U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEIC 3rd Year

April 2020

Distributed Systems

Project 1 – Distributed Backup Service

Class 5 – Group 2

João Praça - up201704748

Lucas Ribeiro - up201705227

Sílvia Rocha - up201704684

Concurrency Design

In order to support the concurrent execution of protocols, we adopted several implementation approaches that contribute to a final highly concurrent and scalable solution.

Each instance of the class Peer contains an attribute for each multicast channel: MC, MDB and MDR. In the main method, three threads are executed (one per channel), all responsible for receiving messages sent by other peers. Whenever a message is received, it is copied and a new thread is created to process it (*ReceiveMessage*), allowing the peer to process multiple messages simultaneously.

However, creating and terminating many threads can lower the scalability of the design, so we decided to use *ThreadPoolExecutor* to avoid this issue. Furthermore, *Thread.sleep()* can also lead to many co-existing threads, limiting the scalability, therefore we implemented a *ScheduledThreadPoolExecutor* instead, that allows to schedule a timeout handler, without using any thread before the timeout expires.

Besides these improvements, we also decided to use *ConcurrentHashMap* instead of *HashMap* as a data structure to store *File*, *InitChunk* and *Chunk* instances in the class *Storage*, because it is the most suitable to use in contexts of high concurrency, assuring better performance and being thread-safe.

Finally, to assure that all data shared between threads is consistent, we´ve made some methods synchronized so that the data is not being accessed and/or modified at the same time.

Enhancement #1 - Backup

To achieve the exact replication degree indicated, avoiding problems such as space depletion, whenever a non-initiator peer receives a STORED message, it checks if the current replication degree of the chunk is already the desired one, and if so, if that chunk is in the list of chunks to send STORED messages, it deletes that chunk from its storage and from that list, meaning that it does not store a chunk if it is not strictly necessary (if the desired replication degree has already been achieved).

Enhancement #2 - Restore

When chunks are too large, to avoid all peers receiving the chunk, we decided to send the chunk through TCP only to the initiator peer, and the headers of the messages still through the multicast channel. This way, we assure interoperability between non-initiator peers and avoid sending to all peers a large chunk.

Enhancement #3 - Delete

When an initiator peer sends a DELETE message to a peer that has some chunks of the specified file but is not currently active, it stores the chunks to be deleted in a serializable file named *deleteEnhancement.ser* so that when the peer is active again, it can deserialize the file to delete the files indicated and that are present in its storage.