Distributed Backup Service

Project 1 - Final Report

Master in Informatics and Computing Engineering

Distributed Systems

Class 6 - Group 08:
Diogo Machado - up201706832
José Pedro Baptista - up201705255

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

14 de Abril de 2020

Conteúdo

1 Introduction

This report was developed in the context of the first assignment of the curricular unit Distributed Systems, of the 2nd semester of the 3rd year of the Master in Informatics and Computing Engineering of the Faculty of Engineering of the University of Porto.

The goal of this report is to specify some of the key aspects of our implementation of the assignment, such as:

- **Compiling and Executing:** The necessary instructions to compile and execute our program, in Windows and Linux.
- **Concurrent Execution of Protocols:** Description of the data structures and mechanisms used to allow the concurrent execution of the different protocols.
- **Implemented Enhancements:** Description of the proposed enhancements that we implemented, as well as an explanation of our implementation.

2 Compiling and Executing

2.1 Compiling Instructions

To compile our project, on both Windows and Linux, you need only change directory to the */src* folder, and compile all *.java* files using javac:

```
cd <base_directory>/src
javac *.java
```

2.1.1 Scripts

- **Make.sh:** compiles the code, preparing it for use;
- **Clean.sh:** deletes the compilation files from the *src/* directory.

2.2 Execution Instructions

To execute our project, there are 3 things you must do:

2.2.1 Starting the RMI Registry

Since this project implements the *RemoteMethodInvocation* (RMI) Java Interface, to be able to use it you must first start the RMI registry, by running:

```
rmiregistry &         // for Linux ...
start rmiregistry     // for Windows ...
```

2.2.2 Running the Peers

To run a Peer from the terminal, you use the following command:

```
java Peer <version> <peer_id> <peer_ap> <mc_address> <mc_port> <mdb_address> <mdb_port>
    <mdr_address> <mdr_port>
```

- **version:** Version of the protocol that the *Peer* uses;
- **peer_id:** unique identifier of a given *Peer*;
- **mc_address:** IP address of the multicast control channel;
- **mc_port:** port of the multicast control channel;
- **mdb_address:** IP address of the multicast data backup channel;
- **mdb_port:** port of the multicast data backup channel;
- **mdr_address:** IP address of the multicast data restore channel;
- **mdr_port:** port of the multicast data restore channel.

For the effects of testing this project, we used the multicast IP Address 224.0.0, since the range [224.0.0.0, 224.0.0.255] is the range of multicast addresses reserved for local networks.

2.2.3 Running the test Application

The command to execute the test *Application* is as following:

```
java Application <peer_ap> <operation> [<operand_1> <operand_2>]
```

- **peer_ap:** access point for the initiator *Peer*;
- **operation:** protocol to be executed;
- **operand_1:** in the case of the protocols BACKUP, RESTORE and DELETE, the relative path of the file to use; in the case of the RECLAIM protocol, the maximum disk space (in KBytes) that the *Peer* can use;
- **operand_2:** only used in the BACKUP protocol, where it specifies the *replication degree*.

2.2.4 Scripts

- **LaunchPeers.sh:** opens N terminals, and launches a *Peer* in every terminal;

```
sh LaunchPeers.sh <n_peers_to_launch> <peer_protocol_version>
```

- **Backup.sh:** opens a terminal and starts a Backup (through Application.java) for a given file, with a given desired replication degree, to a given *Peer*;

```
sh Backup.sh <peer_id> <file_path> <desired_replication_degree>
```

- **Restore.sh:** opens a terminal and starts a Restore (through Application.java) of a given file in a given *Peer*;

```
sh Restore.sh <peer_id> <file_path>
```

- **Delete.sh:** opens a terminal and sends a Delete message (through Application.java), of a given file, to a given *Peer*;

```
sh Delete.sh <peer_id> <file_path>
```

- **Reclaim.sh:** opens a terminal and starts a Reclaim process (through Application.java), of a given *Peer*'s memory;

```
sh Reclaim.sh <peer_id> <desired_space_usage>
```

- **State.sh:** opens a terminal and asks for the state of a given *Peer* (through Application.java); Stays open 10 seconds, in order for the user to be able to read the requested info.

```
sh State.sh <peer_id>
```

3 Concurrent Execution of Protocols

In order to ensure that the protocols could be executed concurrently, to a good degree of efficiency, we had to take a few factors into account. These factors include the number of simultaneous messages being sent by the various *Peers*, the synchronization of "common"resources across the *Peers* (e.g. the information about the occurrences of the backed up files), and the fact that a *Peer* can possibly have to save, delete and send files, all at the same time.

The first measure we took to implement the concurrent execution was the use of threads. For that, we used the Java class *ScheduledThreadPoolExecutor*, which was discussed in the classes relative to the labs (namely, Lab02). This class provides a simple interface through which to implement *multi-threading* solutions, since it makes all the lower level mechanisms available for high level usage. In addition to this, the fact that it "recycles"threads, thus reducing the amount of resources spent on the creation and destruction of threads, also helps to improve the efficiency of our implementation. Furthermore, this Java class limits the maximum number of threads that it can schedule/execute, meaning that we mustn't concern ourselves with the case that the number of concurrent threads becomes too large, as that situation is handled by the implementation of the class.

Our *multi-threading* implementation follows the following structure:

- When a protocol is invoked by the application, it calls the initiator *Peer*'s function that implements the protocol;
- For all protocols (with the exception of STATE), a thread (implemented on our class *MessageSender*) is started, which receives the message to be sent and the multicast channel to which it must be sent, and sends it (the fact that each message is sent on a new thread of this kind allows for multiple protocols to be started);
- Whenever a peer receives a message, it starts a new thread (implemented on our class *MessageReceiver*) which handles the message, interpreting the type of message received and starting a new thread that handles that specific message type.

When it comes to the data structures used to store the information on the *Peers*, we chose to use the Java class *ConcurrentHashMap*, instead of the standard *HashMap*. The reason for this is that, as indicated by the name, *ConcurrentHashMap* is designed to work correctly when used/accessed by several threads, as opposed to *HashMap*, which doesn't guarantee this.

4 Implemented Enhancements

In this assignments, enhancements were suggested for the protocols BACKUP, RESTORE and DELETE. Of there, we successfully implemented the ones relative to the RESTORE and DELETE protocols.

4.1 Restore Protocol

Suggested Enhancement: *If chunks are large, this protocol may not be desirable: only one peer needs to receive the chunk, but we are using a multicast channel for sending the chunk. Can you think of a change to the protocol that would eliminate this problem, and yet interoperate with non-initiator peers that implement the protocol described in this section? Your enhancement* ***must use TCP*** *to get full credit.*

Implementation

For this enhancement, we made it so that the chunk transfer would be made using TCP, and only one peer (the initiator one, or the one that sent the GETCHUNK), would receive the chunk data. In order to achieve that, we made a few changes to the CHUNK message:

```
// Protocol version 1.0
<version> CHUNK <sender_id> <file_id> <chunk_no> <CRLF><CRLF><body>
// Protocol version 2.0
<version> CHUNK <sender_id> <file_id> <chunk_no> <CRLF><CRLF><ip_address> <port>
```

With this improved version, after a peer (that isn't the initiator one) receives a GETCHUNK message, checks if it has the requested chunk stored, and if it does, waits a random time interval (0-400ms) and if after that time there was no recent (in the last 400ms) CHUNK message for the same chunk sent by another peer, it creates a ServerSocket, sends a CHUNK message and proceeds to write the requested chunk data to the socket. When the initiator peer receives the CHUNK message with the IP Address and Port for connection, creates a Socket using the information read, and reads the message that the non-initiator peer sent. The message has the following format:

```
// Message written by the non-initiator peer
<chunk_data_size> <chunk_data>
```

After reading this message, the initiator peer checks if ***chunk_data_size*** is equal to the length of ***chunk_data***. If it is, saves ***chunk_data***, and proceeds to request the next file chunk. Otherwise, it is discarded, and the GETCHUNK message is resent (up to 5 times). After every chunk is received the file is restored. This enhancement will only be applied if both peers (initiator and non-initiator) use the protocol version 2.0, otherwise 1.0 protocol will be used. This allows peers that are using different protocol versions to still communicate and restore the requested file flawlessly.

4.2 Delete Protocol

Suggested Enhancement: *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, the space used by these chunks will never be reclaimed. Can you think of a change to the protocol, possibly including additional messages, that would allow to reclaim storage space even in that event?*

Implementation

For the implementation of this enhancement, we took an approach that would work not only locally, but also remotely. For this, we added two new message types to the protocol:

```
// HELLOWORLD message
<version> HELLOWORLD <sender_id> <CRLF><CRLF>
// DELETEDFILE message
<version> DELETEDFILE <sender_id> <file_id> <CRLF><CRLF>
```

- **HELLOWORLD:** This message is sent by a *Peer* whenever it is started, in order to alert other *Peers* that it is running;
- **DELETEDFILE:** This message is sent by a *Peer* whenever it deletes a file's chunks, by consequence of the DELETE protocol, and informs the remaining *Peers* that the sender has deleted the respective file.

With these new messages, the implementation follows the following logic:

- When the DELETE protocol is invoked on one of the active *Peers*, it saves the ID of the file to be deleted on a data structure (as well as a file, so that this information is not lost in case the *Peer* is restarted);
- Then, whenever the other *Peers* receive the DELETE message, they now send an *ACK-type* message (DELETEDFILE) to confirm to the initiator *Peer* that they have deleted the file;
- On receiving this message, the initiator *Peer* updates the information on the occurrences of the chunks of the specified file, only giving the operation as completed when all the occurrences of the file are deleted;
- In the case that a non-active *Peer* has chunks of the file, when it is started it will alert the other *Peers* with a HELLOWORLD message. Upon receiving this warning, the initiator *Peer* for the DELETE protocol will send the DELETE message once again, until one of two things happens:
  - All registered occurrences of the file are deleted (i.e. the initiator *Peer*'s perceived replication degree for all chunks of that file becomes zero);
  - The BACKUP protocol is called for that same file (whether it be by the initiator *Peer*, or by another one).

This ensures that, whenever the initiator *Peer* is active and there are still occurrences of the file to delete, they will be eliminated as soon as the *Peers* that hold them become active.