U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

SDIS 2019/2020 - 2nd Semester

Project 2 - Final Report

Distributed Backup Service for the Internet

May 30, 2020

T2G23

Ana Filipa Campos Senra (up201704077)

Cláudia Inês da Costa Martins (up201704136)

Joana Catarina Teixeira Ferreira (up201705722)

João Nuno Carvalho de Matos (up201705471)

Index

Overview

As the second project assignment for FEUP's Distributed Systems course, we were tasked with building a reliable, secure and scalable peer-to-peer distributed backup service for the Internet. It features the backing up with replication, restoring and deletion of files, as well as an administration API.

As part of the semantics of our system, the clients have access to the following API:

- Functional protocols:
    - Backup - backs up file with desired degree of replication;
    - Restore - restores a file that was previously stored;
    - Delete - deletes the stored copies of the specified file.

- Administrative protocols:
    - Reclaim - sets a ceiling on a peer's disk usage, offloading stored files to other peers if needed;
    - State - retrieves useful information about a peer's internal state;
    - Shutdown - remotely triggers the safe shutdown of a peer, offloading its stored files to other peers.

To satisfy the reliability, security and scalability needs of our system, the following features were also implemented:

- Scalability was assured by utilizing a distributed hash table implemented over the *Chord* protocol, and by parallelizing IO and processing through the use of thread pools, selectors, and non-blocking sockets.

- Security was assured by encrypting traffic between peers, making use of Java SE's JSSE package through the *SSLEngine* API.

- Reliability was assured on an application level by enabling users to specify a desired replication degree, and on a service level by implementing *Chord*'s fault tolerance protocols.

Protocols

As a decentralized, distributed backup service, our peers exposed two APIs/protocols. The client interface is served over Java's RMI protocol, and the peer-to-peer protocol was implemented with TCP and TLSv1.2.

Client Protocol

The protocol between our client, TestApp, and the peer was implemented using Java's Remote Method Interface API. This API provides a simple way of implementing RPC communication, taking care of service discovery, serialization and deserialization and the implementation of both client and server stubs.

What was left to do was defining a standard Java interface class that the client depended on and the peers implemented, which ended up looking like the following listing:

```
public interface InterfacePeer extends Remote {
    void backup(String file_path, int replication_degree) throws
Exception;
    void deletion(String file_path) throws Exception;
    void restore(String file_path) throws Exception;
    void reclaim(String file_path) throws RemoteException;
    void shutdown() throws RemoteException;
    String state() throws IOException;
}
```

Peer-to-Peer Protocol

To use a protocol between each peer in the cluster, we chose to use a simple protocol implemented over TCP, encrypting the messages with TLSv1.2. Each message was made up of a text header encoded as ASCII, followed by two line breaks (each represented by the ASCII characters CR and LF) and, optionally, a message body. The header has several mandatory fields, notably the protocol version as a decimal, the message type as a unique identifier string, and the request id, which represents different things in different contexts.

In the following subsections the possible message formats are presented.

Chord Protocol Messages

Find Successor

When a node needs to find a successor of an id, he may ask another node to find it. This can be made by calling a message of type:

<version> FIND_SUCCESSOR <request_id>

And it receives a message of type:

<version> SUCCESSOR <request_id> <succesor_address> <successor_port>

Otherwise, it will determine that no successor of request_id can be determined.

Closest Preceding Node

When updating the list of successors, the node needs to find out the predecessor of a successor.

<version> FIND_PREDECESSOR <node_id>

And it receives a message of the following type, if the node has a predecessor determined:

<version> PREDECESSOR <request_id> <predecessor_address> <predecessor_port>

Or it receives a message of the following type if is has not a predecessor determined:

<version> NOTFOUND

Notify

When a node joins the network, it needs to notify its successor. It can do it by sending a message of type:

<version> NOTIFY <succesor_id> <successor_address> <successor_port>

And receives a message of type OK. If not, something went wrong and we try again later.

Health Check

Periodically, each peer requests a health check from its predecessor, calling a message of type:

<version> CHECK_UP <predecessor_id> <predecessor_address> <predecessor_port>

And receives a message of type I_AM_OK. If not, it will forget about it and find a new predecessor afterwards.

Backup Service Protocol Messages

To support our service's functionality, we had to implement the following protocols.

Backup

After performing the *Chord* lookup to identify which peers should store the file and its replicas, the first peer sends a message with the following format to said peers:

<version> BACKUP <fileId> <address> <port> <repDegree> <chunkNo> CRLF CRLF <body>

If they store the chunk successfully, they send a BACKUP_COMPLETE message:

<version> BACKUP_COMPLETE <nodeId>

Otherwise they discard the file and send a BACKUP_FAILED message back to the initiating peer:

<version> BACKUP_FAILED <nodeId>

Restore

The initiating peer locates the peers responsible for keeping the backup and sends a RESTORE message of the type:

<version> RESTORE <fileId> <address> <port>

If the receiving peer has a copy of the file, it will send a message:

<version> SENDING_FILE <fileId> <address> <port>

And initiate a file sending protocol, which is composed of messages of the type:

<version> RESTORED <fileId> <address> <port> <chunkNo> CRLF CRLF <chunkData>

If the initiator sees all is correct, it issues a RESTORE_COMPLETE response.

Delete

When a peer receives a DELETE message from the Test App, it will check if it currently has the file to delete stored. If it does, it will initiate the DELETE protocol. If not, it cannot start the delete protocol as it does not have the replication degree stored. Therefore, it will pass the responsibility to another peer by sending a DELETE_RESPONSABILITY message:

<version> DELETE_RESPONSABILITY <fileId> <address> <port>

If a peer receives a DELETE_RESPONSABILITY message, it will verify if the desired file is stored. Given this verification and given that only peers which store a file have access to its replication degree, the peer will take one of the following actions:

1. The current peer is storing the file:
    a. Responds with a DELETED_RESPONSABILITY_ACCEPTED message:

        3 DELETED_RESPONSABILITY_ACCEPTED <nodeId>

b. Initiates a new DELETE action but also sending the delete message to itself first.

2. The current peer is not storing the file:
    a. The peer sends a new DELETED_RESPONSABILITY_ACCEPTED to the next peer.

Once a peer receives a DELETE message, it will verify if the desired file is stored. Given this verification, it will take one of the following actions:

3. Success:
    a. The file is deleted and responds with a DELETED confirmation message:

        3 DELETED <fileId> <address> <port>

4. If the deletion fails or the file is not the peer:
    a. The peer responds with a NOT_DELETED message:

        3 NOT_DELETED <fileId> <address> <port>

Reclaim

Once a peer receives a RECLAIM message, it will start deleting some of its file until the occupied space is smaller than the new desired overall space.

For each deleted file, REMOVED messages will be sent to the next peer in the following format:

- Header: version, REMOVED, file number, chunk number, replication degree
- Body: chunk's data

If the receiving peers store the chunk successfully, they send a BACKUP_COMPLETE message:

<version> BACKUP_COMPLETE <fileId>

Otherwise they discard the file and respond with a BACKUP_FAILED message:

<version> BACKUP_FAILED <fileId>

Concurrency design

To allow the protocols to run concurrently we adopted several practices.

This project reflects an environment in which several threads run simultaneously. There for, instead of using *HashMaps* as the data structure to save the information of the communication between peers, we used *ConcurrentHashMap*. This data structure is an efficient solution because it is thread-safe and has a great performance in multithreaded environments. In the code below it's demonstrated the usage of this data structure in the class *Storage*, where we save information about the files a peer has sent to backup in the *backedUpfiles* field, for example.

```
//files he has sent a backup
private final ConcurrentHashMap<BigInteger, FileInfo> backedUpFiles =
new ConcurrentHashMap<>();
 //rep of files stored
private final ConcurrentHashMap<BigInteger, Integer>
storedFilesReplicationDegree = new ConcurrentHashMap<>();
 //buffer to store files while they are being received
private final ConcurrentHashMap<BigInteger, ArrayList<byte[]>>
bufferFiles = new ConcurrentHashMap<>();
//buffer for file path
private final ConcurrentHashMap<BigInteger, String> bufferFilePath = new
ConcurrentHashMap<>();
```

We also use threads several times for allowing the peer to perform several tasks simultaneously. For example, in the protocols (backup, delete, restore), in the *Chord Protocol* and in the *SSLEngine*. In the picture below is an example of the thread's usage in the backup protocol (notice the use of a *ScheduledThreadpoolExecutor* on *exec.execute(...)* to run the task on a worker thread):

```
public void backup(String filepath, int replicationDegree) throws
Exception {
    System.out.println("\nBACKUP SERVICE");
    System.out.println(" > File path: " + filepath);
    System.out.println(" > Replication Degree: " + replicationDegree);
    System.out.println();

    File file = new File(filepath);
    BigInteger fileId = Backup.generateFileId(file.getName(),
file.lastModified(), file.getParent());
    byte[] fileData = Files.readAllBytes(file.toPath());

    if (!PeerClient.getStorage().getStoredFiles().contains(fileId))
        exec.execute(new Backup(fileId, fileData, filepath,
replicationDegree));
```

```
    else
        System.out.println("File already backed up!");
}
```

We also used *Java NIO* in the *SSLEngine* implementation. For example, in the *SSLEngine* classes and *ReceivedChordMessagesHandler* class we use non-blocking *socketChannels* to communicate between the peers and we also use a *Selector* so that we need less threads to handle the channels. In the code below is an example of this usage extracted from the *SSLEngineServer* constructor:

```
selector = SelectorProvider.provider().openSelector();
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.configureBlocking(false);
serverSocketChannel.socket().bind(new InetSocketAddress(address, port));
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

JSSE

Data that travels across a network can be easily accessed or even modified by someone who is not the intended recipient. It is important to protect private information and make sure data hasn't been modified, either intentionally or unintentionally, during transport.

The Java Secure Socket Extension (*JSSE*) provides a framework and an implementation for a Java version of the SSL and TLS protocols, protocols designed to help protect the privacy and integrity of data while it is being transferred across a network.

We were given the choice between using the *SSLSockets* or *SSLEngine* from *JSSE Java Packet*. We opted to use the more advanced interface: *SSLEngine* with the *TLSv1.2* protocol. This interface allowed us to use *Java NIO*'s non-blocking *SocketChannels*, achieving higher concurrency as opposed to *SSLSockets* (this was further described in the Concurrency Design section). We felt that *TLSv1.2* protocol was a good choice since it is the latest standard version of the *SSL* and *TLS Protocols*.

For implementing the communication protocol with *SSLEngine*, we created 3 classes: *SSLEngineHandler*, *SSLEngineServer* e *SSLEngineClient*.

SSLEngineHandle

The *SSLEngineHandler* class serves as a base for the Server and Client classes. This class implements methods that handle the core functionalities needed to communicate between peers:

- ***handshake(****SocketChannel socketChannel, SSLEngine engine)**:* handles the initial *handshake* between two peers that allow the authentication of both of them.
- ***read(****SocketChannel socketChannel, SSLEngine engine)**:* allows a peer to read an incoming message.
- ***write(****SocketChannel socketChannel, SSLEngine engine, byte[] message****)*** or ***write(****SocketChannel socketChannel, SSLEngine engine, String message)**:* allow a peer to write a message to another peer.
- ***shutdown(****SocketChannel socketChannel, SSLEngine engine****)*** - when we are the peer that initiates the shutdown - or ***closeConnection(****SocketChannel socketChannel, SSLEngine engine****)*** - when we respond to a shutdown request: handle the shutdown protocol.

SSLEngineServer

The *SSLEngineServer* class is used by all peers and listens to incoming messages or requests to establish a connection from clients. The *SSLEngineServer* class implements the *Runnable* Interface. It should be executed as a Thread in order to listen to new messages.

In order to receive the requests, we choose to use the *Java NIO*'s Selector class. *Java NIO*'s *Selector* class (and more generally readiness selection and multiplexing) makes it possible for a single thread to efficiently manage many I/O channels simultaneously. Opting to use the *Selector* class simplified the listening to new requests without any face value trade-offs.

When we want to close the server, we should do it by calling the *stop()* method which will execute an orderly exit.

When creating an *SSLEngineServer* class, a Class that implements the Interface *MessagesHandler* must be provided. This class will determine what behaviour should be performed after receiving a message. For example, we can simply display the message or respond to it. This design allows the *SSLEngine* Layer to run independently from other layers.

```
public interface MessagesHandler {
    public void run(SocketChannel socketChannel, SSLEngine engine,
byte[] message);
}
```

SSLEngineClient

The *SSLEngineClient* class is used by peers when they want to send a request (asking for information) to another peer.

The client connects to a server by a given address and port. After initialization, the *connect()* method should be called. This will establish a connection with the server. After the connection is established, it is possible to write to the server through *write(byte[] message)* or *write(String message)* methods and read a message from the server through the *read().*

When the Client wants to shutdown the connection with the server, it can do it by calling the *shutdown()* method which will execute an orderly shutdown.

We use *JSSE SSLEngine* in all protocols within our Application. Implementing several communication protocols would greatly increase the complexity of our Application and using the *SSLEngine* in all the messages provides a secure environment for messages by

encrypting the information. Therefore, as well as it protects messages for prying eyes, it makes sure they haven't been modified intentionally or unintentionally, during transport. We opted to use the *Cipher Suits* enabled by default when creating an *SSLEngine* since we do not master this subject. Client authentication is not very common in the real world, so we felt it wasn't necessary to require it.

Scalability

Since we wanted our Application to be a Distributed Scalable System, we opted to implement the *Chord Protocol*. *Chord* uses a variant of consistent hashing to assign keys to *Chord* nodes. The *Chord Protocol* supports just one operation: given a key, it maps the key onto a node.[1] In the case of our application, that node is also responsible for storing the data associated with the key. The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible (*ibid*). More specifically, in the steady state, in a N-node system, each node maintains information only about *O(log N)* other nodes, and resolves all lookups via *O(log N)* messages to other nodes (*ibid*).

Scalability with *Chord Protocol*

Keys

The *Chord Protocol* requires that Nodes and the Info they are saving have unique Keys. We have used an SHA-256 to generate an unique Id and used the most M significant bits. This diminishes the collisions that can possibly happen when mapping files or nodes to keys.

Node

A very small amount of routing information suffices to implement consistent hashing in a distributed environment. Each node need only be aware of its successor node on the circle (*ibid*). Since we implemented the Fault-tolerance features of *Chord,* we are not only aware of one the successor but of *r* nearest successors (approached in more detail in the Fault-tolerance section). We represent each Node through a class with the same name.

```
SimpleNode predecessor;
private final Finger[] fingerTable;

//the node's first r successors
private final SimpleNode[] listOfSuccessors;
```

[1] "Chord - MIT CSAIL Parallel and Distributed Operating ...." https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf. Data de acesso: 30 maio de 2020.

Finger Table

To avoid linear search, the *Chord Protocol* mandates that each Node maintains a routing table with (at most) M entries, M being the number of bits in the key/node identifiers. We represent the finger table through an array of M size. Each entry of the table is represented by the Finger Class.

```
public class Finger {
    BigInteger start;
    SimpleNode node;

    public Finger(SimpleNode node, BigInteger start) {
        this.node = node;
        this.start = start;
    }
}
```

Find Successor

Finding the Successor of an Id is essential for the *Chord Protocol.* We have implemented a simple algorithm to resolve this query. First, we check if the id is equal to the id of the current node. If it is, then the current node is the successor. If the id is between the current node id and its predecessor, then the current node is also the successor. If none of the other cases applied, we try to find the closest node present in the finger table and ask for its successor. When asking for the successor of an Id, to another node, this process is repeated.

Stabilization

When we perform a Lookup operation, all the successors must be currently up to date.

When a new node joins or leaves the network, the network is unstabilized and look up operations can be performed incorrectly in some cases. To ensure this does not happen, we need to stabilize the network by performing three important operations periodically in the background that keep the finger table, successors and predecessor up to date:

1. To keep the list of *r* successors up to date, we use the stabilize method (explained in detail in the Fault-tolerance section).

2. To keep the predecessor up to date, when the first successor of a node changes, we notify the successor. When the successor node receives a ‘NOTIFY’ message, it updates its predecessor to the correct one which was the one who sent the message.

```
public boolean notify(SimpleNode n_) {
    if(n_ == null)
        return false;
    if(n_.id.equals(this.id))
        return false;
```

```
    if(this.predecessor == null)
        this.predecessor = n_;
    else if( isBetween(n_.id, this.predecessor.id, this.id))
        this.predecessor = n_;

    return true;
}
```

3. To keep the Finger Table up to date, we ask periodically the node to find the successor of a Finger Table entry.

```
public void fix_fingers(){
    //find last node p whose i_th finger might be n
    SimpleNode p = find_successor(this.id.add(new
BigInteger("2").pow(next)));

    this.fingerTable[next] = (p == null)? null : new Finger(p,
calculate_start(next));
    next++;

    if(next >= m)
        next = 0;
}
```

4. We also need to make sure our predecessor is still alive. If it is not, then we set the predecessor to null. The other methods will make sure this is updated correctly.

```
//called periodically. checks whether predecessor has failed
public void check_predecessor(){
    if(this.predecessor != null && !this.predecessor.is_alive())
    predecessor = null;
}
```

Fault-tolerance

As our system was developed using a Decentralized Design, we have implemented *Chord*'s fault-tolerant features.

According to the *Chord Protocol*, when a node n fails, nodes whose finger tables include *n* must find *n*'s successor. Besides that, the failure of that node *n* must not be allowed to disrupt queries that are in progress as the system is re-stabilizing. Therefore, the most important thing about fault-tolerance in *Chord* is maintaining the nodes' successors actualized.

To achieve that, instead of saving just their first successor, all nodes store a list with their *r* nearest successors in the *Chord* Ring. That way, if a node n fails, the nodes whose finger tables include n have easy access to it's next successor.

In this project, we set *r* to half of the value of *m* and created a field in the Node class called *listOfSuccessors*, that is a *SimpleNode* array which is updated on the stabilize function that is called periodically:

```
//called periodically.
// verifies n's immediate successor and tell the successor about n
public void stabilize() throws Exception {
    int i = 0;
    while(i + 1 < this.listOfSuccessors.length) {
        SimpleNode x;

        if(!this.getSuccessor().id.equals(this.id) &&
!this.listOfSuccessors[i].is_alive()){
            this.listOfSuccessors[i] = this.listOfSuccessors[i+1];
        }

        if (this.listOfSuccessors[i].id.equals(this.id))
            x = this.predecessor;
        else
            x = this.listOfSuccessors[i].find_predecessor();

        if (x != null && !this.id.equals(x.id) &&
(this.id.equals(this.listOfSuccessors[i].id) || isBetween(x.id, this.id,
this.listOfSuccessors[i].id))) {
            this.listOfSuccessors[i] = x;
            this.listOfSuccessors[i+1] =
(this.listOfSuccessors[i].id.equals(this.id)) ? this.listOfSuccessors[0]
: this.listOfSuccessors[i].getSuccessor();

        if(i == 0)
```

```
            fingerTable[0].node = x;
        }

        if (!this.listOfSuccessors[i].id.equals(this.id) && i == 0)
            if (!this.listOfSuccessors[i].notifyIntern(this))
                this.listOfSuccessors[i] = this;

        i++;
    }
}
```

Besides that, each time the peers communicate, through *SSLEngine*, we have a verification to prevent the protocols from stop working because of the sudden unavailability of a peer. An example of this situation is in the backup protocol, in the class *Backup*, namely in the run function, as shown below, we have a verification where we confirm if the connection and messages exchange between the peers worked. If in that time, the peer that was supposed to backup the file fails, the initiator peer stops sending backup messages to that peer and tries to backup the file in the next successor.

```
SSLEngineClient client;
try {
    client = new SSLEngineClient("TLSv1.2", sn.getAddress(),
sn.getPort());

    client.connect();
    client.write(message);
    client.read();
    client.shutdown();
}catch (Exception e) {
    successful = false;
    break;
}
```

Lastly, if a peer ends its connection, leaving the *Chord* Ring, we have implemented a function in the *Peer* class called shutdown (presented below) to work in a similar way as the reclaim protocol. In this function, we set the peer available space to 0 and that way the peer executes the reclaim protocol and there is no loss of the files it had backed up.

```
public void shutdown() {
    try {
        System.out.println("\nSHUTDOWN SERVICE");
        var reclaims = PeerClient.getStorage().setOverallSpace(0, exec);
        reclaims.forEach((reclaim) -> {
            try {
                reclaim.get();
            }
            catch (Exception e) { e.printStackTrace(); }
```

```
        });
        stop();
        System.out.println();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```