Implementation Details

Distributed Systems
1st Assignment

Section 6 - Group 07

Ana Margarida Ruivo Loureiro - up201705749
João Miguel Ribeiro de Castro Silva Martins - up201707311

Concurrent Execution

In the development of the project, we achieved a highly concurrent and scalable implementation.

The strategy used supports several tasks at the same time by using threads and a manager for their execution.

To be easier to explain how we manage the concurrency implemented in the assignment, we'll start to explain the structure of the project. Therefore the code structure can be divided into the following categories:

- **Channels**: channels and manager that aggregates all;
- **Messages**: structure of the messages used;
- **Storage**: holds the information for each peer;
- **Tasks**: subdivision of protocols into smaller tasks;

Now we'll go through the common steps of a subprotocol to better explain how our implementation is allowing concurrency.

First, a request arrives at the initiator peer. Then the initiator passes the necessary information to the other peers so that they can also process it. This communication is achieved through **multicast channels** where information is sent and received.

To support the concurrent processing of different messages received on the same channel **each peer has a specified thread for each multicast channel.**

Once the information is received it is passed to a **new thread to process** it. This allows several **tasks** to be treated at the same time. However, the creation of threads alone comes with some overhead in its creation and termination. As an alternative to it, we opted to use a **thread pool** - *ThreadPoolExecutor*. It implements a more scalable design which allows avoiding creating a new thread to process each message.

It should also be pointed out that many structures are shared between tasks. Therefore we used structures adapted to concurrent accesses. More specifically we applied the java ***ConcurrentHashMap*** to store the shared information. It has the same functionalities as a hash table but supports full concurrency of retrievals and high expected concurrency for updates. The functionalities of each of these structures will be mentioned more specifically next in the report.

Following, we'll discuss some details in the implementations of the subprotocols where we apply concurrency strategies we consider relevant to be discussed in this report.

Stored Messages and Replication Degree

We should explain the architecture behind the counting of the stored messages since it builds a strategy that handles many concurrency aspects.

To collect the information about the *STORED* messages and subsequently the actual replication degree of a chunk we created some new auxiliary structures. All those structures are kept in the ***StorageManager*** where all the information related to the peer storage is available.

A peer needs to process a message if it is the initiator of the respective backup process or if is one of the peers who stored the corresponding chunk. However, a peer needs to know which *STORED* messages coming from the control channel are meant to be processed. Thus they add the information of the backup chunks working on at the moment in a string set - **stored_chunk_request -** before sending the *PUTCHUNK* messages or when backing up the chunk.

The information of *STORED* messages is stored by the peers. This way it's possible to keep track of the number of messages sent. Consequently, those numbers correspond to the replication degree of that specific backup process. To achieve the aforementioned we implement a *ConcurrentHashMap* - **stored_senders** - of the form <String, Set<Integer>> where we can keep track of all the senders that already stored the chunk. Therefore it's possible to exclude duplicates *STORED* messages since the backup process asks more than once about the stored chunks in order to have a more **foolproof** value.

Restore Process

The initiator restore process asks progressively for each chunk related to a specific file. This progress is based on the response it gets from the other peers. Furthermore, it should store the received chunks so that in the end it can restore the file with all the chunks collected.

Like the *STORED* messages in the backup subprotocol, the chunks are sent to all the peers via the *MDR* channel. Thus an *ArrayList* - **restore_requests** - is used to store information about each restore request sent by the initiator. Consequently, we can associate the peer that wants to process the sent message based on the matches made between the information in the structure and the one in the *CHUNK* message.

These CHUNK messages that arrive at the initiator are handled in tasks. Each task stores the chunks in a *ConcurrentHashMap* - **restored_files -** of the form <*String*, *Map<Integer*, *byte[]>>*. The use of this structure allows the peer to store all the chunks of

a file in the same key. In addictions it is possible to verify if the chunk received was already stored (duplicate).

Enhancements

Delete

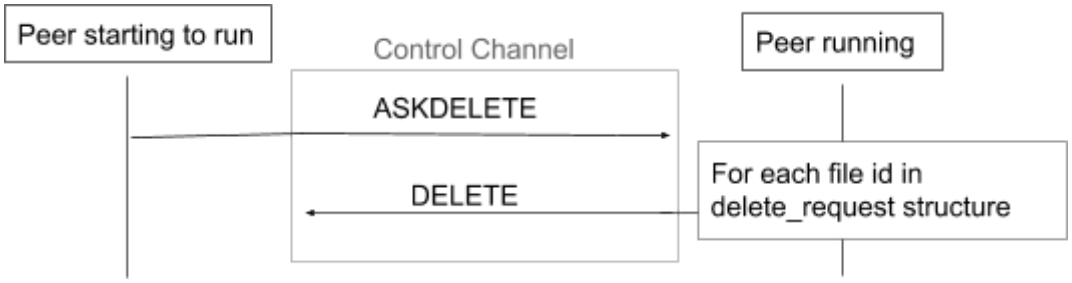
The enhancement applied to this subprotocol solves the problem that happens when peers are **offline** in the moment of a delete request and subsequently won't process it. Therefore maintaining in storage **unnecessary space**.

Thus to solve this problem we implement an auxiliary structure that keeps track of all the delete requests initiated by the peer. This auxiliary structure - ***delete_requests*** - stores the file id correspondent of the delete request and it's kept in the *StorageManager* that holds all the information related to a peer storage.

The enhancement strategy consists of handling the delete request as the normal version except for adding its file id in the auxiliary structure. However when a peer that was offline starts running a new task is executed. There the peer asks for the information related to the delete requests already processed by the other peers. In order to do so, it **sends a message** to the control channel with a structure similar to:

<Version> ASKDELETE <SenderId> <CRLF><CRLF>

The other peers receive it and start to process it in a new task as well. There they create a normal delete request task for each file id stored in the delete_request structure. This way the requests that weren't processed by the once offline peer will now be normally processed.



Restore

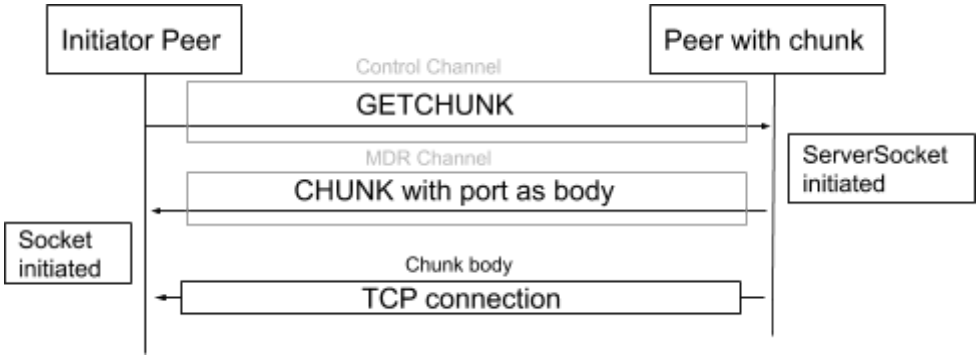
In this subprotocol, the initiator firstly sends a *GETCHUNK* message to the Control channel. Next, all the peers with chunks associated with the request answer by sending the chunk body in a *CHUNK* message. However, this message is sent to the *MDR* channel and all the peers will receive it even though only the initiator peer will need its information.

Therefore, instead of sending the chunk body to the *MDR* channel, we use a **TCP connection** between the initiator peer (client) and the peer with the chunk information (server).

However, to establish the *TCP* connection the initiator peer (client) needs to know which **port** and **address** should use. With that in mind, after the peer receives the *GETCHUNK*, it starts the *ServerSocket* and sends a *CHUNK* message to the *MDR* channel, which contains the port in the **body**. The address will be obtained later as explained below. After the client begins the *TCP* connection as well the *ServerSocket* accepts it to exchange data and send the chunk body.

Meanwhile, the initiator processes the message and starts the socket with the information received: the address is withdrawn from the *DatagramPacket* and the port is within the message body. After this, it reads the chunk from the *TCP* socket and continues normally with the remaining parts of the subprotocol.

Although this solution still uses the MDR channel to pass a message only needed by the initiator we reduce its size considerably. Because of this, the usage of the channel is reduced and it upholds an higher traffic of messages.



Backup

The problem with the original implementation of this subprotocol is that we only verify if the desired replication degree was reached at the beginning of each iteration in the initiator. This leads to many peers backing up **unnecessary** chunks.

This way in the enhancement the verification in the initiator is still implemented. However, the peers that receive the chunk don't wait a random time at the end of the backup task (before sending the *STORED* message). Instead, the peers wait at the beginning of the task. Therefore, after the waiting time, the replication degree is checked in the peer, thus allowing them to do a verification of the replication degree before the backup process. Consequently, the number of unnecessary backed up chunks will be reduced significantly.