U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEIC 3rd year

SDIS

April 2021

Distributed Backup Service

Report

(Project 1)

Class 4 - Group 7

Rui Filipe Mendes Pinto - up201806441

Tiago Gonçalves Gomes – up201806658

Abstract

This report aims to explain all the enhancements implemented in this project, namely in the backup, restore and delete subprotocols. Furthermore, we will describe the used concurrent design.

The project was developed within the scope of the Distributed Systems course.

Backup Subprotocol Enhancement

The “1.0” version scheme can deplete the backup space rather rapidly, and cause too much activity on the nodes once the space is full. Because of that, we designed an enhancement, included in version “2.0”, that ensures the desired replication degree, avoids these problems and can interoperate with peers that execute the chunk backup protocol of version “1.0”.

In the PeerStorage class, which holds all information regarding a peer’s filesystem, we created a ConcurrentHashMap data structure, named **numberOfStoredChunks**, that has a key that follows the “fileID chunkNumber'' format and stores the number of occurrences a chunk was stored, by him or by other peers. So, every time we receive a **STORED** message we increment that counting in the ConcurrentHashMap for that specific chunk, and every time we receive a **REMOVED** or a **DELETE** message we decrement it. That way, before storing a chunk, we can check if the chunk’s desired replication degree is already satisfied because we always keep this counting up to date. If so, the chunk is discarded, otherwise, the chunk is stored, its counting is updated, and the normal response procedure is done, allowing other peers to update their counting.

Overall, this enhancement brought us a nice performance improvement. In practical terms, if we have for example 10 peers and we make a backup with the desired replication degree of 2, according to version “1.0”, all 10 peers will store that chunk and its final replication degree will be 10. When running version “2.0”, the chunk’s final replication degree will highly probably be 2.

That being said, we clearly showed the kind of optimization this enhancement does.

Restore Subprotocol Enhancement

In the “1.0” version, if the files are large, the restore protocol may not be desirable: only one peer needs to receive the chunks, but we are using a multicast channel for sending all the file’s chunks. For that reason, we implemented this enhancement, included in version “2.0” that uses TCP and is able to interoperate with non-initiator peers in case the initiator runs the “2.0” version and the other runs a mix between the “1.0” and the “2.0” versions.

First of all, we implemented a function “findAvailablePort()” that returns an available port, so that we can initiate the TCP connection. This function is extremely important because we shouldn’t bind the same TCP Socket for 2 different file restores, as different information will be sent. So, for each file we want to restore, we have a different port number, stored in a ConcurrentHashMap, named **filePorts**, in the PeerStorage class. This structure has a key that follows the “fileID chunkNumber'' format and stores the number of the respective TCP Port.

For sending the TCP port number to the peer that has the chunk we want to receive, we created a new message **TCP_PORT** with the following header:

<version> TCP_PORT <senderId> <FileId> <portNumber> <CRLF><CRLF>

The restore protocol regarding version "2.0" always starts with the sending of that message in the **MDRChannel.** After that, the peer can connect to the initiator peer that acts as a Server, listening for inbound connections, and send the respective chunk in a much reliable way, via TCP. This way, on the server-side, we have a ServerSocket waiting for connections (ReceiveChunkTCP class) that will parse every chunk received and allocate it in the **restoredChunks** data structure that then will be used to restore the actual file. On the client-side, we create a Socket object identified by an IP fetched from the UDP packets sent by the Server (initiator peer) and get the port specified in the beginning by the initiator. We then send the CHUNK to the initiator. To decrease the level of concurrency in the socket we also use a mutex that ensures the sending of each Chunk is made independently.

This enhancement brought us many performance improvements. With the TCP connection, we don't flood the **MDRChannel** with **CHUNK** messages that have the chunk's body, like in the "1.0" version. We just send a **CHUNK** message without the body to warn other peers that that specific chunk was already sent, and establish a TCP connection to start sending the respective chunks without the use of any Multicast channel.

Delete Subprotocol Enhancement

In the "1.0" version, if a peer that backs up some chunks of the file is not running at the time, the initiator peer sends a DELETE message for that file, and consequently, the space used by these chunks will never be reclaimed. Because of that, we implemented an enhancement, included in version "2.0", that allows reclaiming storage space even in that event.

To implement this, we created a new type of message, with the following header:

<version> ON <senderId> <CRLF><CRLF>

Every time a peer is initiated, it sends the above-mentioned multicast message to the **MCChannel**. It's important to mention that every peer holds a ConcurrentHashMap data structure named **peersBackingUp** that has a key that follows the "fileID chunkNumber'' format and stores the peers backing up that specific chunk. We also have a Set of file Ids, named **filesToRemove**, that has the ids of the files that are meant to be deleted. When the initiator peer receives a **STORED** message, it registers the sender peer in that ConcurrentHashMap and when a **REMOVED** message is received, it removes it. But here we have a problem. How to know if a peer successfully deleted a file? For this, a confirmation message was used. It has the following header:

<version> RECEIVED_DELETE <senderId> <fileId> <CRLF><CRLF>

Like in the **REMOVED** message, when an initiator receives a **RECEIVED_DELETE** message, it also removes the peer who sent it from **peersBackingUp**.

This way, when an initiator receives an **ON** message it starts searching if the peer who sent it has some pending deletion. If so, a **SPECIFIC_DELETE** message is issued in the **MCChannel** and the peer will have to delete the respective file chunks. This message has the following format (where *peerId* stands for the peer's id that will perform an action over the reception of the message):

<version> SPECIFIC_DELETE <senderId> <fileId> <peerId> <CRLF><CRLF>

Likewise, when an initiator peer starts its execution it issues delete messages directed to specific peers that have a pending file deletion.

Having that said, we conclude that this enhancement successfully makes deletion of file chunks possible even if a peer is offline.

Concurrent Design

The design implemented in our project allows the simultaneous execution of all subprotocols. We tried to follow the hints in the "interoperability" section mentioned in the project's guide. As a result, we came up with a structure that, in the already mentioned **peerStorage**, uses *ConcurrentHashMap* instead of *HashMap* that is more reliable and has an outstanding performance in multi-threaded environments. During the development of the project, we faced some problems regarding messages being lost/not arriving on time, duplicates, etc. Most of this because we were using UDP as our Transport Protocol. To solve this, the use of *timeouts* was crucial, therefore we used ***Thread.sleep()***. Most importantly, we used a *ScheduledThreadPoolExecutor* that allows immediately executing a Thread/Runnable object or schedule that execution for a defined amount of time. This uses a set of workers that handle several tasks assigned by a dispatcher which is internal to the Java class. The use of a *ScheduledThreadPoolExecutor* also won't have to consider scalability errors regarding the number of threads created, and that's a huge win. So, as mentioned, *ScheduledThreadPoolExecutor* is located in Peer's class. Across the project, as this variable is static, we use it everywhere.

Regarding the three multicast channels used for message exchanging: **MCChannel**, **MDBChannel** and **MDRChannel** we have an attribute for each one of those in the Peer's class and we then call *ScheduledThreadPoolExecutor.execute()* over each one of the channels so that a thread can be created for each one of them.

```
public static MCChannel mcChannel;
public static MDBChannel mdbChannel;
public static MDRChannel mdrChannel;
public static PeerStorage storage;
public static ScheduledThreadPoolExecutor scheduledThreadPoolExecutor;
public static Semaphore semaphore;

public Peer(String IP_MC, int PORT_MC, String IP_MDB, int PORT_MDB, String IP_MDR, int PORT_MDR) throws SocketException, UnknownHostException {
    mcChannel = new MCChannel(IP_MC, PORT_MC);
    mdbChannel = new MDBChannel(IP_MDB, PORT_MDB);
    mdrChannel = new MDRChannel(IP_MDR, PORT_MDR);
    scheduledThreadPoolExecutor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(Macros.NUM_THREADS);
    isInitiator = false;
    semaphore = new Semaphore( permits: 1,  fair: true);
}
```

Figure 1. Creation of the Channels and *ScheduledThreadPoolExecutor*

```
scheduledThreadPoolExecutor.execute(mdbChannel);
scheduledThreadPoolExecutor.execute(mcChannel);
scheduledThreadPoolExecutor.execute(mdrChannel);
```

Figure 2. Use of *scheduledThreadPoolExecutor* to instantiate a new thread for each channel

On each channel, every time a message is received, it's parsed by the **MessageManager** class and a new thread is created according to the type of message.

Furthermore, we used **synchronized** methods mostly in the **PeerStorage** class to allow multiple access to the shared data structures, but without compromising its integrity. What synchronized methods do is lock a method for that specific object and no other thread that shares that same object can access it until it is unlocked.

To keep all the information about chunks in non-volatile memory we used **Java Serialization**. With this, in our **peerStorage** class, we could represent the entire class concerning the storage of information, as a sequence of bytes. Or, in other words, as a file. When a peer starts, it deserializes the storage causing it to be like it was in its previous state. This way, all attributes and data structures we had earlier are now restored and we know exactly what we did before. This was a fundamental step to save the peer state every time it went on and off.

Conclusion

The development of this project was quite challenging because we had to think about many aspects that could happen if we did this or that. All those problems faced were successfully overcome and we managed to develop all the required functionalities and also all the extra ones.

That being said, we improved our knowledge in Distributed Systems and can now build complex systems that work in various computers that can or not be in the same network.