U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Class 7 Group 5

Diogo Samuel Gonçalves Fernandes up201806250@fe.up.pt
Juliane de Lima Marubayashi up201800175@fe.up.pt

11th April 2021

SDIS Project - 2020/21 - MIEIC

Teachers

Pedro Souto pfs@fe.up.pt
Rui Rocha ruirocha@fe.up.pt

Index

1 Introduction

The aim of this project is to elaborate a distributed system service for backing up files in a LAN. All the features requested for this work were implemented:

- Chunk backup
- Chunk restore
- Delete a file
- Manage a local service storage
- Retrieve local service state information

We also implemented the following enhancements:

- **Backup Enhancement** - Each chunk is not stored more times than its replication degree;
- **Restore Enhancement** - Using TCP Protocol;
- **Delete Enhancement** - Upon a delete command, peers that are not working by the time of the request will receive the delete command when connected.

In this report we will explain our solution to the multi-threading problem of our project and also how we implemented the three enhancements mentioned above.

2 Concurrent Execution

To allow the concurrent execution of the protocols, we've used Threads, Concurrent Hash Maps and Timers.

When the Peer is initialized, it will start running each Multicast Channel and if the program is running in the version 2.0, an additional TCPChannel is created. Each one of Multicast Channel implemented runs in a separated Thread, allowing messages to be received at the same time and are implemented bellow:

```
1  public static void initChannel(String mcast_addr, int mcast_port, String mdb_addr, int
       mdb_port, String mdr_addr, int mdr_port) throws IOException {
2      new MCChannel(mcast_port, mcast_addr).start();
3      new MDBChannel(mdb_port, mdb_addr).start();
4      new MDRChannel(mdr_port, mdr_addr).start();
5      if (version.equals(Singleton.VERSION_ENH))
6          new TCPChannel().start();
7  }
```

Each time a message is received in one of this channel, it will be processed in a new Thread. Those threads, are located in the `process` package in our code and the channel threads are at the `channel` package. The Concurrent Hash Map is used to allow simultaneous writing in the Hash. For example, at the class `State` (class responsible for storing the Peer state), the Peer will have the State updated by multiple simultaneous protocols:

```
1  public ConcurrentHashMap<String, FileState> filesBackup = new ConcurrentHashMap<>();
2  public ConcurrentHashMap<String, ChunkState> chunkStored = new ConcurrentHashMap<>();
3  public ConcurrentHashMap<String, List<String>> filesToDelete = new
       ConcurrentHashMap<>();
```

In order to save the State of a Peer as a file, we implemented the serialization of the State. The serialized file is saved using a Thread that stores it every 30 seconds.

Also, by when the program shuts down, the State is saved using a Java Hook. The Serialization of Java Classes was fundamental since this allowed us to save the State of each Peer in non-volatile memory.

We also used the Timer to schedule the TimerTasks instead of Sleep, since the last one can lead to busy-waiting due the presence of many threads running simultaneously. Those were used mainly for the delays between receiving and sending the answer for the message. As an example, the code below show how we implemented this feature at the reclaim protocol, where a Peer, upon checking that a replication degree of a chunk is below the desired, it will start a backup process after a certain delay between 0 and 400ms:

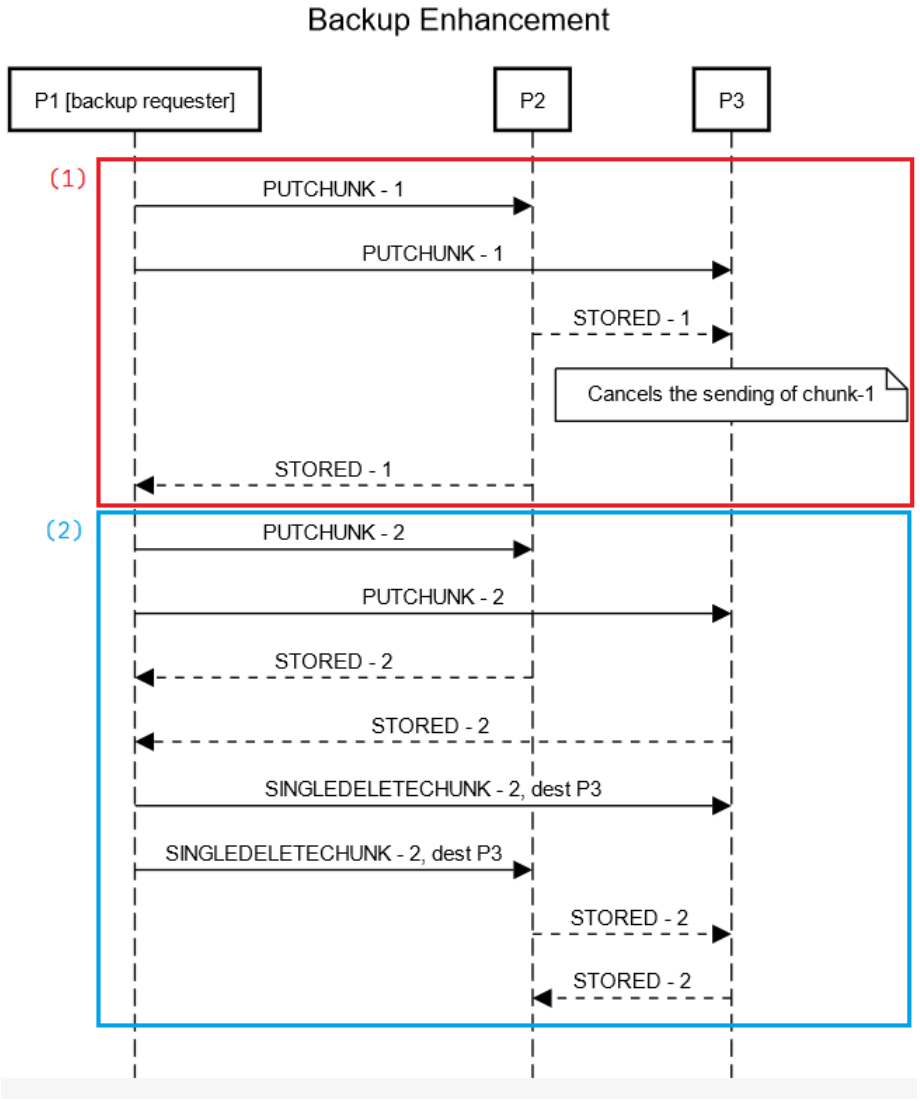
```
1   /*
2    * Will create Timer to schedule the operation.
3    */
4   private void scheduleBackup(String replicationDegree) {
5       Timer timer = new Timer();   // A new thread Timer will be created.
6       timer.schedule(createTimerTask(replicationDegree), new Random().nextInt(401));
7       Peer.reclaimBackupTasks.addTask(chunkId, timer);
8   }
9
10  /*
11   * This method will create the timerTask to be scheduled by the timer.
12   */
13  private TimerTask createTimerTask(String replicationDegree) {
14      return new TimerTask() {
15          @Override
16          public void run() {
17              new RequestPutChunk(chunkId, replicationDegree, 0).start();
18              this.cancel();    // Do not repeat.
19          }
20      };
21  }
```

3 Enhancements

This chapter is reserved to explain how the enhancements were implemented.

3.1 Backup Enhancement

In order to avoid other peers to deplete it's storage by storing files that has already achieved the desired replication degree, the backup enhancement guarantees that each chunk is not stored more than necessary. To explain the implementation in details consider the image bellow, where Peer 1 has requested a backup of a file with 2 chunks and replication degree of 1:



In the first part (1) of the protocol, when the Peer 2 receives the **PUTCHUNK** request, it waits a random time x between 1 and 400 ms to send the **STORED** message to the Peer 1. The Peer 3 also chooses a random time, but in this case, the time chosen was higher than x. Thus, the Peer 2 will send the **STORED** message before the Peer 3. By this time Peer 3 receives the stored message from Peer 2, it will cancel the saving file process that was scheduled to execute after the random time.

However, consider the situation (2) that the random time chosen by Peer 2 and Peer 3 are very close to each other. For this case, Peer 3 will not receive **STORED-2** message from Peer 2 before saving the file locally. In that situation, the Peer that has requested the backup (Peer 1) will receive the **STORED** message from Peer 2 and Peer 3. When it receives the message from Peer 3, it will notice that the replication degree for that chunk has been achieved, thus it will send a **SINGLEDELETECHUNK** message in the multicast channel with Peer 3 as destination. By reading this message the Peer 3 will delete the chunk 2 recently stored and other Peers that are not the destination of the **SINGLEDELETECHUNK** will just update their state.

The **SINGLEDELETECHUNK** has the following structure, where the <DestinationId> field is used to indicate the number of the Peer of which this message is intended:

```
<Version> SINGLEDELETECHUNK <SenderId> <FileId> <ChunkNo> <DestinationId>
    <CRLF><CRLF>
```

3.2 Restore Enhancement

The goal of the restore enhancement is to have a reliable restore service that does not lose chunks in the process. Since the usage of the UDP protocol could lead to some messages lost, we decided to implement the TCP protocol to avoid this situation.

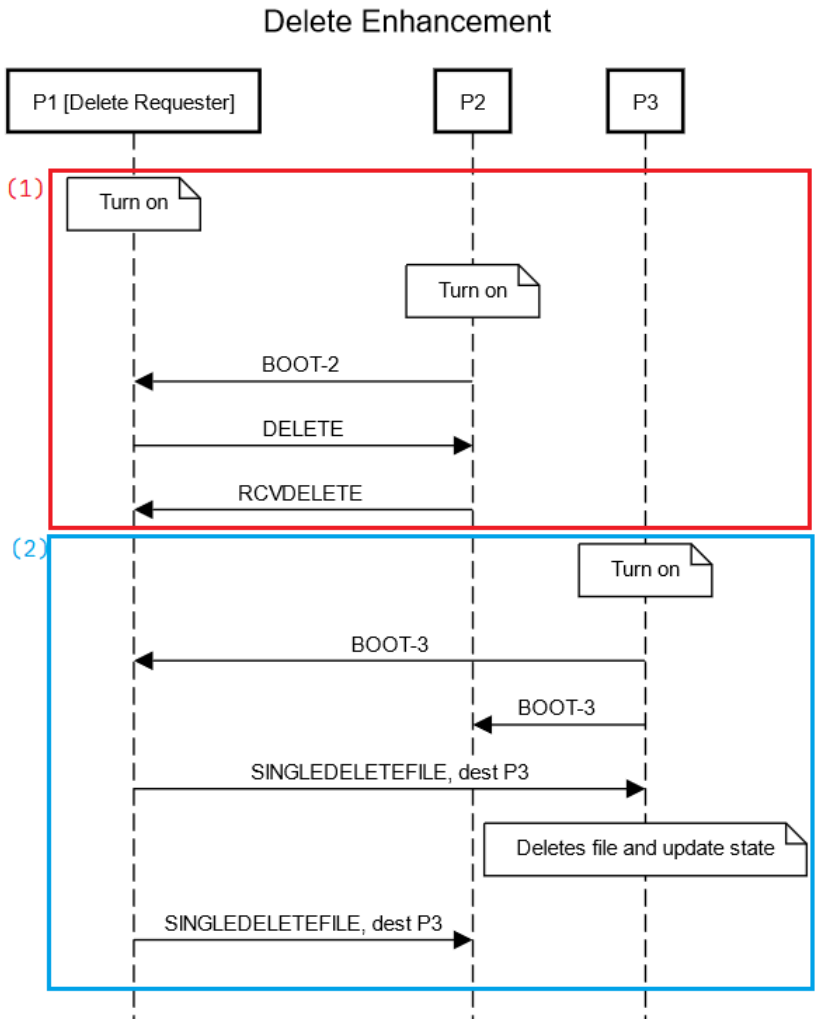
To implement this, every Peer on boot will start a TCP Server on an open port. Then, when the restore protocol is called, the peer that has started the store operation will send the port of their TCP server in the header of the **GETCHUNK** message:

```
<Version> GETCHUNK <SenderId> <FileId> <ChunkNo> <CRLF> <Port> <CRLF><CRLF>
```

The peer that has the corresponding chunk will send **CHUNK** message to the TCP port and the information will not be lost.

3.3 Delete Enhancement

This enhancement allows Peers, that are not connected by the time of the delete command, remove the supposed file when it turns on. To explain the protocol, consider the image below, where both Peer 2 and Peer 3 have a specific file locally stored and Peer 1 requests to delete it when Peer 3 is off:



To sustain this protocol we have implemented the **BOOT** message:

```
<Version> BOOT <SenderId> <CRLF><CRLF>
```

When a peer turns on, it announces this event in a multicast channel. In the graphic above, for example, by turning on, the Peer 2 announces in a multicast channel that it's connected. When a Peer requests a **DELETE** it creates a list (RemainDelete) with the Peer's Id that are supposed to receive that message, that is, peers that have a backup of that file. When other peers receives that request, they send a **RCVDELETE** response, notifying that they have received the request. When the Peer that has requested the delete receives a **RCVDELETE** notification, it updates the RemainDelete, by removing the SenderId from it.

```
<version> RCVDELETE <SenderId> <FileId> <CRLF><CRLF>
```

As an example, in the graphic above, the Peer 1 will generate a RemainDelete list [3,2]. When it requests a **DELETE**, Peer 2 will send a **RCVDELETE** response and by the time Peer 1 read this, the RemainDelete will be update to [3]. Which means that Peer 3, still didn't receive the **DELETE** message.

After the Peer 3 boots (2), it sends in the multicast channel the **BOOT-3** message. Since the Peer 3 remains on the RemainDelete, Peer 1 will send in the multicast channel the **SINGLEDELETEFILE** with Peer 3 as destination and delete Peer 3 from the RemainDelete, causing it to be empty. The **SINGLEDELETEFILE** message is shown above:

```
<Version> SINGLEDELETEFILE <SenderId> <FileId> <DestinationId> <CRLF><CRLF>
```

The peers that don't have the same Id of the <DestinationId> will ignore the **SINGLEDELETEFILE** message. Otherwise, it will delete the file with FileId specified in the SINGLEDELETEFILE message.