Distributed Backup Service*

João Paulo Gomes Torres Abelha[1] & Vitor Hugo Pereira Barbosa[1]

Faculade de Engenharia do Porto, Portugal

Abstract. In this document, we will describe how we implemented a solution to the proposed project as as well as the enhancements made.

Keywords: Distributed Systems · RMI · Concurrency · UDP · TCP · Java

1 Introdução

The document is divided in two different sections, namely the one where the architecture is explained and how concurrency was implemented and the other where we explain our enhancements.

2 Concurrency Design

One of the most important classes for concurrency is the the Threadpool class which has a ScheduleExecutorService object made to run actions corresponding to the different protocols and allows you to schedule actions to be executed after a specified time, which has proven very useful for some parts of the protocols where a random time is requested for sending messages. The action schedule also proved to be important as we can get a Future object that can later be cancelled, this feature was used in multicast channels to cancel actions that were scheduled when messages were received for the same chunk, as can be seen in checkPutChunks and checkChunks, in ChannelBackUp and ChannelRestore classes.

Since different threads share the same data structure and data, we use Concurrent Collections from the java package java.util.concurrent in order to prevent problems when multiple threads try to access and modify the data simultaneously, the most used one was the ConcurrentHashMap. In some cases it has also proved necessary to use the the synchronized keyword, if there are blocks of code that modify and access shared resources and do not correspond to the java Concurrent Collection.

In order to have different simultaneous procedures on each Peer running, it is necessary for each protocol to run asynchronously in each Peer. This is done to divide the workload in different units corresponding to different threads.

In order to make achieve proper use of concurrency in the project, we started by implemeting a Channel class which implements Runnable. By doing so, we

* Supported by FEUP.

created a class that can be executed asynchronously. Indeed, what the @override void run()... method does is to continuously listen for DatagramPackets on the corresponding Multicast Group. Whenever one of this packets arrives, a new function will be placed in the ThreadPool a new function which is responsible for parsing the DatagramPackets into a Message class and then a and then depending on the action of the message create a new class responsible for handling that type of message. If this message was one the sent by the current Peer, then it is discarded, except for PUTCHUNK messages, where we check if the peer has the required chunk and send a stored message if so.

The internal state of the Peer is loaded from the non-volatile memory from the file System. We also use a Timer and TimerTask to update the state of our memory made in the disk every five seconds. For that some classes implement the Serializable interface to save in memory the object in java itself. With this and by using the ObjectOutputStream and ObjectInputStream we can write and read to and from the disk at once. Each time a peer is shutdown we use a class named ShutDownHook to save to the disk all the information that may not have been saved and we also delete in the backup directory all the sub folders that are empty. The Peer class, implements the RMI service methods so it can be simultaneously a peer initiator to one of those protocols and a peer which responds to other peers requests.For that we use use two thread pools, one used to run the requests made, for instance, send putchunks and receiving the stored messages and the other one to respond to requests. On an extra note, to guarantee that we don't fill the multicast channel with a lot of messages, we have at most ten threads to manage the SendPutchunk and SendGetChunk classes. Only after receiving a confirmation from those action we start a new one. To verify that our application supports concurrent tasks without any problem we tested by executing various protocols at the same time using different peers to respond to the requests or to make them themselves.

2.1 Important classes for Concurrency

The folder src is where all the code is present.

ThreadPool This is where an ScheduledExecutorService object is encapsulated.It is going to work as a fixed window for threads to run. It is going to be used to help performing the protocols and handle the messages that arrive and its consequent action.

PeerChannels This class abstracts all multicast channels that a peer has and it is constantly listen to.

PeerInformation This class has all information related to the peer. Since this information is shared among various threads we use the packet java java.util.concurrent and synchronized keyword to help us to maintain a consistent information.

Message This class is used to extract from the DatagramPacket all the information related to that message. It encapsulates this, making it easier to handle and query its values.

3 Enhancements

All the enhancements were made and they all try to respond the best way possible to the problems that were risen without them. For this, there are two different version, the standard version - "1.0" and the one with all the enhancements - "2.0". All the versions are inter-operable with each other.

3.1 Backup subprotocol

Problem: This scheme can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full.

Solution: This enhanced version allows the decrease of activity once the backup space is full or almost full. Moreover, the chunks that achieve the replication degree that was desired are not stored since the replication does not need to be higher.

While the standard protocol, waits a random time interval between 0 and 400 ms after saving the chunk and before sending the stored message, the enhanced version **after receiving a putchunk message waits a interval** which is **calculated accordingly to the available memory**, making the wait probabilistic higher in the peer which have less available memory:

$$wait = randomBetween(percentageOfMemoryUsed * 4, 400)ms$$

During this time, **they listen to STORED messages**. A peer with 0 KB of used memory will wait a random time between [0,400] ms while if it has half of his memory used will wait a random time between [200,400] ms.This way **there is a higher chance that a peer with less memory available will wait more and with that there is also a higher chance that more stored message were listened** and with that the replication degree might have been achieved.

If the number of stored achieved the desired replication degree, the thread exits since it is not necessary to store the chunk. However if that does not happen, the thread saves the chunk and send the stored message as soon as it can since it already waited a random time earlier.

3.2 Reclaim subprotocol

Problem: In the restore of a file, since a multicast channel is used to send the messages with the chunk content, all the peers will have to interpretate these messages, but only one of them is waiting for that message, leading to a waste of time on the remaining peers that will have to interpret large messages.

Solution: In order to solve that, an improved protocol was created, using a TCP connection instead of a multicast channel.

The initiator peer to establish a **TCP connection uses the ServerSocket** class, **passing 0** as an argument in the constructor that will automatically generate a value for the port that is then accessed through getLocalPort.

A GETCHUNK message is then sent, that in addition to the usual parameters **has the port obtained from ServerSocket**. After sending the message the initiator peer creates a Socket that will wait for a connection to accept.

The remaining peers when receiving the GETCHUNK message and, if they are aware of the chunk requested, they will **establish a connection to the Socket from the initiator peer through the port passed in the message and the DatagramPacket address**, if they cannot establish a connection, the peer ignores it and proceed since this happens when the intiator peer is not the first to try to answer, **since the intiator peer closes the Socket after reading the Chunk**.

To read and write the chunk content the **ObjectOutputStream** and **ObjectInputStream** classes were used since this guarantees that the content will be written as an object, that is, **it will be written all at once**.

The format of the modified GETCHUNK message is as follows:

```
<protocolversion> GETCHUNK <senderID> <fileID> <chunkNo> <port>
    <CRLF><CRLF>
```

3.3 Delete subprotocol

Problem: A a peer is down and has saved chunks of a file that executed the delete protocol, causing it to store unwanted and needless files, making this space never reclaimed.

Solution: In order to solve that, an improved protocol was created, **using extra messages** to act as a garbage collector of its owns files. Without this, the available memory of a peer can deplete whenever it misses a delete message.

To achieve a reliable enhanced protocol, a new message was created, **namely CHECKDELETE** and new data structures were created.

Everytime it **happens a delete**, the initiator peer **saves the file ID** of the deleted file as well as **when a DELETE message is received**.

When the peer is boot it sends to the multicast a message CHECKDELETE. This messages aims to warn all the peers that this peer may have not deleted

some chunks since by the time that the delete was performed it could have been down.

When a peer receives this message, it will **access a ConcurrentHashMap** with the files that have been removed from the system and **send that information** to the peer who sent the message over a **TCP connection**. This connection is established in the same manner as the restore protocol.

When the peer receives these files they **add them to their deleted files so they can correctly inform other peers** and **delete any chunks they may have saved from the already deleted files**.

The format of CHECKDELETE message is as follows:

```
<protocolversion> CHECKDELETE <serverID> <port> <CRLF><CRLF>
```