Distributed Backup Service

Project 1

Hugo Miguel Monteiro Guimarães - up201806490
Paulo Jorge Salgado Marinho Ribeiro - up201806505

Work carried out within the scope of
Course Unit Distributed Systems

Master in Informatics and Computing Engineering
Faculdade de Engenharia da Universidade
do Porto
11th march 2021

Contents

1 Concurrency Design

This section analyzes our implementation of the concurrent execution of instances of the required protocols. Both versions of the developed program allow simultaneous executions of these protocols.

When a peer starts running, the class PeerInitializer begins by storing the input parameters.

The peer information is organized in the class DataStored, which contains several thread-safe ConcurrentHashMaps and CopyOnWriteArraySets. The information inside this class is maintained through multiple program executions, since we serialize its data on program shutdown, saving it in a file. This data is then read as soon as the peer starts running.

On invocation, the class Peer will also be responsible for initializing the three multicast channels:

- MulticastControlChannel, used for control messages.
- MulticastDataChannel, used for backing up file chunk data.
- MulticastDataRecovery, used for restoring file chunk data.

```
1  // Initialize mc channel
2  controlChannel = new MulticastControlChannel(mcAddr, mcPort,
       ↪ peerID);
3  // Initialize mdb channel
4  backupChannel = new MulticastDataChannel(mdrAddr, mdrPort,
       ↪ peerID);
5  // Initialize mdr channel
6  restoreChannel = new MulticastDataRecovery(mdbAddr, mdbPort,
       ↪ peerID);
7
8  executor.execute(controlChannel);
9  executor.execute(backupChannel);
10 executor.execute(restoreChannel);
```

Each of these classes extends the class MulticastChannel, which implements the Java Runnable Interface, to execute code on a concurrent thread. This interface requires the implementation of the method run(), with void as return type, by the instances of that class. This way, each channel is executed as a single thread, by an object of class ScheduledThreadPoolExecutor with a core pool size of 250 threads, to avoid the invocation overhead of creating a new thread for each execution needed. When they are executed, the MulticastChannel run() method will be called, which consists of an infinite while loop, whose function is to receive data packets corresponding to the messages to be interchanged by the peers.

Whenever a message is received, the peer's ScheduledThreadPoolExecutor starts a new thread of the class MessageHandler, which also implements the interface Runnable.

In this class, the message received will be parsed by the class MessageParser, in other words, the message components will be divided, to be used in the future. One of those components is the sender ID, which is compared to the ID of the peer that received the message, to ignore self-messages.

Another component is the message type, used to decide what protocol will be executed. Our program has protocols defined for the messages of types PUTCHUNK, STORED, GETCHUNK, CHUNK, DELETE, REMOVED, DELETED, and HELLO (the last two are only executed on the enhanced version, 2.0). Despite this, every other type is accepted but ignored, for interoperability purposes.

```
1  @Override
2  public void run() {
3      byte[] msgReceived = new byte[65507];
4      try {
5          while(true) {
6              // Waiting to receive a packet
7              DatagramPacket requestPacket = new DatagramPacket(
                   ↪ msgReceived, msgReceived.length);
8              this.multicastSocket.receive(requestPacket);
9              byte[] realData = Arrays.copyOf(requestPacket.
                   ↪ getData(), requestPacket.getLength());
10             Peer.executor.execute(new MessageHandler(realData,
                   ↪ this.peerID, requestPacket.getAddress()));
```

```
        }
    } catch(Exception e) {
        e.printStackTrace();
        System.out.println("Error receiving message from
            ↪ Multicast Data Channel (MDB)");
    }
}
```

Finally, the majority of the functions responsible for handling each message type launch a new thread that will execute the requested protocol. This is done using the method schedule of the ScheduledThreadPoolExecutor, which initializes the desired thread only after the given delay. We followed the project's guide advice of using a random delay uniformly distributed between 0 and 400 milliseconds. That way, we lower the probability of the protocol being executed at the same time by different peers, avoiding "collisions" and assuring that the many peers that received the same message start and finish the required protocol at different times.

One example of this is the launching of the thread ChunkThread, while handling a GETCHUNK message, which will be responsible for sending the data of the chunk requested if the peer that received the message has a local copy of that chunk.

A different thread is also launched every time we need to send a message to the multicast channels, which will execute the function sendMessage() of the MulticastChannel class, after a random delay.

```
void sendMessage(byte[] message) {
    try {
        DatagramPacket packetSend = new DatagramPacket(message,
            ↪  message.length, this.addr, this.port);
        this.multicastSocket.send(packetSend);
    } catch(Exception e) {
        e.printStackTrace();
        System.out.println("Error sending message to Multicast
            ↪ Data Channel (MDB)");
    }
}
```

On the client-side, the TestApp class starts by parsing the command line arguments and requesting the desired protocol. This is done by calling the corresponding function of the PeerProtocol object of the peer with the RMI object name given as input to the TestApp. After this, the function will start a new thread, responsible for executing that protocol, in the vision of the initiator peer.

Finally, during some protocols, some new threads are raised to execute some minor tasks at the same time that the main protocol is occurring. The class PutChunkThread is an example of these calls, since it happens during the backup protocol, and for each chunk created during this protocol. Its purpose is to send the PUTCHUNK messages of each chunk created. So, as the project guide suggests, the initiator will send at most 5 PUTCHUNK messages per chunk, in the case of not receiving the desired confirmation messages (replication degree), and doubles the delay between each resend. In short, this mechanism is done by this class PutChunkThread, using an independent thread for each chunk to backup.

Another example of these calls is the class GetChunkThread, called upon receiving all the chunks of the file to restore, which creates the file copy.

2 Enhancements

We have implemented all required enhancements, which will be described in the following subsections.

To run the enhanced version of the project, each peer must be initialized with the version argument 2.0.

Example of initialization of one peer:

```
..\scripts\peer.sh 2.0 1 Peer1 225.0.0.1 8000 225.0.0.1 8001
    ↪ 225.0.0.1 8002
```

2.1 Backup Enhancement

Since the 1.0 version of the Backup protocol only verifies the replication degree before sending a PUTCHUNK message via UDP Multicast, all the peers that receive that message will store the chunk.

This scheme is problematic since it depletes the backup space rather quickly by storing unnecessary additional chunks. We have solved this problem using a simple and efficient method.

Since all peers are aware of the replication degree of every chunk, via our chunkRepDegrees ConcurrentHashMap, which is updated in the STORED message handler, we can verify if a given chunk is already being stored by another Peer. Hence, we have fixed this issue by prohibiting the storage of a new chunk if the replication degree has already been fulfilled.

```
1 if(Peer.getVersion().equals("2.0")) {
2     int chunkRepDeg = getChunkReplicationNum(chunkID);
3     int desiredRepDeg = chunk.getDesiredReplicationDegree();
4     if (chunkRepDeg >= desiredRepDeg) {
5         System.out.println("Chunk " + chunkID + " already
              ↪ fulfilled repDegree. Ignoring chunk...");
6         return;
7     }
8 }
```

Although this method is simple, it is not perfect. Given many concurrent threads are running, attempting to back up the chunk, there are read and write operations constantly being issued to the hashMap. Even though the data structure is thread-safe, there is no guarantee that the number of chunks stored will be the same as the desired replication degree every time.

There is a possible way to fix this problem, by resending the protocol until the replication degree value is the same as the desired, deleting exceeding chunks.

Although the aforementioned procedure would present as a solution, we have decided to not implement it, as it would flood the multicast channels with several other messages without significantly reducing the space depletion problem.

2.2 Restore Enhancement

This enhancement was designed to avoid sending all the file chunks through a UDP Multicast channel. In the 1.0 version the UDP protocol is unreliable and there are too many messages being continuously sent through the multicast channel, and most of them would be ignored by every peer except the initiator, which is the only one that needs the chunk data to restore the file.

To solve this problem, we came up with a solution by using the TCP protocol to exchange the body of the message between the peer that contains the chunk and the initiator peer requesting the protocol.

First, we initialized the TCPHandler thread, which is blocked in a while loop, launching a ClientHandler thread whenever there is a Client Socket starting the TCP connection.

```
1 public void run() {
2     while(true) {
3         try {
4             new ClientHandler(serverSocket.accept()).start();
5         } catch (IOException e) {
6             e.printStackTrace();
7         }
8     }
9 }
```

The ClientHandler thread reads a CHUNK message through the established TCP connection and restores the file as soon as all the chunks have been received, by launching a GetChunkThread.

The establishment of the TCP connection is made for each chunk after a Peer that has a local copy of that chunk receives a GETCHUNK message and initializes a ClientSocket.

To implement this enhancement, we have changed the header of the GETCHUNK subprotocol by adding an extra line containing the ServerSocket port of the initiator Peer. This port is essential to establish the TCP connection, as well as the IP address used for the server socket, which can be accessed via the UDP Multicast DatagramPacket getAddress() method.

Even though the body of the message is now being sent through a TCP connection to the initiator peer, the multicast channel is still used to send the header of the CHUNK message. Such is necessary because all peers must update their storage to be able to know if the chunk has already been sent by another peer, preventing replicated CHUNK messages.

2.3 Delete Enhancement

In the 1.0 version of our project, 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.

This enhancement aims to fix that problem, by creating two new messages, DELETED and HELLO.

The fist one has this structure:

```
1  <Version> DELETED <SenderId> <FileId> <CRLF><CRLF>
```

This message works as a confirmation of the DELETE message, being sent after a peer successfully removes the chunks of the file requested by the DELETE message.

This confirmation message allows us to update our deletedFiles CopyOnWriteArraySet that contains the IDs of the files that have not yet been fully deleted by all other peers.

We have also created the HELLO message, whose structure is:

```
1  <Version> HELLO <SenderId> <CRLF><CRLF>
```

This message is sent by every Peer via Multicast as soon as it starts running. Subsequently, all peers receiving this message will check the deletedFiles CopyOnWriteArraySet and initialize the DELETE subprotocol for every file that is yet to be fully deleted.

This way, we have managed to remove the chunks that couldn't be fully deleted because a peer with a local copy of it was not running.