Distributed Backup Service

Projeto RCOM - 2019/20 - MIEIC

Class 1

Practical Classes: Pedro Alexandre Guimarães Lobo Ferreira do Souto

Authors

André Rocha, up201706462 (up201706462@fe.up.pt)

Mário Mesquita, up201705723 (up201705723@fe.up.pt)

Porto, 14 de Abril de 2020

1. Concurrency Design

During this section, it will be described the concurrency architecture developed for the Backup Service. It is important to notice this architecture stands for both the original version of the protocol as well as its enhanced version.

All three channels use a similar structure in order to process a received message: a message is received, it is extracted from its *DatagramPacket*, and then it is properly parsed and analyzed for its type to then be processed. This meant this could be generalized to the point of having an abstract class (called *Receiver*) to unify this process as well as single-handedly control with a single thread-pool the processing of all messages coming to all channels.

Thus, three classes extending this *Receiver* were created in order to properly determine which messages were attributed to each channel. It was also their responsibility to process those messages and act accordingly to their needs.

Furthermore, this receiver's thread pool is using a *CachedThreadPool* implementation from the *Executors* class. This thread pool is especially useful in this case since it creates new threads as they are needed, reusing previously created ones after they are available. Such an implementation thrives in an environment where the program will execute several short-lived tasks, which the processing of this service's messages can be considered.

Moreover, it is also important to mention the concurrency system implemented for the processing of RMI messages, that are called by the Testing Client Application. Messages are once again quite similar and this could be generalized to be processed by a single instance. In this case, messages had very different lifespans, since some of them (such as the Backup), required the Peer to periodically check for changes in the information, in order to check if it was necessary to resend the message to try and get more peers to process. This lead to the use of a ScheduledThreadPool, which allows the scheduling of a given task at a fixed or variable time period.

The use of this thread pool allowed for several concurrent requests to be made to a single Peer, and for them to be processed in a timely manner.

2. Backup Enhancement

In this section, the enhancement to the Backup subprotocol is described. The initial version of this subprotocol caused two major problems: may deplete the backup space quickly; too much activity on the nodes once that space is full.

Firstly, it was crucial to determine what exactly caused this depletion of the space allocated to backing up files. It is rather natural to conclude that this could be due to the fact that the desired replication degree of each chunk was not taken in account to limit the number of peers who stored that chunk, resulting in, unnecessarily, all peers who received the PUTCHUNK message storing the chunk, with a replication degree far above the desired one.

Hence, it was added a simple condition to the PUTCHUNK message to check if enough STORED messages had been received by this peer from other peers, in order to prevent storing the chunk in excess.

However, this did not completely solve the issue, since there would still be some peers who would be processing the message at the same time, not having access to the updated amount of peers that had the given chunk stored.

In order to solve this, two approaches were considered:

- The addition of a random delay before processing the PUTCHUNK message : this sort of approach was able to improve the excess number of Peers who stored a chunk, however, the results were only significantly improved if the range of the delay was big enough to make an impact on its performance.
- The addition of a new message to remove excess chunks : whenever the initiator Peer receives a STORED message for a chunk whose goal replication degree has already been hit, a message is sent (by the name of EXCESS), which indicates to a given Peer what chunk to remove. This approach was able to steadily control the maximum replication degree hit by a given chunk. **This was the approach taken.**

To conclude, this new message, sent in the multicast channel, added a new parameter to the header in the given, creating this new possible header:

`<Version> EXCESS <SenderId> <FileId> <ChunkNo> <PeerId> <CRLF><CRLF>`

Where Version must be ‘1.1’ and the PeerId is the id of the peer who needs to remove its instance of the chunk.

3. Restore Enhancement

In this section, the enhancement on the Restore subprotocol will be addressed, in order to answer to the problem created by the CHUNK message: many peers would be answering to the GETCHUNK message with the same body, flooding the Multicast Data Recovery Channel. It was also required by the specification for this enhancement to use TCP.

Therefore, the GETCHUNK message was changed in order to have a different format:

`<Version> GETCHUNK <SenderId> <FileId> <ChunkNo> <IP>:<PORT> <CRLF><CRLF>`

Where IP and PORT make up the address to connect to a Server Socket that the initiator peer had launched to receive the requested chunk. This way, the other peers will be able to connect to it and send the needed chunk. Even though more than one peer will be connecting itself to the initiator, it will only accept the first to successfully connect and receive the chunk from him.

4. Delete Enhancement

In this section, it will be discussed how the enhancement of the deletion protocol was handled, helping with issues regarding peers that are not running when the DELETE request is sent (or do not receive it), and, therefore, are unable to delete the file.

Several approaches were considered and, given the current structure of the protocol, the solution taken is the one who works most similarly to the Backup protocol: the initiator peer sends its DELETE message periodically, checking the current number of Peers who have already deleted their chunks. This information is obtained by a new message created for this matter: DELETED.

The DELETE and DELETED messages are therefore analogous to the already existing PUTCHUNK and STORED messages, with the substantial difference that the initiator peer will resend the DELETE message at a much longer period, since speed is not such an important factor in this specific subprotocol.

Since all the metadata regarding backed up files and locally stored chunks is kept in non-volatile memory, and available even if a peer crashes and has to restart, the deletion of chunks is possible even if a peer with a chunk to delete is not immediately up.