Distributed Systems - Second Project

Distributed Backup Service for the Internet - Report

Gonçalo Teixeira	**201806562**
André Gomes	**201806224**
Isla Cassamo	**201808549**

Group G27

BSc + MSc on Informatics and Computing Engineering

U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

June 2, 2021
Porto

Contents

Chapter 1

Overview

The main goal for this project is to upscale the first project to work over the internet. Given this constraint the transactions should be made with encryption providing security and confidentiality, and the system should be scalable and fault tolerant.

Regarding security and confidentiality we've decided to use JSSE's SSLEngine as it provides an interface to work with NIO's ByteBuffers allowing the peer's to be non-blocking, and thus enhancing the system performance and scalability. With security left aside, to improve the scalability and fault tolerance we've implemented a Chord network for the peers to be able to connect to other peers, the Chord Network provides a way to locate resources and peers over the circle, while storing just M_BIT peer addresses.

Regarding the application's features, it was natural for us to take the full advantage of the NIO's methods to serve the full files instead of its chunks, this allows for a big enhancement in terms of performance, for the sake of exemplification, in the tests we've run, we were able to backup a 3GB file with an average transfer rate of 250MB/s (using an SSD and the same PC for the peers), meaning it took around 12 seconds to transfer the file, which in comparison to the first project, sending a 3GB file would be overwhelming. The java's NIO also allows us to start multiple backup tasks concurrently, which is very appropriate given the replication degree constraint which we have carried out from the first project.

The chord implementation was based on a paper [2] which can be found here.

Chapter 2

Protocols

The client stub for the user to test and start operations on the system was developed taking advantage of the RMI features. The client starts a request, and for every request, it awaits a reply. In terms of architecture, the client, besides the peer, is also a `Remote` object, allowing the called peer to able to send a message back to the client. Bellow we present a table containing the supported protocols/operations, the required options for each operation and the expected output.

Protocol	Arguments	Output
BACKUP	<filepath> <replication degree>	Operation Result
RESTORE	<filepath>	Operation Result
DELETE	<filepath>	Operation Result
RECLAIM	<space:bytes>	Operation Result
STATE	–	Peer Internal State
CHORD	–	Chord Network Information
LOOKUP	<key>	Chord Reference with <key> successor

Table 2.1: Remote Interface Protocols

On the next sections we will explain each protocol with some detail, leaving the details on the transport layer to the JSSE chapter.

2.1 Backup Protocol

The backup protocol starts by checking if the peer is already serving the file, and if that proves to be true the backup is canceled.

If the file is not being backed up already, the peer generates $4 * replication\_degree$ keys, which will be used to discover potential peers to store the file, these keys are distinct and between 0 and $2^M - 1$ (the reason for this relates to the Chord architecture [2], which will be explained further on). The peer then loops the generated keys in order to find peers which can store the file, stopping when the number of peers reaches the replication degree or there are no more keys to test.

After the discovery is completed, the peer starts a backup task for each target peer with a key assigned, concurrently. Bellow we present two sequence diagrams: one with a successful backup task, and another with a failed backup task.

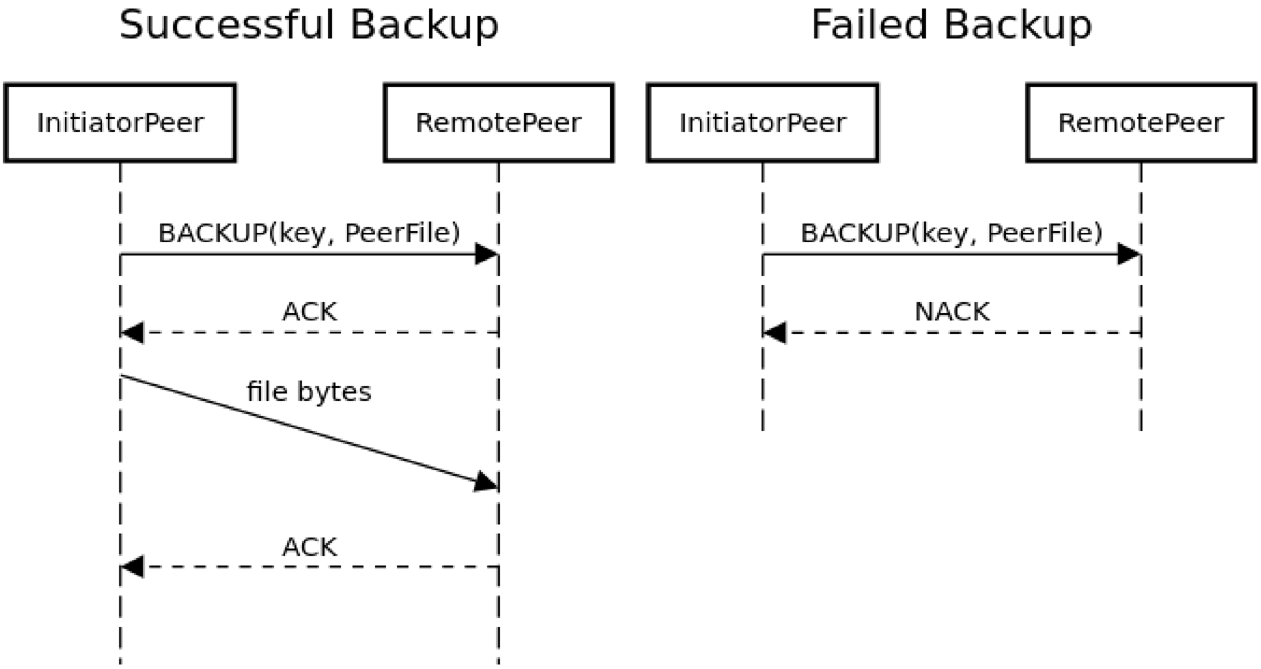

Figure 2.1: Backup Sequence

We've left out the handshakes and `close_notify` exchanges on purpose, as we will discuss them further on, with this in mind, every diagram on this chapter will leave the SSL/TLS properties aside, focusing on the application messages.

The BACKUP message (`messages.application.Backup`), takes the file's details (key, ID, owner, size and replication degree) which the target peer will use to receive the file. After receiving a BACKUP message, the peer will determine if the file is already being saved and if it is not, it will calculate if it has enough space to store the file. If the peer cannot backup the file it will send a NACK (`messages.application.Nack`) message with "HAVEFILE" or "NOSPACE", signaling the reason for the failure. If the peer can store the file it will send an ACK (`messages.application.Ack`) message, signaling everything is OK. The initiator peer will then proceed to send the file using the NIO's `FileChannel`. After sending the file it will wait for an ACK confirmation, signaling the remote peer has received the file without problems. The initiator peer can now close the connection as no other messages are expected to be sent/received.

After every concurrent task is finished it will send the report back to the client, using a Callback Interface. The client receives a report with which peers are storing the file and/or if there were any problems backing up the file. The file backed up metadata is stored on the initiator peer alongside the associated keys. The target peers store the file under their own root folder.

It may happen the peer cannot fulfill the desired replication degree, we've considered the option to generate keys while the replication degree has not been fulfilled yet, but we've come the conclusion it would add a potentially large overhead to the application. By generating four times the replication degree keys we assume the replication degree will be reached, or it will come very close to it. Regardless, we inform the user of that event, and the user can start yet another backup operation.

The last thing we have to mention regarding the backup, or more specifically, the process of receiving files, is that the remote peers are required to cancel the key associated with the connection when they receive a BACKUP message, the reason for this is because when the key is readable, the selector will try to process a message, and there will be no message available when the file is being transferred. This led us to implement this strict sequence of messages, so that the peer can process the bytes read from the socket with guaranty that it will not try to parse a message when there is no message to be parsed.

The source code for the backup implementation can be found under these files:

- peer.Peer:143
- operations.application.BackupOp

2.2 Restore Protocol

The restore protocol starts by checking if the peer has the file backed up, if that proves to be false the restore is canceled as the peer has no way of getting the file.

If the peer has the file backed up, it will pull the keys associated with the file when the backup was processed, these keys' successors are (hopefully) storing the file, so the peer will loop through the keys and try to get the file.

For each key, it will lookup that key's successor, if the successor is found, it will start the restore protocol which, if it is successful, it will terminate the restore protocol and send the report to the client, but if the protocol fails it will pass on to the next key.

Bellow, a diagram representing the sequence of messages regarding the restore protocol is presented.

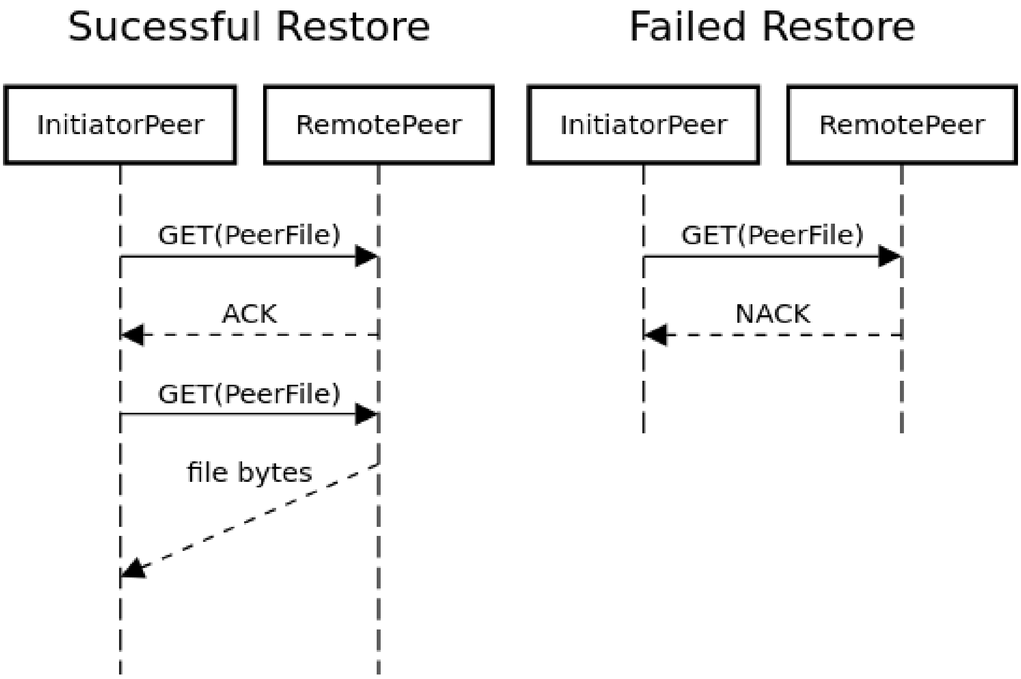

Figure 2.2: Restore Sequence

The initiator peer sends a GET (`messages.application.Get`) message, containing the target file ID it wishes to retrieve, the target peer replies with an ACK message if it can send the file or a NACK message if it cannot send the file because it is not storing it or for other unknown reason. If the peer can send the file it will expect a second GET message signaling it can start sending the file to the initiator peer. After the initiator peer has received the file it can close the connection and return the result to the client.

The file is received with a NIO FileChannel, in order to take full advantage of this library, enhancing the performance of the system.

It is worth mentioning that, since this operation (like the backup) requires more than two messages (request/reply) we need to cancel the key so we can control the flow of the task, with that in mind we've designed this sequence of messages.

The restored file is saved under the peer's root directory with a "restored_" prefix, so the user can know that this file is the product of a restore protocol.

The source code for the restore implementation can be found under these files:

- peer.Peer:358
- operations.application.GetOp

2.3 Delete Protocol

The delete protocol is very similar to the first project: when an initiator peer receives a request to delete a given file, it will send a DELETE (`messages.application.Delete`) message to the peers saving the file. To discover which peers are saving the file, the initiator peer loops through the file's associated keys, looking for the successor of each key.

The delete protocol, or more precisely, the DELETE message, requires a reply with REMOVED (`messages.application.Removed`), to signal the remote peer has removed the given file. The owner can now remove the key, as it is no longer available on the system.

The source code for the delete implementation can be found under these files:

- peer.Peer:403
- operations.application.RemovedOp

2.4 Reclaim Protocol

The reclaim protocol is also very similar to the reclaim protocol implemented on the first project. The client requests a peer to allow storage of files up to a given number of bytes, or if the requested number of bytes is zero, it means the peer should remove every file and reset the capacity.

The reclaim protocol may lead to deletion of files, for every file removed, a notification is sent to the owner with a REMOVED message, if the replication degree drops bellow the desired, the owner will start a new backup operation for that file, provided the file is still available on the system. The drawback here is that if the file is no longer available it will not be possible to recover and replicate the file once it's deleted by another peer. A possible solution would be to send a message to a peer backing up the deleted file, requesting it to start a new backup.

The source code for the reclaim implementation can be found under these files:

- peer.Peer:507
- operations.application.RemovedOp

Chapter 3

Concurrency Design

As mentioned before, for this project to stay stable and work on a scalable and fault tolerant environment, it was important for us to take as much advantage as possible from the Java's NIO library, as it provides means to work with non-blocking I/O calls. The main uses regarding the NIO cover the following classes:

- `ServerSocketChannel`;
- `SocketChannel`;
- `ReadableByteChannel`;
- `Selector`;
- `ByteBuffer`;
- `FileChannel`;
- `Files`

These classes were helpful since they enhance the system's performance and, most importantly, they work well with the `SSLEngine` class, which we will discuss further on. The Socket Channels bring an huge upgrade in terms of performance since they can make non-blocking read/write calls. On the server side, we keep the selector in a loop, reading from the socket whenever a connection is ready to be accepted or there are application bytes (plain data) ready to be read, leaving the peer enough resources to work on other tasks while there are no bytes to be read. As for the Byte Buffers, they were a must when dealing with the `SSLEngine` interface, for every successful connection four buffers are established, two for encrypted data, and two for application data, both in and out data. These buffers' capacities are defined by the session associated with the engine created for the connection.

As there are multiple tasks which are required to execute concurrently, the managing of those tasks was delegated to `ThreadPoolExecutor`s, the `peer.ssl.SSLPeer` has an executor to handle the delegated tasks when performing an handshake, the `peer.chord.ChordPeer` has an executor to handle the tasks associated with the Chord Network [2] (will be discussed later) and the `peer.Peer` has an executor to handle the tasks associated with client or application requests. As long as there is a place available on the executors to start a task it will run that task right away, otherwise it will be put on a queue, ready to execute once it's its turn.

Regarding I/O operations, we've decided the best option was to use the NIO's capabilities to read and write files to non-volatile memory. Since we need to send replies when the data is fully written or read, we've decided not to use the Asynchronous version of the `FileChannel`.

Chapter 4

Java Secure Socket Extension - JSSE

This entire project is based on the JSSE features, in fact, except RMI communication, all data is exchanged with security and confidentiality. This implementation has only one drawback, since the maximum size allowed for a TLS message is in the vicinity of 16KB, for us to send a file with around 200MB the application must send around 12 500 records with encrypted data, on the tests we've run, there was no real difference in sending the file by a simple socket or using encryption, thus the choice was fairly simple. With this in mind, we know the connection will be substantially slower when the peer's are connected through WiFi, or in different LANs. The trade-off was based on security and confidentiality, leaving the performance a bit lower.

Regarding authentication and cipher suites, the server requests client authentication, and the cipher suites supported are the ones available by the JVM (min. JDK11) and TLSv1.2 which is the protocol version being used by the `SSLContext` on the `SSLEngine`. The authentication is used based on self-signed certificates, for sake of usability and testing, the keystore and truststore passwords and location are passed on methods we've created, since the `SSLContext` uses special classes to handle keystores and truststores. The client authentication is required by the server.

We opted to use the `SSLEngine` which proved to be a real challenge, since this interface is very complex, because it provides means to work the Java's NIO classes, such as the `ByteBuffer` class. An engine is created for every connection and it consumes/produces encrypted data, so it can be ready to be sent or read by the application.

To implement the SSL, or rather, the TLS support for our application, several classes were designed:

- peer.ssl.SSLConnection
  This class encapsulates the engine, socket channel and the four buffers allocated for the connection. Every time a peer starts a connection to another peer, an `SSLConnection` is created, which will then be used to read and write data.

- peer.ssl.SSLCommunication
  This class provides lower level means to process an handshake, read/write data, and read/write files to the socket channel, it also has the means to deal with the Buffer Exceptions thrown while consuming/producing the encrypted data, as well as means to close the connections according to the RFC 2246

- peer.ssl.SSLClient
  This class provides means to connect to another peer.

- peer.ssl.SSLServer
  This class provides means to accept incoming connections, and it also contains the server loop, previously described, using the `Selector`.

- peer.ssl.SSLPeer
  This class provides higher level methods for the I/O operations implemented on the SSLCommunication class, dealing with the exceptions. Since this is the Peer class, it provides means to accept and make new connections, thus having the client and server functionalities. This class is the interface used by the `ChordPeer` (described later on) to send and receive data, by performing all the heavy-lifting related to the exchanging of messages.

We've also consulted a book [1], starting on page 191, since the `SSLEngine` is a challenging interface to implement by ourselves.

In sum, the JSSE features were widely explored on this project, and we can safely guaranty the security and the confidentiality of the data being send between any two peers.

Chapter 5

Scalability

Regarding scalability, we have addressed this issue by implemented a Chord Network [2], as previously referred, and as described on the previous chapter, by taking advantage of the NIO's features.

On a non-scalable system, every peer would know the existence and how to communicate directly with every peer on the network, this is not ideal, as every hash table would have to be updated whenever a peer joins/leaves the network. The solution was to implement a network with a Distributed Hash Table [1].

Following the need to use a DHT system, we opted for the Chord System, widely used and studied on class. For this application, we've used, a $M\_BIT = 8$, allowing up to $2^8 = 256$ peers to exist on the network, thus making the finger table only saving $M\_BIT = 8$ entries, which is enough for the peer to successfully connect to every other peer, according to the paper, and verified on the tests we've run.

Regarding the replicated files stored on each peer, each file has an associated key, which is used to locate said file on the network, and also to maintain the consistency of the network. The file with a key K should be stored by the peer with GUID K or the successor of K if K is not online. If a peer p1 enters the network and its successor has files which should actually be stored by p1, p1 reclaims them, thus maintaining the integrity of the Chord Network. Unlike the first project, we do not save the actual peers which are storing the files, but rather the keys associated with the files, this is not only a scalability feature but also a fault tolerance feature.

The Chord Network implementation can be found on peer.chord.ChordPeer.

[1] A distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table: key-value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. The main advantage of a DHT is that nodes can be added or removed with minimum work around re-distributing keys. Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. Wikipedia

Chapter 6

Fault Tolerance

When a peer crashes (exits unexpectedly), the system is prepared to deal with that event, since there are asynchronous checks to verify if a peer is online or not. This is provided by the Chord features, namely the `checkPredecessor` task. Even though when a peer crashes it cannot let the other peers know it is not online anymore and thus try to replicate its files, we can workaround this by having a replication degree, making a file available on more than one peer, meaning for a file to be completely lost every peer storing said file would have to fail, which is unlikely. On this note, if a call to the successor fails, it proceeds to next available peer on the finger table, making the system fault-proof, unless every peer fails, which we assume it does not occur.

Chapter 7

Final Considerations

7.1 Apache's Log4j 2 Library (Logger)

This system uses a library to log the output on a more sophisticated way, the library is included on the source code, thus having no need to download external sources. The library reference guide can be found here.

7.2 Compilation

To compile the project, execute the following line with JDK11 under `src/`

```
> sh ../scripts/compile.sh
```

it will save the compiled classes, keystores and log4j2.xml (used on logger) to `src/build/`.

7.3 Peer Execution

To start a peer, run the following line with JDK11 under `src/build/`

```
> sh ../../scripts/peer.sh <SAP> <BOOT_IP> <BOOT_PORT> [-b]
```

- SAP - Service Access Point (used for RMI)
- BOOT_IP/BOOT_PORT - Used to Join the Chord Network
- Flag -b - used to signal if the peer is Boot
  This flag will mean the peer will take the boot peer IP/port passed as argument as its own, making it the boot peer, this flag is used for convenience, as we would have to start a peer and check it's IP/Port, and then start the other peers with it as reference for the Chord Network.

7.4 Client/TestApp Execution

To start a client, and send a request to a peer run execute the following line with JDK11 under `src/build/`

```
> sh ../../scripts/test.sh <operation> [<operand_1> [<operand_2]]
```

- operation - BACKUP | RESTORE | DELETE | RECLAIM | STATE | CHORD | LOOKUP
  as described on table 2.1, these operations have arguments associated.

7.5 Multiple Peers Execution

To start eight peers, with a 1 second delay between, run the following line with JDK11 under `src/build/`:

```
> sh ../../scripts/peers.sh
```

7.6 File Structure

The file's structure will be similar to the following tree example

```
build/
├── application_classes
├── ...
├── peer1/
│   ├── data.ser
│   ├── file1
│   ├── file2
│   └── file3
├── peer2/
│   ├── data.ser
│   └── file2
├── peer3/
│   └── data.ser
├── peer4/
│   ├── data.ser
│   └── file3
├── peer5/
│   ├── data.ser
│   ├── file1
│   └── file3
└── ...
```

7.7 Documentation

We've documented the whole code, and the Javadocs can be consulted here.

Bibliography

[1] Esmond Pitt. *Fundamental networking in Java.* Springer Science & Business Media, 2005.

[2] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.