Distributed Systems

Assignment 1
Final Report

14-04-2020

Vítor Gonçalves up201703917
Vítor Ventuzelos up201706403

Concurrency

To allow for concurrency, the implementation resorts to the usage of multithreading in various manners. As each protocol functions differently, the methods used for this goal were also different.

When the **Peer** object is created, it creates three **MCastListenerThread** objects, one for each of the multicast channels used. These objects contain an infinite loop that reads from the assigned channel and act accordingly. Their behavior will be explored with more detail on the sections of the protocols.

All protocols instantiate an object that extends **Thread** which then executes all the operations of the protocol. These objects are **BackupMainThread**, **RestoreMainThread**, **DeleteMainThread** and **ReclaimMainThread**. multiple of these can be running simultaneously.

For the **Backup** protocol, a **MCastBackupSenderThread** object is created for each chunk that is read from the target file, which in turn sends the **PUTCHUNK** message to the **MDB** channel in accordance with the retry mechanism that was suggested. This class possesses a **Semaphore** object to prevent overloading the multicast channel. On the receiving side of this protocol is a **MCastListenerThread** which, upon receiving the message, stores the chunk and opens a **MCastResponseSenderThread** to send the **STORED** message after the recommended interval, leaving the listener thread open to any more oncoming messages.

For the **Restore** protocol, the **GETCHUNK** messages are sent sequentially, while **MCastListenerThread** assigned to the **MDR** channel is already active. On the non-initiator side, the **MCastListenerThread** assigned to the **MC** channel verifies if the Peer possesses the requested chunk and creates a **MCastResponseSenderThread** which is left responsible with checking whether the chunk was already sent by another Peer and, if not, sending the **CHUNK** message. Instances of this protocol may run alongside instances of other protocols, but not alongside instances of the same protocol.

For the **Delete** protocol, there are no additional improvements, as it is a simple protocol.

For the **Reclaim** protocol, the **REMOVED** messages are sent to the **MC** channel as the chunks are removed from the database. On the non-initiator Peers, the **MCastListenerThread** receives the messages and, in the event of the replication degree of a chunk falling under what is desired, creates a **MCastResponseSenderThread** which executes a normal **PUTCHUNK** subprotocol for the aforementioned chunk. In a similar manner to the **Restore** protocol, instances of this protocol may run alongside instances of other protocols, but not alongside instances of the same protocol.

Enhancements

1) Delete enhancement

This enhancement was implemented to solve the problem related to Peers that have chunks from a file that is being deleted but that at the moment the **REMOVED** message is launched, they are not running. For this reason, the space delegated to the storage of these chunks will never be reclaimed.

To solve this problem, it was necessary to create a **CONNECTED** message, with the following format <Version> CONNECTED <SenderID> <CRLF><CRLF>, which is launched by all peers of *version "2.0"* as soon as they are started.

When a Peer wants to delete the chunks associated with a file, after sending the **DELETE** message, the fileID of that file is saved in the Peer's storage. Thus, in this way, when the peer receives a **CONNECTED** message on the **MC** channel, he is able to know which delete requests to send on the **MC** channel to ensure that the chunks of his files are eliminated on all Peers that keep this information.

To ensure that all peers receive these messages, 3 **DELETE** messages are sent, with an interval of 500 ms between them. In order not to overload the **MC** channel, each peer must wait a random time between 0 to 400 ms before sending their messages to the other peers.

2) Backup enhancement

Given that the backup protocol in **version 1.0** leads to a huge amount of activity on the channels, due to the exchange of messages between peers, we decided to implement an improvement that would allow to solve, or in some way improve the protocol, making sure that **no functionality was lost** and to keep **communication between peers with different versions** possible.

In the Backup protocol of **version 1.0**, a peer that received the message **PUTCHUNK** would create a new chunk with the information received from the **MDB** channel and then immediately try to save it in its storage, **PeerDataBase**, waiting for a random time from 0 to 400 ms to send the STORED message to **MC** channel.

For **version 2.0** it was necessary to slightly reverse the order in which the protocol was performed in relation to the base version. In this approach, a peer that receives a **PUTCHUNK** waits a **random time** between 0 to 400 ms and only then checks whether it is necessary to send the **STORED** message. That is, if the Peer receives **PUTCHUNK** from a chunk with a replication degree of 3, and if during the waiting time it receives an **equal or greater** number of messages **STORED** on the MC channel for that same chunk, the peer will not proceed with storing the chunk and consequently it will not send a STORED message. In order to be able to count the stored messages that were received each peer has at its disposal a **ConcurrentHashMap**, which is only created if the version is 2.0, and when it receives PUTCHUNK creates an entry in the HashMap, where the key is **fileID: chunkID** and the value of the same **0**. For each message STORED for that same chunk, the key **value is increased**. Thus, when the waiting time passes, the peer compares the replication degree received with the stored value. If the last is less than the first, the chunk is stored and the message STORED is sent, in the opposite case nothing is done.