Distributed Backup Service

Integrated Master in Informatics and Computing Engineering

Distributed Systems

Report

Group T3G05

João Francisco de Pinho Brandão - up201705573@fe.up.pt
Nuno Miguel Teixeira Cardoso - up201706162@fe.up.pt

14 de Abril de 2020

Introduction

This report aims to explain in detail the improvement implemented in the project's basic protocol: backup, restore and delete. As well as, describe the chosen design that allows the simultaneous execution of protocols and explain its implementation. Finally, the project was developed within the curricular unit of Distributed systems.

Backup Protocol

Regarding the improvement of the Backup protocol, this has the objective to guarantee the desired replication degree and, consequently, save memory. In the normal version, we use the replication degree as a minimum chunk repetition value, so all peers except the initiator one will have a chunk repetition.It was also implemented in the normal version a timeout with five attempts, which is ensured by the Thread *PutChunkAttempts.* However, in the improved version we only store a chunk in a peer if the current number of repetitions is less than the desired replication degree, thus decreasing the amount of space occupied. In short, this solution proved to be quite efficient, given the chunk's replication degree is likely to be higher than the desired replication degree which is very low.

Restore Protocol

Regarding the improvement of the Restore protocol, after implement the normal version which sends a message by multicast asking for the chunks of a given file to be rebuilt, it also sends a message from other peers submitting the chunk with header and body by multicast. In the improved version, the same initial *GETCHUNK* message is sent by multicast, but the corresponding reply only has the header part. Thus, the content of the is sent by TCP directly to the peer that requested the chunks.

Delete Protocol

Regarding the improvement of the Delete protocol, we started by implementing the normal version by simply deleting the respective requested files and chunks from all active peers. To improve this version, we do the same as in the normal version, but peers also receive an AWAKE message whenever a new peer connects to the network. Thus, chunks that had not been removed from deleted files because the peers were turned off, make these peers receive a new message DELETE that deletes the chunks.

Simultaneous Execution of Protocols

Regarding the implemented design that allows the execution of protocols, we took into account numerous factors. Starting with the appropriate choice of data structures, instead of using *HashMap*, we opted for an alternative structure, *ConcurrentHashMap*. This is suitable for multithreading as it is more secure, scalable and has excellent performance when the number of read threads exceeds the number of write threads.

We started by implementing an attribute in the Peer class for the type of channel: a *Channel* array in which the position 0 represents the control channel MC, the position 1 represents the restore channel MDR and the position 2 represents the backup channel MDB. This array is initialized when a peer is initialized and for which we create a thread in the main function. In addition, we take advantage of the features of *ScheduledThreadPoolExecutor*, which allows us to implement a timeout that does not block the thread that is running at the moment.

```
22          //Parse channels
23          Channel[] channels = new Channel[3];
24          try {
25              Channel MC = new Channel(args[3], args[4]);
26              channels[0] = MC;
27              Channel MDB = new Channel(args[5], args[6]);
28              channels[1] = MDB;
29              Channel MDR = new Channel(args[7], args[8]);
30              channels[2] = MDR;
31          }
32          catch (IOException e) {
33              System.out.println(e.toString());
34              return;
35          }
36
37          //Create executor
38          threadExecutor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool( corePoolSize: 10);
```

The *SendMessagesManager* Thread is responsible for sending messages to the respective channel, depending on the action that are executed.
In each channel, whenever a message is received, a thread is created that processes it, that means, it is possible to process several messages at the same time. The thread responsible for processing the messages is the class *ReceivedMessagesManager*. This main Thread then chooses which supplementary Thread should execute in each moment depending on the subprotocol given.
This procedure occurs for all the sub-protocols of version 1.0. In enhanced mode, that is version 2.0, happens exactly the same, except in certain cases. For example, as stated before, in the enhanced restore it is also used TCP. For that the chunk content is sent and received using *SendRestoreEnh* and *ReceiveRestoreEnh*, respectively. Only this two versions are available.
Additionally, the group took advantage of Java synchronization, which is the ability to control multiple thread access to any resource shared. In this way, it was used synchronized in several methods, since it is the best option to allow only one thread to have access to a shared resource at a time. An example of the application of this methodology is in the

backup method in the Peer class. The Java language provides a mechanism, called serialization of an object, which consists of an object being able to be represented by a sequence of bytes that include the data of the same, as well as information about its type and its stored data. After a serialized object has been written to a file, it can be read from it and the serialization canceled, that means, the type information and the bytes that represent the object, as well as its data can be used to recreate the object in memory. This mechanism was fundamental to save a state of the application and start from it, still, seeing the information fully consolidated in the storage attribute of the Peer class, it was only necessary to save the Storage object of each peer. The serialization of this object is done in the savePeerStorage method, and extraction in the method loadPeerStorage.