Distributed Backup Service

Final Report

Integrated Masters in Informatics and Computing Engineering

Distributed Systems

Grupo T6G09:
César Alves Nogueira - up201706828@fe.up.pt
Gonçalo Marantes - up201706917@fe.up.pt

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

April 14, 2020

Abstract

This project consists of a distributed backup service for a local area network (*LAN*) and was developed using communication protocols, concurrency and synchronization methods learned in the Distributed Systems Course. To achieve the desired performance and methodology we used mainly *UDP* communication, but also *RMI* and *TCP* protocols. Our concurrent approach consists of one thread per Multi-cast Channel and many protocol instances processing requests at a time.

Contents

1 Introduction

In this project we developed a distributed backup service for a local area network (LAN). The idea is to use the free disk space of the computers in a LAN for backing up files in other computers in the same LAN. The service is provided by servers in an environment that is assumed cooperative. Nevertheless, each server retains control over its own disks and, if needed, may reclaim the space it made available for backing up other computers' files.

The main goal of this report is to clarify the key aspects of our implementation, such as:

- **Project Structure:** Overall project file structure and java packages choice explanation.
- **Compilation and Execution Instructions:** Necessary instructions for compiling and correctly executing the developed program in both *Windows* and *Linux* environments. For faster testing and executing we provide auxiliary scripts that help on that matter.
- **Concurrent Protocol Execution:** Detailed description of methodologies, synchronization methods and data structures used which allow the concurrent execution of different protocols.
- **Implemented Enhancements:** Detailed description of solutions and implementation.

2 Project Structure

Our project is divided into 4 folders:

- *docs/:* Contains documents regarding this project, such as documentation provided by the professor and this report.
- *scripts/:* Contains *bash* and *batch* scripts to help compile, test and execute the program in different operating systems.
- *src/:* Contains the source code developed during this project.
- *test/:* Contains files to be used by the scripts or manually in order to test the program.

Regarding the *src/* folder, we decided to organize it in packages, each of which are responsible for various key aspects of our implementation:

- *channel:* Responsible for communication and request dispatching.
- *exceptions:* Contains custom exceptions for better error handling and code quality.
- *peer:* Contains the main methods for starting sub protocol actions.
- *storage:* Contains classes responsible for information representation and file system manipulation.
- *threads:* Contains all the threads used by peers to perform actions and achieve concurrency.
- *utils:* Contains useful methods and constants used by other packages.

3 Compilation and Execution Instructions

3.1 Compilation

In order to compile the source code, we provide both a *Makefile* for *Linux* and a *Batch* file of *Windows*. However for *Linux* the scripts we have developed already take care of most of the work, including starting the *RMI* Registry, hence the user simply needs to run the following command, depending on the Operating System:

```
# linux
sh scripts/setup.sh # compiles source code and starts rmi registry
# windows
scripts/setup # compiles source code
cd src/ # go to source folder
start rmiregistry # starts rmi registry inside source folder
```

Listing 1: Executing Compilation Scripts

3.2 Execution

Finally, there are only 2 steps required to execute the program:

Executing *MCastSnooper.jar* (Optional)

To execute the provided *MCastSnooper* program, it is only necessary to run the following script:

```
# linux
sh scripts/run_mcastsnooper.sh
# windows
scripts/run_mcastsnooper
```

Listing 2: Executing *MCastSnooper* Scripts

Executing Peers

The necessary command to execute a Peer (once inside the *src* folder) is the following:

```
java peer.Peer <protocol_version> <peerID> <peer_access_point> <
    MCaddress> <MCport> <MDBaddress> <MDBport> <MDRaddress> <
    MDRport>
```

Listing 3: Executing *Peer* Program

- **protocol_version:** Protocol version ("1.0" for standard protocol, "2.0" for enhanced protocol)
- **peerID:** Peer's unique identifier.
- **peer_access_point:** Access point for *RMI* object.
- **MCaddress:** IP address for Multi-cast Control channel[1].
- **MCport:** Port for Multi-cast Control channel.
- **MDBaddress:** IP address for Multi-cast Data Backup channel.
- **MDBport:** Port for Multi-cast Data Backup channel.
- **MDRaddress:** IP address for Multi-cast Data Recovery channel.
- **MDRport:** port for Multi-cast Data Recovery channel.

However, to execute a desired number of peers without going through this much trouble we provide scripts that execute various peer programs. To run these scripts one must only execute the following commands:

```
# linux
sh scripts/run_peer.sh <protocol_version> <num_peers>
# windows
scripts/run_peer <protocol_version> <num_peers>
```

Listing 4: Executing *Peer* Program Scripts

- **protocol_version:** Protocol version ("1.0" for standard protocol, "2.0" for enhanced protocol)
- **num_peers:** Number of peers that are going to execute.

[1]IP multi-cast addresses used should be between *224.0.0.0* and *224.0.0.255*, *i.e* the reserved *IPv4* multi-cast addresses for local sub-networks.

Executing *TestApp*

To execute the client program (once inside the *src* folder) it is necessary to run the following command:

```
java TestApp <peer_ap> <sub_protocol> [ <opnd_1> | [ <opnd_2> ] ]
```

Listing 5: Executing *TestApp* Program

- **peer_ap:** Access point for *RMI* object.
- **sub_protocol:** Operation that the peer must execute (BACKUP, RESTORE, DELETE, RECLAIM).
- **opnd_1:** Can be either the directory of the desired file (BACKUP, RESTORE and DELETE sub-protocols). Or the maximum amount of disk space (in *KBytes* the peer can use (RECLAIM sub-protocol).
- **opnd_2:** Used in the BACKUP protocol to specify the desired replication degree of the backed up file.

Once again, to simply this process, a test script is provided for faster testing:

```
# linux
sh scripts/run_test.sh
# windows
scripts/run_test
```

Listing 6: Executing *TestApp* Program

4 Concurrent Protocol Execution

To guarantee concurrency and given that peers can "share" resources, *i.e.* these resources are not shared but require consistency in every peer so they must be exactly the same. Also, inside a protocol there are many messages being sent and received at the same time. And finally, a peer can store and send various file chunks at the same time.

The first step to achieve concurrency was to require the use of *threads*, and in order to coordinate thread execution we decided to use the *java.util.concurrent API*. This *API* provides classes such as *ScheduledThreadPoolExecutor*, which makes it easy to execute and schedule thread execution. This class also recycles threads, this makes the program have a faster performance due to the fact that creating and destroying threads is a rather heavy process. Scheduling threads also made it possible, in most cases, to avoid the use of *Thread.sleep()*.

Our *multi-threading* design is as follows:

- When a peer is initialized, it executes 3 *Threads*, 1 per multi-cast channel. These *Threads* are responsible for sending and receiving data from their respectful channel. When receiving a message, they dispatch it to the *MessageReceiverManagerThread* class, which analyses the message type and dispatches the request to the appropriate *Thread* to handle it. This means that we can have multiple *Threads* handling requests as they arrive, achieving concurrency.

```java
@Override
public void run() {
    // Create a new Multicast socket that will allow
    // other programs to join it as well.
    try (MulticastSocket socket = new MulticastSocket(this.
    port)) {
        // Join the Multicast group.
        socket.joinGroup(this.address);
        // Buffer of bytes to store the incoming bytes
        // containing the information from some other peer.
        // Since the message includes:
        //  - A header: less 100 bytes
        //  - A Body: Max of 64KB = 64 000 bytes
        // A buffer of size 65KB is enough
        byte[] buffer = new byte[65 * 1000];
        // Listen for messages
        while (true) {
            // Receive packet
            DatagramPacket packet = new DatagramPacket(buffer,
     buffer.length);
            socket.receive(packet);
            // Clean buffer and dispatch to peer's message
    receiver managing thread
            byte[] message = Arrays.copyOf(buffer, packet.
    getLength());
            this.peer.getScheduler().execute(new
    MessageReceiverManagerThread(message, this.peer));
        }

    } catch (IOException e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
}
```

Listing 7: Channel Class *run* Method

```java
@Override
public void run() {
    // get message type
    String messageType = new String(this.message).split(" ")
    [1];
    // pick a random waiting time between o and 400 ms
    int waitingTime = Utils.getRandomNumber(0, 400);

    // dispatch message request to working threads
    switch (messageType) {
        case "PUTCHUNK":
            this.peer.getScheduler().schedule(new
    PutChunkThread(message, this.peer), waitingTime, TimeUnit.
    MILLISECONDS);
            break;
        case "STORED":
            this.peer.getScheduler().execute(new StoredThread(
    message, this.peer));
            break;
        case "GETCHUNK":
            this.peer.getScheduler().execute(new
    GetChunkThread(message, this.peer));
            break;
        case "CHUNK":
            this.peer.getScheduler().execute(new ChunkThread(
    message, this.peer));
            break;
        case "DELETE":
```

```
            this.peer.getScheduler().execute(new DeleteThread(
    message, this.peer));
            break;
        case "DELETEACK":
            this.peer.getScheduler().execute(new
    DeleteAckThread(message, this.peer));
            break;
        case "AVAILABLE":
            this.peer.getScheduler().execute(new
    AvailableThread(message, this.peer));
            break;
        case "REMOVED":
            this.peer.getScheduler().execute(new RemovedThread
    (message, this.peer));
            break;
    }
}
```

Listing 8: *MessageReceiverManagerThread* Class *run* Method

- In every protocol that requires communication between peers, *i.e.* excluding STATE, the *MessageSenderManagerThread* class is responsible for sending the desired message to the respectful multi-cast channel. Since we create a new instance of *MessageSenderManagerThread* with the *ThreadPoolExecutor*, we have achieve protocol concurrency when sending various messages.

```
@Override
public void backup(String path, int replication) throws
    RemoteException {
    StorageFile file = new StorageFile(path, replication);
    this.storage.addFile(file);

    ArrayList<Chunk> chunks = file.getChunks();

    for (Chunk chunk : chunks) {
        byte[] message = generatePutChunkMessage(chunk, this.
    protocolVersion, this.peerID, chunk.getFileID(), chunk.
    getNumber(), replication);
        this.scheduler.execute(new MessageSenderManagerThread(
    message, "MDB", this));
        this.scheduler.schedule(new
    ConfirmationCollectorThread(this, message, replication),
    1, TimeUnit.SECONDS);
    }
}
```

Listing 9: *Peer* Class *backup* Method

```
@Override
public void run() {
    System.out.println(this.getInfo());
    this.peer.getChannel(this.channelKey).sendMessage(this.
    message);
}
```

Listing 10: *MessageSenderManagerThread* Class *run* Method

- Thanks to the *ScheduledThreadPoolExecutor* we do not need to worry about the number of running *Threads*, due to the fact that this number is limited and handled by the Java *API*.

- For storing *Chunks* and other information accessed by multiple *threads*, such as channel responsible threads, we decided to use *ConcurrentHashMap*. The main advantage of using this type of *HashMap* is the assurance of correct behavior when being accessed by multiple threads.

5 Implemented Enhancements

According to this project's specifications we had the opportunity to upgrade the specified version of the BACKUP, RESTORE and DELETE sub-protocols. This section will explain how we implemented this enhanced versions of said sub-protocols.

5.1 Restore sub-protocol

5.1.1 Proposed Enhancement

The main problem with the initial version of this sub-protocol is that we are using a multi-cast channel for sending the requested chunk, because only one peer needs to receive it. The implemented enhancements uses the *TCP* communication protocol for a direct link between the initiator and the replying peer.

5.1.2 Implementation

When the initiator peer starts the sub-protocol it sends *GETCHUNK* messages and in return hopes to receive *CHUNK* messages. For this it initializes a communication socket where the replying peer can send *CHUNK* messages. And also a new thread created with *TCPChunkThread* class that "listens" for messages on the newly created socket.

After that all *GETCHUNK* messages are sent, when a peer that is storing the requested file chunks receives one of these messages, they start a new thread with the help of *TCPGetChunkThread* class. This thread is responsible for replying with the requested chunk via the created *TCP* channel.

To avoid duplicate chunks being sent by two distinct peers, the initiator peer first receives a request from other peers "asking" if the requested chunk was already received. This is done by first the replying peer sending the requested chunk's number, upon receiving this number the initiator peer checks to see if that chunk has been received from another peer, and it responds with a Boolean value accordingly. After receiving this response, the replying peer either sends the requested chunk or does nothing.

This process is repeated until the initiator peer receives all requested chunks.

5.2 Delete sub-protocol

5.2.1 Proposed Enhancement

The problem with the original approach is that 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. To solve this issue we have created two new messages which are described bellow.

5.2.2 Implementation

The new messages we have created are the following:

- *ANNOUNCE:* This message is sent every time a peer is initiated, so every peer know which peers are online.
- *ACKDELETE:* This message is sent when a peer successfully deletes the requested file's chunks, because of this acknowledgment the initiator peer can keep track of which peers have successfully deleted the requested file.

With the use of these new messages, even if a peer that is storing some chunks related to the specified deleted file, when it comes online it will announce itself. After receiving this *ANNOUNCE* message, the initiator peer can re-send the *DELETE* message, which will be handles by the just online peer, and in return will reply with a *ACKDELETE* message.

6 Conclusion

Having completed this project we have learned a lot about thread-based concurrency using the Java *API* but most importantly message exchanging using multiple communication protocols.

References

[1] docs.oracle.com. (2020). ConcurrentHashMap (Java Platform SE 8). [online] Available at: https://docs.oracle.com/javase/8/docs/api/java/util /concurrent/ConcurrentHashMap.html

[2] docs.oracle.com. (2020). ScheduledThreadPoolExecutor (Java SE 9 & JDK 9) [online] Available at: https://docs.oracle.com/javase/9/docs/api/java/util /concurrent/ScheduledThreadPoolExecutor.html

[3] baeldung.com. (2020). SHA-265 Hashing in Java — Baeldung [online] Available at: https://www.baeldung.com/sha-256-hashing-java

[4] stackoverflow.com. (2019). How do I programmatically determine operating system in Java? - Stack Overflow. [online] Available at: https://stackoverflow.com/questions/228477/how-do-i-programmatically-determine-operating-system-in-java

[5] tutorialspoint.com. (2020). Java - Serialization - Tutorialspoint. [online] Available at: https://www.tutorialspoint.com/java/java_serialization.htm