SDIS

Sistemas Distribuídos

Project 2

Authors:	*Student number:*
António Cruz	up201603526
João Malheiro	up201605926
Ricardo Boia	up201505244
Tiago Castro	up201606186

June 1, 2019

Contents

1 Overview

In this report it is explained how the project was developed, namely, how the protocol implemented works and how we the code was structured to use the **Chord** protocol to create a decentralized distributed system. The backup service supports backups, restores, deletes and reclaims. These services are requested to the peer via the client *testapp*. The project addresses scalibility by using the **Chord** protocol to create a decentralized distributed backup service and has implemented some fault-tolerance features. A **threadpool** is used to manage the concurrency. **JSSE** and **NIO** are some of the other features implemented in the program.

2 Protocols

In this section, it is described the implementation of the RMI interface, as well as how the message are exchanged, their format and the rules. Each message had a brief explanation followed its format and its handling code.

2.1 RMI interface

The *TestApp* - client - and *Peer* are connected using *RMIStub*, that extends *Remote*. The client can send four types of request: backup, delete, restore and reclaim.

```java
public interface RMIStub extends Remote {

    /**
     * Interface method to call the Backup Protocol class
     * @param file  file
     * @param replicationDeg    replication degree
     * @throws RemoteException  exception
     */
    void backupProtocol(String file, int replicationDeg) throws
   RemoteException;

    /**
     * Interface method to call the Restore Protocol class
     * @param file  file
     * @throws RemoteException  exception
     */
    void restoreProtocol(String file) throws RemoteException;

    /**
     * Interface method to call the Delete Protocol class
     * @param file  file
     * @throws RemoteException  exception
     */
    void deleteProtocol(String file) throws RemoteException;

    /**
     * Interface method to call the Reclaim Protocol class
     * @param reservedSpace reserved space
     * @throws RemoteException  exception
     */
    void reclaimProtocol(int reservedSpace) throws RemoteException;
}
```

2.2 Messages

2.2.1 BackupInit

Message used to initiate the backup protocol. This message is used to provide the search for the responsible node for the objective fileHash. When a BackupInit message is received, that peer will search in its finger table for the responsible node. If it is found, it sends a backup ready message to the init peer with the peer in the finger table information. On the other hand, if the responsible node is not accounted for, another backup message will be deployed to the largest table jump. It should be noted that this message does not include the file byte data and that the protocol has 2 stages (init, file saving) .

`BACKUP-INIT {File Hash} {FILE Name} {FILE repDegree}`

2.2.2 BackupReady

Message sent to the initiation peer everytime a peer sees that one of its finger table values is responsible for the queried filehash. When the initiatior peer receives this, it then goes into the second stage of the protocol, the file saving stage. The peer is going to read the data from the file and send a Backup Message to the responsible peer so that it can save it in its peer disk.

`BACKUP-READY {File Hash} {FILE name} {FILE repDegree}`

2.2.3 Backup

This message includes the data from the requested file and when a peer receives it it will, after a rep degree check, save the data in its peer disk. Following that, the peer will then check if the file needs to be resent because of the repDegree. After saving, this class will forward another backup message to its sucessor unless the repDegree - 1 is equal to 0, in that case the chain will be stopped.

```
BACKUP {File Hash} {FILE repDegree} {FILE content}
```

2.2.4 BackupComplete

This message's purpose is to let the initiator peer know that the file has been saved.

```
BACKUP-COMPLETE {File Hash} {FILE repDegree}
```

2.2.5 GetPredecessor

This message is sent as part of the Stabilize protocol where each node periodically sends a message to its sucessor asking for its predecessor, if it responds with the hash of the sender then nothing happens, however, if its sucessor responds with a different hash than the node will change its sucessor to it.

```
GET_PREDECESSOR {Sender IP} {Sender PORT}
```

2.2.6 ResponsePredecessor

This message is sent after receiving a GetPredecessor messager with the information of the Peer's predecessor

```
RESPOSE_PREDECESSOR {Predecessor Hash} {Predecessor IP } {Predecssor PORT}
```

2.2.7 Lookup

This message is constantly being sent throughout the network it order to update each of the the Peer's fingertables. This is used to search for the sucessor of a certain hash. Upon receving this message the Peer will check if the queried hash is bigger than its own hash and smaller than its sucessor's, in which case it means that its sucessor is the node responsible for that hash. If not, the peer will search for the lowest hash in its fingertable that is higher than the queried hash and send a new Lookup message to it until the responsible node for that hash is found.

`LOOKUP {Queried Hash} {Sender IP} {Sender PORT}`

2.2.8 Successor

This message is sent as a response to a Lookup message. This informs the peer that originally sent the Lookup message who is the sucessor of the peer hash that he queried. This message is mostly used to keep the finger table updated. Upon receiving this message the Peer will compare the Queried Hash with the Sucessor Hash to determine which entry in the finger table it corresponds to.

`SUCESSOR {Queried Hash} {Sucessor Hash} {Sender IP} {Sender PORT}`

2.2.9 Predecessor

This message is used to inform the Peer that receives it that the Peer who sent it it is its predecessor.

`PREDECESSOR {Predecessor Hash} {Predecessor IP } {Predecssor PORT}`

2.2.10 Ping

This message is sent from a peer to its predecessor in order to check if it is still alive. When a peer receives this message, it is to send a *Pong* message to assure that it is not dead. Next is expressed the message's format followed by the code that represents de message's handler function content.

`PING {Sender IP} {Sender PORT}`

```
MessageForwarder.sendMessage(
    new PongMessage(ci.getIp(), ci.getPort())
    );
```

2.2.11 Pong

This message is sent from a peer to the peer who sent a *Ping* message to notify it that it is alive. When a peer receives this message, it is to notify the checkPredecessor function that the predecessor is still alive and let the thread resume its course. Next is expressed the message's format followed by the code that represents de message's handler function content.

`PONG`

```java
synchronized(Peer.checkPredecessor){
    CheckPredecessor.dead = false;
    Peer.checkPredecessor.notify();
}
```

2.2.12 ReclaimBackup

This message is sent once a file is deleted due to the Reclaim protocol. Seen as the file must be stored in another peer this message is sent to its sucessor to backup the deleted file, if the Peer that receives this message already owns the file (due to the replication degree of the file) it redirects the message to its sucessor.

`RECLAIM_BACKUP {File Hash} {File Name} {File Content}`

2.2.13 RestoreInit

Message that initiates the restore protocol. It contains both the fileHash and the filename. The searching phase of this protocol goes exactly the same has backupInit. It only deffers after the right peer is found. In the restore protocol, when that happens, the peers sends to the responsible node a RestoreMessage that will trigger the rest of the process.

`RESTORE-INIT {Sender Hash} {Sender IP} {Sender PORT} {File Hash} {File Name}`

2.2.14 Restore

Message sent when the restore init finds the responsible node for the respective filehash. When received, the peer will check if it owns the file in its backup protocol peer disk folder, if it does a restorefile will be sent to the init peer in order to complete the protocol. However, if it does not own the file due to a reclaim per example, a RestoreSucessor is going to be sent to its respective successor to search for the file.

`RESTORE {Sender Hash} {Sender IP} {Sender PORT} {File Hash} {File Name}`

2.2.15 RestoreFile

This is the final stage of the restore protocol, in which the data is sent to the init peer for a file to be stored in the restored folder. When this message is received, the peer that receives it will store the content bytes in the respective folder with the filename therefore restoring the file that was previously backed up.

```
RESTORE WITH FILE {Sender Hash} {Sender IP} {Sender PORT} {File Hash} {File Name}
```

2.2.16 RestoreSuccessor

This Message is sent only when the node that is responsible for a certain file does not have it (reclaim used on that peer, per example). In this case the file will probably be in the Peer's sucessors either due to the replication degree or due to the Reclaim protocol. Therefore this message will be sent to its sucessor until the file is found, where it will send the file to the Peer asking it, or this message returns to the Peer that sent it originally.

```
RESTORE-SUCESSOR {Sender Hash} {Sender IP} {Sender PORT} {File Hash} {File Name}
```

2.2.17 Delete

This message is sent as part of the Delete protocol where this message is sent to all the nodes in the network so that they can delete their local copies of the file. This must be sent to all nodes seen as we cannot keep count of what nodes had to reclaim space and therefore send their local copies of the file to another node, so it order to be sure of its deletion it must be sent to everyone.

```
DELETE {File Hash} {Sender Hash}
```

```java
int index;
if(receivedKey.equals(ChordManager.peerHash.toString())) {
    index = 0;
} else {
        for(index = 0; index < ChordManager.getM(); index++){
            String res = ChordManager.calculateNextKey(ChordManager.
    peerHash, index, ChordManager.getM());

            if(res.equals(receivedKey))
                break;
            }
        }
ChordManager.getFingerTable().set(index,ci);
```

3 Concurrency design

In this section, we explain how we dealt with concurrency of the different services present in each peer.

3.1 ScheduledExecutorService

The class *Peer* has an *ScheduledExecutorService* to which the threads will be submitted. The program uses threads for receiving (*PeerReceiver*) and sending message (*sendMessage*), to manage *Chord* - a main class (*ChordManager*) and three classes to stabilize it (*CheckPredecessor*, *FixFinger*, *Stabilize*) - and for the services offered - backup, delete, restore, reclaim (all classes are located in *protocol* package).

3.2 Java NIO

We also implemented Java NIO that, unlike JAVA IO does not block a thread until there is data to read / write or the data is fully written. Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.

4 JSSE

The message exchange between peers is made using **JSSE**, to assure the security of this communication. Before a *Peer* sends information to another, first he must authenticate himself.

```
private void setServerSocket()
{
    serverSocket = null;

    SSLServerSocketFactory serverSocketFactory = (
   SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

    try {
        serverSocket = (SSLServerSocket) serverSocketFactory.
   createServerSocket(port);
        serverSocket.setNeedClientAuth(true);
        serverSocket.setEnabledProtocols(serverSocket.
    getSupportedProtocols());
        System.out.println("Server socket thread created and ready to
    receive");
    } catch (IOException e) {
        System.err.println("Error creating server socket");
        e.printStackTrace();
    }
}
```

When a *Peer* is started, it creates a *PeerReceiver* thread that is responsible for setting the keystore and trustore, creating a *SSLServerSocket* and listens for communication requests. When a message is received, *PeerReceived* creates a *MessageHandler* thread that will redirect the message to the correct handler.

```
public void run() {
    Object messageObject = null;
    SSLSocket connectionSocket = null;

    while(true) {
        try {
            connectionSocket = (SSLSocket) serverSocket.accept();
        } catch (IOException e) {
            e.printStackTrace();
        }

        ObjectInputStream inFromClient = null;
        try {
            inFromClient = new ObjectInputStream(connectionSocket.
    getInputStream());
        } catch (IOException e) {
            e.printStackTrace();
        }

        if(connectionSocket != null){
            if(inFromClient != null) {
                try {
                    messageObject = inFromClient.readObject();
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                }
                MessageHandler mh = new MessageHandler(messageObject);
                Peer.executor.submit(mh);
            }
        }
    }
}
```

To send a message to a *Peer*, it is used a *SendMessage* thread that will create a *SSLSocket*, connected to the desired *SSLServerSocket*, do the *Handshake* protocol and send the information.

```
public void run() {
    System.out.println("Sending " + message + " to :  " + message.
getIpAddress() + message.getPort());

    if(message.getPort() == Peer.port)
        return;

    SSLSocketFactory socketFactory = (SSLSocketFactory)
SSLSocketFactory.getDefault();
    SSLSocket clientSocket;
    try {
        clientSocket = (SSLSocket) socketFactory.createSocket(
InetAddress.getByName(message.getIpAddress()), message.getPort());
        clientSocket.startHandshake();

        ObjectOutputStream outToServer = new ObjectOutputStream(
clientSocket.getOutputStream());
        outToServer.writeObject(message);
    } catch (Exception e) {
        System.out.println("User disconnected");
    }
}
```

5 Scalability

In order to make the system as scalable as possible, we used the **Chord** protocol to create a decentralized distributed backup service. This means that there are no servers in the system and that each *peer* is equally responsible for keeping the service working correctly. The *ChordManager* class is responsible for managing the chord aspects in each *Peer*.

When a *Peer* joins the system, he will create a thead *ChordManager* that will generate a key, get the *Peer*'s successor and initilize the finger table. It will also start a *FixFinger* thread that, along with the *Stabilize* thread initialized in class *Peer*, will be responsible for the stabilization of the protocol.

5.1 Stabilize

This process is run on a thread every 500ms. First, it asks the *Peer*'s sucessor for its predecessor, then it will check if that predecessor is the *Peer*'s new successor and, if it is, it will set it accordingly. Finally, it will send a message to the *Peer*'s successor notifying it that he might need to update its predecessor.

```java
public class Stabilize implements Runnable {
    @Override
    public void run() {
        try {
            if(ChordManager.getFingerTable().get(0).getPort() == Peer.port && ChordManager.predecessor != null){
                ChordManager.getFingerTable().set(0, ChordManager.predecessor);
            } else if(ChordManager.getFingerTable().get(0).getPort() != Peer.port ){
                MessageForwarder.sendMessage(new GetPredecessorMessage(new ConnectionInfo(null,InetAddress.getLocalHost().getHostAddress(),Peer.port), ChordManager.getFingerTable().get(0).getIp() , ChordManager.getFingerTable().get(0).getPort()));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.2 FixFinger

This process is run on a thread every 500ms and is responsible for updating the finger table. It checks who is the *Peer* resposible for the finger table entry, using the *ChordMamager*'s method *searchSuccessor2*, and, if it is the *Peer* or its successor, it will update the finger table and end, if it isn't, it will send a *LOOKUP* message to the *Peer*, present in the finger table, with the largest key, that is smaller than the wanted key.

```java
public class FixFingers implements Runnable {

    private int index = -1;

    @Override
    public void run() {
        index++;

        if(index == ChordManager.getM() ) {
            ChordManager.printFingerTable();
            index = 0;
        }

        String key = ChordManager.calculateNextKey(ChordManager.peerHash, index, ChordManager.getM() );
        ArrayList<ConnectionInfo> fingerTable = ChordManager.getFingerTable();

        if(index > (fingerTable.size() - 1)) {
            try {
                fingerTable.add(new ConnectionInfo(ChordManager.peerHash, InetAddress.getLocalHost().getHostAddress(), Peer.port));
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }

        Message res = null;
        try {
            res = ChordManager.searchSuccessor2(new ConnectionInfo(new BigInteger(key), InetAddress.getLocalHost().getHostAddress(), Peer.port));
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }

        if(res != null){
            if(res instanceof SucessorMessage) {
                fingerTable.set(index, ((SucessorMessage) res).getCi());
            }
            else if (res instanceof LookupMessage)
            {
                MessageForwarder.sendMessage(res);
            }
        }
    }
}
```

6 Fault-tolerance

In order to keep the system working like expected, fault-tolerance mechanisms are needed to be implemented. Unfortunately we weren't able to fully implement fault-tolerance on our project and we can't say our implementation is completely fault-tolerant. What we were able to implement is the CheckPredecessor process.

6.1 CheckPredecessor

This process is ran by each peer on a thread on a set interval of 1s. This process sends a *PING* message to the peer's predecessor in order to check if it is alive, it then waits a set timeout of 500ms for an answer. If the answer never comes (in the format of a *PONG* message), the predecessor is considered dead and set to null.

```java
public class CheckPredecessor implements Runnable{
    public static boolean dead;
    private int timeout;

    public CheckPredecessor(int timeout){
        this.timeout = timeout;
    }

    @Override
    public void run(){
        synchronized(this){
            if (ChordManager.predecessor != null) {
                try {
                    MessageForwarder.sendMessage(new PingMessage(new ConnectionInfo(null, InetAddress.getLocalHost().getHostAddress(), Peer.port), ChordManager.predecessor.getIp(), ChordManager.predecessor.getPort()));
                    this.wait(timeout);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                if (dead) {
                    ChordManager.predecessor = null;
                }
            }
        }
    }
}
```