SDIS 2019/2020 - Project 2
Distributed Backup Service for the Internet

João Paulo Gomes Torres Abelha - 201706412
João Rafael Gomes Varela - 201706072
Vítor Hugo Pereira Barbosa - 201703591

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal,
`FEUP-SDIS, Class 1, Group t1g24`

Abstract. This article presents the details about our implementation of a Distributed Backup System, namely it's architecture and functionalities. Alongside, we also explain the thinking process behind some technical details, that usually revolves around building a very robust and scalable system. This service was build on top of the Chord protocol, implemented in Java, making use of some of it's powerful packages such as SSLEngine, Java NIO and so on.

Keywords: Online Distributed System, Backup Service, Chord, SSLEngine, Java NIO, RMI, P2P

1 Introduction

To allow for a better comprehension of the project developed and its implementation, in the following sections we explain how we tackled several problems that come along with a distributed service like this. The article can be split in the following sections:

- **Overview** - description main features of the project, including the operations supported by the backup service.
- **Protocols** - description of the implemented and underlying protocols that support the system.
- **Concurrency Design** - support the handling of concurrent service requests.
- **JSSE** - when and why we use JSSE.
- **Scalability** - our approach, on a design and implementation level, to guarantee a scalable system.
- **Fault Tolerance** - description on the system's fault tolerance features.
- **Conclusions and Future Work** - conclusions, results discussion and aspects to be improved.
- **References** - books, articles, web pages and other means used to develop the project.

2 Overview

The project developed consists on a peer-to-peer distributed system for the internet that would allow for the usage of free disk space of the computers on the system to backup files. Besides backing up files, the service allows a user to restore or delete those same files. It is worth to note that each peer participating on the service still retains total control over their own storage.

2.1 Server

This service is built on top of a **decentralized server** whose participants follow the **Chord** protocol. To participate on this server, one must provide an access point (so that it can be later used by a client) and port to be able to communicate with other peers. Besides that, it must also be provided the port and address of any Chord's participant (unless there is none yet) so that this peer can join the chord too. Hence, a peer can be invoked as follows:

```
java Peer <peer_ap> <port_1> [<address> <port_2>]
```

Where:

- *peer_ap* - peer's access point.
- *port_1* - port to be used when communicating.
- *address* - address of an already existing peer on the chord.
- *port_2* - port of an already existing peer on the chord.

To guarantee a high level of **scalability** we made use of Java **thread pools** and ensured an asynchronous communication using Java **NIO**. Building the system on top of **Chord** was also very helpful on this regard, since it allows for peers to join and leave the service at any time with minimal disruption on the network.

We avoided single points of failure not only by using the **Chord's** inherent fault tolerance features, but also by allowing the same file to be **replicated** across different peers. This way we can ensure a very high level of **fault-tolerance**.

In order to maintain a **secure** communication between the peers we opted to use **JSSE SSLEngine**. Even though it might be harder to implement the service around this interface (e.g. when comparing it with SSLSockets) it is much more flexible when working with other API's, allowing us to achieve an higher concurrency.

2.2 Client

To make use of the system, we built a client that can invoke the following service protocols:

- **Backup** - backs up a file with a given replication degree.

– **Restore** - restores a file that was stored on the system.
– **Delete** - deletes a file from the system.
– **Reclaim** - reclaim allocated disk space.
– **State** - retrieves information about a system's participant.

Where:

– *peer_ap* - peer's access point.
– *sub_protocol* - one of the mentioned service protocols (BACKUP, RESTORE, DELETE or STATE).
– *opnd_1* - path of the file to be used on the protocol.
– *opnd_2* - replication degree of the file to be backed up (only used on the Backup protocol).

3 Protocols

3.1 Remote Interface

In order to communicate between the client and the server, we made use of the Java **RMI** API. This way, the client can invoke functions on the peer, that will correspond to the execution of each sub-protocol, using a "remote interface" (found on the *peer* package at RMI.java) that looks like the following:

```
public interface RMI extends Remote {

    public void backup(String file, int replicationDegree) throws
        RemoteException;

    public void restore(String file) throws RemoteException;

    public void delete(String file) throws RemoteException;

    public void reclaim(int max_size_kbs) throws RemoteException;

    public String state() throws RemoteException;
}
```

3.2 Backup protocol

The backup protocol is ensured by the *Backup* class and by a set of messages made for it. This protocol starts by checking if the corresponding file is not backed up (*PeerFiles : addBackupRequest()*). Then if the file is not backed up it starts creating a number of replicas equal to the number of the desired replication degree. To generate each replica, we use the information of the class **fileInfo and the replica number**, starting at zero and going up depending on the replication degree. The information of the file used to create this identifier is

the name, the size and the creation date (*SHA256Hasher : hash()*). We needed to create **different identifiers for each replica** since we **wanted to store them in different peers** so that we can achieve our replication degree and recover the file if lost in one of the nodes. This also contributes for the **system's fault tolerance**.

For each different replica, the system executes the runnable *SendBackupRequest*. To find the node that is going to store each replica the initiator peer calls the method *findSuccessor* for each replica's key (*SendBackUpRequest : SendBackUpRequest()*) . After that it sends the message BackupRequestMessage (*SendBackUpRequest : run()*) in order to fullfill the protocol, so then it simply waits for an answer which can either be : a *BackupConfirmMsg*, *BackupAck* or *BackupNack*.

The first message serves for when it **already has the replica** in that peer stored. If the message *BackupAck* is received then it has **green light to store the replica** while the *BackupNack* **informs of the lack of space** in the peers. If we received the permission to store the file, it will be sent a *BackupContent* message. This message contains the size of the file, it's id and the replica's id. This parameters **are important** since we could have various simultaneous requests, which could lead to the peer who responded with a *BackupAck* message **no longer having enough space** when he receives the *BackupContent* message. After sending this message the initiator peer sends the file (*FileSender : transfer()*) to the peer who has space to store the replica. The receiver checks if it still has enough space and adds the replica to the **backup_files** *ConcurrentHashMap* (*PeerInformation : addReplica()*), and then starts reading the file (*FileReceiver : readFile()*). In case everything goes as planned the peer answers with a *BackupConfirmMsg* otherwise answers with a *BackupNack*.

Additionaly, we have the message *BackupSuccessorMsg* which is sent by the **successor of the replica** in case he **cannot store it**. This messages propagates if the node does **not have the space needed** to store the replica or have **already saved a replica of the requested file**. If there is a node that has enough space, it sends a message to confirm the possibility of storing it and the peer saves the location (*PeerInformation : addRemoteReplica()*) so it can do the remaining protocols later.

3.3 Restore protocol

The restore protocol is managed by a set of messages and the class *Restore*. This protocol starts by checking if the corresponding file was backed up by this peer (PeerFiles : getFile()), then if the peer backed up the required file the protocols tries to retrieve a replica of the file. **There are two distinct situations**. In the first one, the node **has a replica** (*PeerInformation : hasFile()*) of the required file and therefore it reads the content of the replica and restores the file in the corresponding directory.

In the second and most frequent one, the node **doesn't have a replica of the file** and needs to find where the file was stored. First the replica's id of the file we want to restore are generated and a *SendRestoreRequest* is initialized for

the first replica. It starts by finding the successor of the replica to know where to send the message. After finding the successor a *RestoreRequest* message is sent to the responsible node. When the message is sent the node will be waiting for a reply message. If the message is of type *RestoreContent* the node will **start reading the file** (*FileReceiver : readFile()*), if no message is received **the node will start a *SendRestoreRequest* for the next replica**.

When a node receives a *RestoreRequest* messages it will check if it **has the replica** (*PeerInformation : hasReplica()*) or **knows where the required replica is stored** (*PeerInformation : hasRemoteReplica()*). If it has the replica a *RestoreContent* is sent and the node starts sending the file (*FileSender : transfer()*), if it knows where the replica is the **request message is forwarded**.

3.4 Delete protocol

The delete protocol uses the class *Delete* and a set of messages to work properly. This protocol has the objective of deleting all the replicas present in the chord. To begin with, it checks if the corresponding file was backed up by this peer (PeerFiles : getFile()), and if so it generates all the replicas of the file we want to delete and start for each one a *SendDelete*.

The *SendDelete* starts by using the method **findSuccessor** so that we find in which nodes are the replicas stored. Following this, a message is sent, namely a *DeleteRequestMsg* to the responsible node. Once the target nodes receive the messages, it checks if it **has the replica** or **knows where the replica is stored**. In case it has the replica, it deletes (FileDeleter : deleteFile()) and removes it from the **backup_files** *ConcurrentHashMap* (*PeerInformation : deleteReplica()*) and then it sends a *DeletedAck* messages to confirm its success. Otherwise, it **forwards the message** to the node with the replica.

3.5 Reclaim protocol

This protocol is ensured by the Reclaim class and a set of messages, namely the ones used in the backup protocol. Since a peer must have total control of the space that it uses, when we need to free some space we have to try to find a different node to save those replicas. This poses a problem as the Chord algorithm computes the key location given only the key itself, thus a **change in location would make it impossible to find the location of a replica**. To overcome this problem, **we kept the node as responsible for the replica but stored it in another node**, registering the new location.

To reach the maximum size imposed the peer will go through its stored replicas until the **space occupied is less than or equal to the max space** *(PeerInformation : isOversized())*. **For each replica a new backup protocol** (*SendBackupRequest* and *SendBackupContent*) is started and if everything goes well the location of the replica is updated.

3.6 State protocol

This protocol is ensured by the *PeerInformation* class and unlike the others it does **not require any messages**. This protocol is used to print to the console the **node state**, namely its stored replicas, the ones it is responsible for, its port and adress and lastly the size used and the maximum size it has available.

3.7 Redistribute protocol

To understand why we implemented this protocol, let's think about the situation where we have stored in a node a certain replica and meanwhile a new node enters the chord.Within a node that possesses files stored, it **can happen that those files actually belong to another node that is closest to the previous said node and that precedes it**. When one runs the protocol, the peer initiator sends a message *RedistributeReplica* to its predecessor so as to **inform it about the location of some files that it might be responsible for**. This way, when trying to find the successor even if those files are not physically there, there is a way to reach the replicas .

3.8 Messages

The system's peers communicate between themselves, using TCP, through the usage of different types of Messages. This messages are mostly used to ensure the sub-protocols functionalities and to follow the Chord protocol. All of this message have two things in common: their first field indicates the type of message; their fields are separated by the sequence '0xD' '0xA' and terminate with that same sequence, duplicated.

Each type of message transports the necessary content to be given to the receiving peer. For instance, if the receiving peer needs to respond back, the message will contain the address and the port where the sender is listening. Other times, the message transports keys of files and even their contents.

4 Concurrency Desgin

So that multiple request can be handled concurrently, each Peer has its own **ExecutorService** that automatically provides a **thread pool** and an API for assigning tasks to it. The ExecutorService interface allows this tasks to be executed in asynchronous mode and is initialized as follows (*Peer : initPeer()*):

```
this.threadPool = Executors.newScheduledThreadPool(POOL_SIZE);
```

where the constant POOL_SIZE corresponds to the maximum number of threads on the pool.

Each thread on this pool will either be responsible for running a sub-protocol, processing received messages or to execute the runnables that ensure the correct state for the chord (*CheckPredecessor*, *FixFingers* and *Stabilize*).

To allow for a **non-blocking and asynchronous communication** between peers, we made use of the Java **NIO** API, namely its *AsynchronousServerSocketChannel* (an asynchronous channel for stream-oriented listening sockets) and *AsynchronousChannelGroup* (containing the threads responsible for processing the received messages for each peer) classes. When creating the socket channel, we immediately associate it with the channel group and bind it to the peer's local address, as follows (*Peer : initPeer()*):

```
AsynchronousChannelGroup group;
AsynchronousServerSocketChannel server;
...
ExecutorService threadPool = Executors.newFixedThreadPool(POOL_SIZE)
group = AsynchronousChannelGroup.withThreadPool(threadPool);
server = SSLUtils.getServerSocketChannel(group);
InetSocketAddress socket = new InetSocketAddress(localAddress, port);
server.bind(socket);
```

We also resort to the Java NIO *CompletionHandler* to consume the result of asynchronous I/O operations on the socket channel such as: accepting a connection, connecting, writing and reading.

This handler allows us to define two different functions (in case of success or failure) that will be executed asynchronously after the operation finished executing. For instance, this CompletionHandler will be useful when waiting for another peer to connect to it's channel and we do that in the following way (*Peer : listen()*):

```
public void listen() {
    this.server.accept(null, new
         CompletionHandler<AsynchronousSocketChannel, Void>() {
        public void completed(AsynchronousSocketChannel channel, Void
             attachment) {
            server.accept(null, this);
            threadPool.execute(() -> {
                SSLUtils.readMessage(channel, new
                     ReadMessageCompletionHandler(Peer.this, channel));
            });
        }

        public void failed(Throwable exc, Void attachment) {
            // throw new RuntimeException("unable to accept new
                 connection", exc);
        }
    });
}
```

This makes handling the response a much more easier task. In this case, we simply put the code that we want to execute, after a connection was made, inside the *completed* function and the same thing for when the connection fails (we

throw and exception inside the *failed* function). The same structure is followed for the rest of the operations, just needing to substitue the *accept* call by the required function (*connect*, *write* or *read*).

5 JSSE

By using **JSSE** (Java Secure Socket Extension), it is possible to ensure that data traveling over the network is only accessible to the intended recipient.

For this purpose, the *SSLEngine* class was used in this project, this class enables applications to deal in secure protocols such as **SSL** and **TLS** but is transport-independent. It does not deal with sockets or channels or streams: it deals only with *ByteBuffers*. It is therefore possible to use the *SSLEngine* in conjunction with **non-blocking I/O**, a very important advantage for the server side of secure communications protocols.

In this project *SSLEengine* was used in the development of *SLLAsynchronous-SocketChannel* and *SSLAsynchronousServerSocketChannel*. This classes can be used as *AsynchronousSocketChannel* and *AsynchronousServerSocketChannel*, mentioned above, since they implement the same methods.

The *SSLEngine* class was used in this classes, during connection establishment (*SSLAsynchronousSocketChannel : connect()*) to make the handshanke (*SSLEngine : beginhandshake()*), handling the respective *SSLEngineResult.HandshakeStatus* (*SSLAsynchronousSocketChannel : doHandshake()*) to allow the exchange of information from both sides that will allow secure communication. *SSLEngine* was also used to encrypt and decrypt the information coming from the network, after reading (*SSLAsynchronousSocketChannel : read()*) and before writing (*SSLAsynchronousSocketChannel : write()*), using the **wrap** and **unwrap** methods.

After the creation of SSLEngine (*SSLContext : createSSLEngine()*) it was defined which cipher suites to use in order to choose which algorithms to use during the connection. Some research was made to reduce the set of available to a subset with the strongest. Firstly, the key exchange algorithm used is **ECDHE**. Although ECDHE does not provide authentication, this was enabled through the following method: SSLEngine.setNeedClientAuth(true) (*SSLAsynchronousServerSocketChannel*). Next, two certificate authorities are allowed: **ECDSA** and **RSA**, for compatibility. The preferred message encryption mechanism preferred is AES 256-bit with GCM, which provides encryption, as well as the CBC does, but also integrity checking, which the latter does not. Finally, the preferred hashing algorithm is SHA-384, which provides the strongest hashes. Therefore, the cipher suites are as follows:

1. TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
2. TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

6 Scalability

At an **implementation level** there are two main reasons that ensure this system to be a very scalable one. By using **thread pools** we can achieve an higher concurrency level leading to a system that can hold more operations at the same time, while keeping a relative fast execution time of the sub-protocols.

The same thinking proccess can be applied to the usage of the Java **NIO** API. Its asynchronous communication features permit other processing operations to continue before the transmission has finished, resulting in much less overhead in the sub-protocols execution.

Regarding the **system's design**, as stated before, we opted to build the system on top of the **Chord** protocol. We opted for this peer-to-peer protocol due to its simplicity, proved correctness and performance. Chord plays an important role in the scalability of our system since it aims to not overload any peer on the network but also because it uses an hashing scheme designed to let nodes enter and leave the network with minimal disruption. We implemented a chord's Node interface that the Peer class implements, having to define basic chord functionalities such as:

- **join chord** (*Peer : joinChord()*) - let's a node join the chord
- **stabilize** (*Stabilize.java*) - node n asks its successor for its predecessor p and decides whether p should be n's successor instead
- **notify** (*SendNotifcation.java*) - notifies n's successor of its existence, so it can change its predecessor to n
- **fix fingers** (*FixFingers.java*)- updates finger tables

These last three functionalities are known as the *stabilization* of the chord and ensure that all successors pointers are up to date, guaranteeing that the lookups retrieve the correct node, hence they must be executed periodically. In our case, all three of them are represented by java runnables and are run with intervals of 300 milliseconds within the peer's threadpool:

```
public void stabilize() {
    this.threadPool.scheduleAtFixedRate(new CheckPredecessor(this), 0,
        300, TimeUnit.MILLISECONDS);
    this.threadPool.scheduleAtFixedRate(new Stabilize(this), 0, 300,
        TimeUnit.MILLISECONDS);
    this.threadPool.scheduleAtFixedRate(new FixFinger(this, 10), 0, 300,
        TimeUnit.MILLISECONDS);
}
```

The joinChord function allows a peer to join the chord, as long as we give him information about one existing peer in the chord.

```
    public boolean joinChord(PeerSignature successor) {
        PeerSignature result =
            this.requestSuccessor(successor,this.signature.getId());
```

```
        if (successor == null) {
            System.out.println("Cannot find successor");
            return false;
        }

        this.setSuccessor(result);
        // start running threads
        this.stabilize();

        return true;
    }
```

By requesting its successor to a known peer, the peer trying to join the chord will be able to set a successor and establish a connection with it. This peer will then notify its successor (*Stabilize.java*) enabling him to update its predecessor (if it is the case). The only thing missing is that the joining peer still needs to know its predecessor, which will happen when that predecessor executes its stabilizer functions.

This stabilizer functions are key for the service to be scalable, since they are responsible for keeping the peer's connected to each other even when some of them leave the system or others enter.

7 Fault Tolerance

In order to avoid single points of failure, we allow a file to be stored multiple times (as long as there are enough peers on the chord) in different peers. We enable that, as stated before, on the backup protocol by generating a different key for each file replica to be backed up.

8 Conclusion and Future Work

The development of the project allowed the group to put into practice the knowledge acquired during the Distributed Systems classes. We were able to reinforce our understanding of a distributed system, and it's importance on creating efficient solutions for a diverse set of problems.

We were able to build a Peer-to-Peer distributed application, where the peers could communicate between themselves to provide a backup service. Understanding the importance of a scalable design, we've developed the application on top of the Chord protocol, using asynchronous I/O and threadpools. These are all important features that ensure the system's high concurrency and high scalability level.

Being a system than communicates through the internet, it is very important to guarantee that messages could not be intercepted. We tackled this problem by using the java SSLEngine API.

Since TCP communications can not be one hundred percent reliable, it was important to implement fault-tolerance features. We faced some difficulties when

it came to this topic, and could improve it on the future. Nevertheless, we allow a file to be backed up multiple times in different peers, which itself contributes to a fault-tolerant system.

Summarizing, we have come to the conclusion that building a distributed system for the internet that is secure, fast, scalable and reliable can represent a demanding challenge, having required the group to think profoundly about the system's architecture before and while building it. Nevertheless, we still achieved a very satisfactory solution that reflects the group's solid efforts.

References

- Project Specification:
  https://paginas.fe.up.pt/ pfs/aulas/sd2020/projs/proj2/proj2.html
- Name Resolution in Flat Name Spaces by professor Pedro Souto
  https://paginas.fe.up.pt/ pfs/aulas/sd2019/at/15dhts.pdf
- Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications
  https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407?download=true
- Chord Implementation
  http://web.mit.edu/6.033/2001/wwwdocs/handouts/dp2-chord.html
- Understanding and Implementing Chord
  https://www.cs.princeton.edu/courses/archive/fall18/cos561/docs/ChordClass.pdf
- Fundamental Networking in Java, by Esmond Pitt 2006