager*), that do need to be manually synchronized, so that operations on them can be thread-safe. That was done using synchronized blocks, each time those variables are accessed. The look used was the instance of the *FileManager*, and since the *FileManager* is shared throughout the different threads, it ensured that only one thread at a time could access those variables. Moreover, the method that saves the peer's information/metadata to files is also synchronized, to ensure consistency in the stored data.

3.4 Elimination of sleep calls and blocking calls

Three of the four subprotocols require the peers to wait a given amount of time (0 to 400 miliseconds), before sending a message to the multicast channels, as to, for example, avoid flooding the message space needlessly. As such, as a first implementation, the group decided to use *thread.sleep()* calls to generate such waiting times. However, these calls can lead to a large number of co-existing threads, each of which requires some resources, therefore limiting the scalability of the design and proving to be a sub-optimal solution. As an improvement, all of the *thread.sleep()* calls were substituted by *schedule()* calls from the *java.util.concurrent.ScheduledThreadPoolExecutor* class. In section 3.1, 4 thread pools were mentioned (3 for receiving + 1 for initiating protocols upon client requests). There is a fifth thread pool: each protocol has an instance of this class, that allows scheduling a "timeout" handler, without using any thread before the timeout expires.

As a final improvement, all of the remaining blocking calls that would be accessed by multiple threads, such as in file reading and writing, were removed to achieve ultimate scalability. More specifically, chunk reading/writing for the backup and restore subprotocols and file writing for the restore subprotocol started using the *java.nio.channels.AsynchronousFileChannel* class. This class allows concurrent access to files and spawns a thread for such operation, that can be dealt with in different ways: sometimes callback functions were used to execute a given task as soon as the file reading/writing was finished; other times, the operation returned a *Future* object, that allowed to execute other operations before requesting the result of said *Future* instance when needed, therefore taking full advantage of its asynchronous capabilities.

3.5 Summary of the main classes related to concurrency

To finalize this section, we will now do a quick rundown of the main classes that the group developed that are directly connected with the project's concurrency design.

peer.Protocol - Abstract class that contains the "skeleton" of the protocol, that is, it defines all methods that concrete implementations of the protocol should have. It also stores and initializes some fields that are common to all protocols.

peer.Protocol1 - Subclass of the *peer.Protocol* abstract class. Represents the protocol 1.0, that is, the base protocol with no enhancements.

peer.Protocol2 - Subclass of the *peer.Protocol* abstract class. Represents the protocol 1.1, that is, the protocol with the backup, delete and restore enhancements.

peer.ChunkManager - Class that contains information about the chunks stored, like their perceived replication degree, the desired replication degree, information for deleting and restoring files, etc. Uses *ConcurrentHashMap* and *ConcurrentSkipListSet* to store and modify this information.

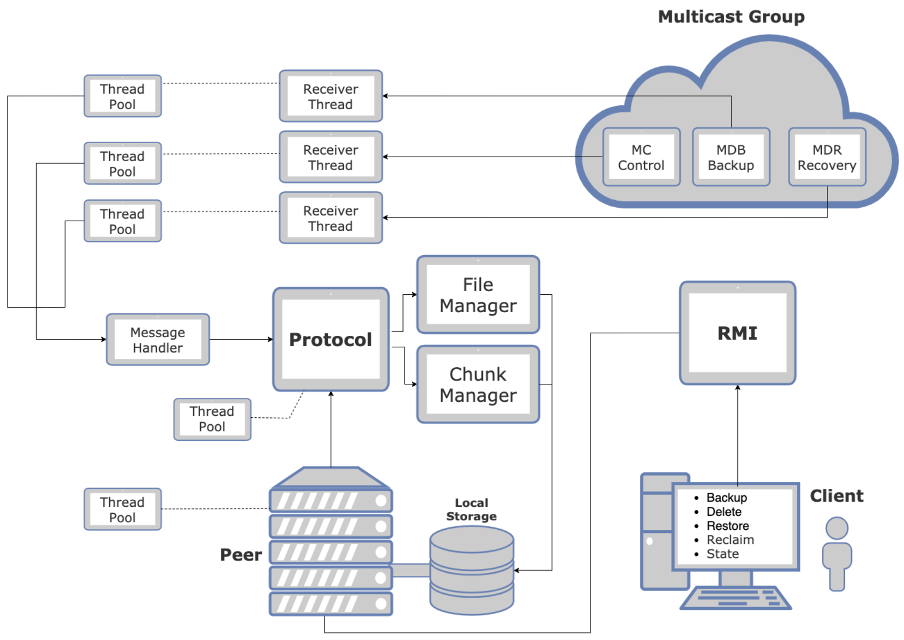
peer.FileManager - Class that contains information about the files (and chunks) stored, like the association between files and chunks, the hashes of backed up files, etc. Uses *ConcurrentHashMap* and *ConcurrentSkipListSet* to store and modify this information. Also contains the *availableStorageSpace* and *maximumStorageSpace*.

peer.MessageHandler - Its purpose is to receive a datagram packet coming from another peer, parse it, create a new message (instance of *peer.Message*) and call the appropriate protocol method for that message, given its type. It is done in its *process()* method.

peer.ReceiverThread - Class that represents a thread (and can be run like one) that is always listening to a specific multicast channel and receiving messages from it. Upon receiving a datagram packet, it dispatches it to its thread pool, which then calls the *process()* from the *peer.MessageHandler*.

peer.Peer - The class that implements the RMI remote interface, and establishes a connection with the client application. Also has a thread pool. Upon receiving client requests through RMI, it dispatches them to the thread pool, which calls the appropriate method of the peer's protocol.

To better represent the classes and entities that were mentioned above, we include this diagram to explain the structure of our code and program.



Multicast Group
MC Control
MDB Backup
MDR Recovery
Receiver Thread
Receiver Thread
Receiver Thread
Thread Pool
Thread Pool
Thread Pool
Message Handler
Protocol
File Manager
Chunk Manager
Thread Pool
RMI
Thread Pool
Peer
Local Storage
Backup
Delete
Restore
Reclaim
State
Client