Mestrado Integrado em Engenharia Informática e Computação
Sistemas Distribuídos

Distributed Backup Service for the Internet

Final Report

Grupo:
José Manuel Faria Azevedo - up201506448
Hugo Daniel Gonçalves Fernandes - up201909576
Henrique José de Castro Ferreira - up201605003

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

May 22, 2020

Contents

1 Overview

In this section, we'll describe the main features of our project. Our project consists in a distributed backup system, which can be used to backup files in different systems, each one of them called a *Peer*.

For this purpose, we implemented 4 protocols:

- **Backup**: Used when a peer wants to backup a file stored in its system;
- **Restore**: Used when a peer wants to restore a previously backed up file in one of the other peers;
- **Delete**: Used when a peer wants to delete a previous backed up file in all the peers;
- **Reclaim**: Used when a peer to reclaim space in its file system for backup files.

The main features of our projects are:

- It has a decentralized implementation, using *thread-pools* to manage the various requests;
- It uses *RMI* interface;
- It uses *JSSE SSL Sockets* to communicate between peers;
- It uses *Chord*, for peers and files management.

2 Protocols

In this section we will address each of the protocols implemented detailing the messages sent and presenting the code implemented. The main classes involved in each protocol are:

- *TestApp*: class which handled user inputs;
- *Peer*: class responsible for initiating the protocol;
- *SendMessageThread*: class responsible to send messages to other peers (more details provided on section JSSE);
- *Receiver*: class responsible for receiving messages from other peers;
- *MessageManagement*: class responsible for handling tasks from messages received and then initiating the process of responding to the message sender;
- *Storage*: auxiliary class responsible for handling operations related to the file system.

2.1 RMI Protocol

To connect the *TestApp* and the *Peer* RMI was used, *RMIInterface* that extends *Remote*, this interface declares the four type of requests the client can send: Backup, Restore, Delete and Reclaim.

```
public interface RMIInterface extends Remote {
    void backupProtocol(String pathname, int replicationDegree) throws
        RemoteException;
    void restoreProtocol(String file) throws RemoteException;
    void deleteProtocol(String pathName) throws RemoteException;
    void reclaimProtocol(int newStorageSpace)throws RemoteException;
}
```

The RMI implementation was also handled in the creation of each peer, which binded it using its id:

```
try {
    RMIInterface stub = (RMIInterface) UnicastRemoteObject.exportObject(instance,
        0);
    try {
        Registry reg = LocateRegistry.getRegistry();
        reg.rebind(Integer.toString(id), stub);
        System.out.println("peer.Peer connected through getRegistry");
    } catch (Exception e) {
        Registry reg = LocateRegistry.createRegistry(1099);
        reg.rebind(Integer.toString(id), stub);
        System.out.println("peer.Peer connected through createRegistry");
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

2.2 Backup

To run the backup protocol, we run the command:

```
java TestApp <peer_id> BACKUP <filepath> <replication_degree>
```

in which *peer_id* represents the peer id of the initiator peer, *filepath* represents the file path we wish to backup and *replication_degree* which represents the number of times we wish to replicate the file.

The backup protocol using the Finger table generated from the *Chord* algorithm, selects the peer(s) which will backup the file. Then the file is subdivided in chunks of 16 kilobytes each and sent to the peer selected using *SendMessageThread* and a PUTFILE message:

```
public void backupProtocol(String pathname, int replicationDegree) {
    File f = new File("files/" + pathname);
    String fileId = Util.sha256(f.getName() + f.lastModified());
    if(f.length()<=storage.spaceAvailable) {
        storage.backedUp.put(fileId, replicationDegree);
        try {
            //send message
            ArrayList<String> fingerTable = ChordManager.getFingerTable();
            ArrayList<Integer> usedFingers = new ArrayList<>(fingerTable.size());
            int i = 0;
            while (i < fingerTable.size()) {
                if (storage.getReplicationsInSystem(fileId) >= replicationDegree)
                    break;
                int targetPort = Integer.parseInt(fingerTable.get(i).split(" ")[2])
                    ;
                if (!usedFingers.contains(targetPort)) {
                    FileInputStream fis = new FileInputStream(f);
                    BufferedInputStream bst = new BufferedInputStream(fis);
                    int chunkMax = (int) Math.ceil((float) f.length() / (float)
                        (1000 * 16));
                    byte[] buff = new byte[1000 * 16];
                    int j;
                    int chunkNo = 0;
                    while ((j = bst.read(buff)) > 0) {
                        byte[] chunk = Arrays.copyOf(buff, j);

                        //putfile message
                        String header = "PUTFILE " + fileId + " " +
                            replicationDegree + " " + chunkNo + " " + chunkMax + " "
                             + Peer.port;
```

```
                        Runnable sendMessageThread =new SendMessageThread(header,
                            chunk, targetPort);
                        executor.execute(sendMessageThread);
                        chunkNo++;
                    }
                    usedFingers.add(targetPort);
                }
                i++;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The PUTFILE message has the following format:

```
PUTFILE <file_id> <replication_degree> <chunk_no> <chunk_max> <sender_id> <CRLF><
    chunk_body>
```

In which *file_id* represents the encrypted name of the file which is being backed up, *replication_degree* which is the number of times the file is being backed up, *chunk_no* which is the number of the chunk being backup, *chunk_max* which is the total number of chunks being backed up, *sender_id* which is the id of the peer sending the message and *chunk_body* which is the content from the chunk being backed up.

After receiving a PUTFILE message through the *Receiver* class, the peer stores the chunk and when all chunks have been received sends a STORED message to the peer that requested the backup:

```
private void putFile() {
    Set<String> map = Peer.getStorage().repDegree.keySet();
    for(String f: map){
        if(f.equals(msgHeader[1]));
        {
            return;
        }
    }
    //verifying if there is space for the
    byte[] data = getFileData();
    if(Peer.getStorage().totalSpace<data.length){
        System.out.println("not enough space for this file data:" + data.length
            /1024);
        return;
    }else {
        //save the data transmitted in the PUTFILE Message
        String fileId = msgHeader[1];
        int repDegree = Integer.parseInt(msgHeader[2]);
        int chunkNo =Integer.parseInt(msgHeader[3]);
        int chunkMax =Integer.parseInt(msgHeader[4]);
        int portOfSender = Integer.parseInt(msgHeader[5]);

        boolean fileComplete = Peer.getStorage().addChunk(fileId,chunkNo,data,
            chunkMax,repDegree);

        //if the file is complete send stored of file
        if(fileComplete){
            //Final part creating and sending the STORED message
            String msg = "STORED" + " " + fileId + " " + (char) 0xD + (char) 0xA +
                (char) 0xD + (char) 0xA;
            ArrayList<Integer> usedFingers = new ArrayList<>(ChordManager.
                getFingerTable().size());
            int i = 0;
```

```
            while (i < ChordManager.getFingerTable().size()) {
                int targetPort = Integer.parseInt(ChordManager.getFingerTable().get
                    (i).split(" ")[2]);
                if (!usedFingers.contains(targetPort)) {
                    usedFingers.add(targetPort);
                    Thread t = new Thread(new SendMessageThread(msg, portOfSender))
                        ;
                    t.start();
                    //Waiting for the random amount of milliseconds
                    try {
                        Thread.sleep((long) (Math.random() * 400));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                i++;
            }
        }else{
            System.out.println("Saved chunk number " + chunkNo + " from file " +
                fileId + " out of " + chunkMax + " chunks" );
        }
    }
}
```

The STORED message has the following format:

```
STORED <file_id> <CRLF>
```

in which *file_id* represents the encrypted name of the file which was backed up.

The protocol repeats the whole process until the replication degree is met, or the number of active peers.

2.3 Restore

To run the restore protocol, we run the command:

```
java TestApp <peer_id> RESTORE <filepath>
```

in which *peer_id* represents the peer id of the initiator peer and *filepath* represents the file path we wish to restore.

The reclaim protocol sends a GETFILE message to each peer in its Finger Table, or if the file exists in the *Peer* backup folder,due to other peers asking for backups it restores the file itself instead of sending the message:

```
public void restoreProtocol(String fileName){
    File f = new File("files/"+ fileName);
    String fileId = Util.sha256(f.getName() + f.lastModified());

    File backed = new File("./fileSystem/" + Peer.getID() + "/backup/" + fileId);
    if(backed.exists()){
        File restored = new File("./fileSystem/" + Peer.getID() + "/restore/" +
            fileName);
        try {
            Files.copy(backed.toPath(), restored.toPath(), StandardCopyOption.
                REPLACE_EXISTING);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        ArrayList<Integer> usedFingers = new ArrayList<>(ChordManager.
            getFingerTable().size());
```

```
        int i = 0;
        while (i < ChordManager.getFingerTable().size()) {
            int targetPort = Integer.parseInt(ChordManager.getFingerTable().get(i).
                split(" ")[2]);
            if (!usedFingers.contains(targetPort)) {
                Runnable sendMessageThread =new SendMessageThread("GETFILE " +
                    fileId + " " + fileName + " " + Peer.port,targetPort);
                executor.execute(sendMessageThread);
                usedFingers.add(targetPort);
            }
            i++;

        }
    }
}
```

The GETFILE message has the following format:

```
GETFILE <file_id> <file_name> <sender_id> <CRLF>
```

in which *file_id* represents the encrypted name of the file which is being restored, *file_name* which is the name of the file being restored and *sender_id* which is the id of the peer sending the message.

Upon receiving the message GETFILE, if the peer has the file backed up, it subdivides the file in chunks of 16 kilobytes and sends them to the peer which requested them, using a FILE message:

```
private void getFile() {
    File fileName = new File("files/" + msgHeader[2]);
    try {
        FileInputStream f = new FileInputStream("./fileSystem/" + Peer.getID() + "/
            backup/" + msgHeader[1]);
        BufferedInputStream bst = new BufferedInputStream(f);
        byte[] buff = new byte[1000 * 16];
        int chunkMax = (int)Math.ceil((float)fileName.length() / (float)(1000*16));
        int j;
        int chunkNo = 0;
        while((j = bst.read(buff)) > 0){
            byte[] chunk = Arrays.copyOf(buff,j);

            //putfile message
            Thread t = new Thread(new SendMessageThread("FILE " + msgHeader[1] + "
                " + chunkNo + " " + msgHeader[2] + " "+  chunkMax, chunk, Integer.
                parseInt(msgHeader[3])));
            t.start();
            chunkNo++;
        }

    } catch (FileNotFoundException e) {
        System.out.println("Peer does not contain file");
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

The FILE message has the following format:

```
PUTFILE <file_id> <chunk_no> <file_name> <chunk_max> <CRLF><chunk_body>
```

in which *file_id* represents the encrypted name of the file which is being restored, *chunk_no* which is the number of the chunk being restored, *file_name* which is the name of the file being restored, *chunk_max* which is the total number of chunks being restored and *chunk_body* which is the content from the chunk being restored.

After receiving a FILE message, the peer stores the chunk and when all its chunks are received, the peer restores the file:

```
private void file() {
    byte[] chunkData= getFileData();
    Peer.getStorage().addRestoredChunk(msgHeader[1], msgHeader[3], Integer.parseInt
        (msgHeader[2]),chunkData, Integer.parseInt(msgHeader[4]));
}

public boolean addRestoredChunk(String fileId, String fileName, int index, byte[]
    data,int maxChunks){
    //if the fileid is in the map add that chunk to the Array
    if(restoredChunks.containsKey(fileId)) {
    }
    else{
        //if the fileis isnt in the map add a new entry to the map with a null
            filled array and the fileid
        ArrayList<byte[]> temp = new ArrayList<>();
        for(int i = 0; i<maxChunks; i++)
            temp.add(null);
        restoredChunks.put(fileId,temp);
    }
    restoredChunks.get(fileId).set(index,data);

    //checks if any of the chunks is still missing if there is any missing stops
        the method
    for(int i = 0; i<restoredChunks.get(fileId).size();i++){
        if(restoredChunks.get(fileId).get(i)==null){
            return false;
        }
    }

    //if the array is filed it starts to create the file
    File temp = new File("./fileSystem/" + Peer.getID() + "/restore/" + fileName);
    if(!temp.exists()) {
        byte[] fileData = new byte[0];
        ByteArrayOutputStream outputMessageStream = new ByteArrayOutputStream();
        for (int i = 0; i < restoredChunks.get(fileId).size(); i++) {
            try {
                outputMessageStream.write(Arrays.copyOf(fileData, fileData.length))
                    ;
                outputMessageStream.write(Arrays.copyOf(restoredChunks.get(fileId).
                    get(i), restoredChunks.get(fileId).get(i).length));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        byte[] data2 = outputMessageStream.toByteArray();
        try {
            FileOutputStream streamToFile = new FileOutputStream("./fileSystem/" +
                Peer.getID() + "/restore/" + fileName);
            streamToFile.write(data2);
            streamToFile.close();
        } catch (IOException e) {
            System.out.println("Could not Create file");
        }
    }
```

```
    return true;
}
```

2.4 Delete

To run the delete protocol, we run the command:

```
java TestApp <peer_id> DELETE <filepath>
```

in which *peer_id* represents the peer id of the initiator peer and *filepath* represents the file path we wish to delete.

The delete protocol deletes all the backed up files in the initiator and then sends a DELETE MESSAGE to each peer in its Finger Table:

```
public void deleteProtocol(String pathName) {
    File f = new File("files/" + pathName);
    String fileId = Util.sha256(f.getName() + f.lastModified());
    File backed = new File("./fileSystem/" + Peer.getID() + "/backup/" + fileId);
    if(backed.exists()) backed.delete();
    if(storage.backedUp.containsKey(fileId)) {
        storage.backedUp.remove(fileId);
    }
    else{
        System.out.println("This peer did not backup the specified file");
    }
        //message
        String header = "DELETE " + fileId + " " + Peer.port + " " + (char) 0xD + (
            char) 0xA + (char) 0xD + (char) 0xA;

        ArrayList<String> fingerTable = ChordManager.getFingerTable();
        ArrayList<Integer> usedFingers = new ArrayList<>(fingerTable.size());
        int i = 0;
        while (i < fingerTable.size()) {
            int targetPort = Integer.parseInt(fingerTable.get(i).split(" ")[2]);
            if (!usedFingers.contains(targetPort)) {
                Thread t = new Thread(new SendMessageThread(header, targetPort));
                t.start();
                usedFingers.add(targetPort);
            }
            i++;
        }
    storage.backedUp.remove(fileId);
}
```

The DELETE message has the following format:

```
DELETE <file_id> <sender_id> <CRLF>
```

in which *file_id* represents the encrypted name of the file which is being restored and *sender_id* which is the id of the peer sending the message.

After receiving the DELETE message, the peer verifies if it has the file backed up in its system, and if it has removes it:

```
private void delete() {
    //getting the id of the file
    String fileId = msgHeader[1];
    int portOfSender = Integer.parseInt(msgHeader[2]);
    if(Peer.getStorage().backedUp.containsKey(fileId)) {
        Peer.getStorage().backedUp.remove(fileId);
```

```
    }
    //checking for the file
    File f = new File("fileSystem/" + Peer.getID() +"/backup/" + fileId);
    if(f.exists()){
        f.delete();
        Peer.getStorage().repDegree.remove(fileId);
    }
}
```

2.5 Reclaim

To run the reclaim protocol, we run the command:

```
java TestApp <peer_id> RECLAIM <storage_space>
```

in which *peer_id* represents the peer id of the initiator peer and *storage_space* represents the space we want to reclaim in disk, in bytes.

The reclaim protocol, verifies if it value provided in the argument is higher or lower than the space already occupied in disk. If its higher, it simply changes the total space value. If its lower, it verifies if it still has empty space after reducing the space and if not, it deletes files and backs them up in other peers, until has some free space (process similar to the backup protocol):

```
public void reclaimProtocol(int newStorageSpace) {
    //checks if the current space is smaller then the new storage space
    if(newStorageSpace >= storage.totalSpace){
        storage.spaceAvailable+=newStorageSpace-storage.totalSpace;
        storage.totalSpace=newStorageSpace;
    }else{
        storage.spaceAvailable=newStorageSpace-(storage.totalSpace-storage.
            spaceAvailable);
        if(0 < storage.spaceAvailable){
            storage.totalSpace=newStorageSpace;
        }else {
            System.out.println("Space needed: " + Math.abs(storage.spaceAvailable))
                ;
            //in case of needing to remove files the storage will remove the oldest
                 files first until there is enough space
            ArrayList<File> filesToDelete = new ArrayList<>();
            Set<String> map = storage.repDegree.keySet();
            for (String fileId : map) {
                if (0 < storage.spaceAvailable) {
                    break;
                } else {
                    File backed = new File("./fileSystem/" + Peer.getID() + "/
                        backup/" + fileId);
                    reclaimBackup(fileId);
                    filesToDelete.add(backed);
                    storage.spaceAvailable += backed.length();
                    if (storage.spaceAvailable > storage.totalSpace) storage.
                        spaceAvailable = storage.totalSpace;
                    try {
                        Thread.sleep((long) (Math.random() * 400));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
            storage.totalSpace = newStorageSpace;
            try {
                Thread.sleep(2000);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            for (File file : filesToDelete) {
                file.delete();
            }
        }
    }
    System.out.println("New storage available space is:" + storage.spaceAvailable);
}

private void reclaimBackup(String fileId){
    File f = new File("fileSystem/" + Peer.id + "/backup/" + fileId);
    if(f.length()>=storage.spaceAvailable) {
        try {
            //send message
            ArrayList<String> fingerTable = ChordManager.getFingerTable();
            ArrayList<Integer> usedFingers = new ArrayList<>(fingerTable.size());
            int i = 0;
            while (i < fingerTable.size()) {
                if (storage.getReplicationsInSystem(fileId) > storage.repDegree.get
                    (fileId)) {
                    break;
                }
                int targetPort = Integer.parseInt(fingerTable.get(i).split(" ")[2])
                    ;
                if (!usedFingers.contains(targetPort)) {
                    usedFingers.add(targetPort);
                    FileInputStream fis = new FileInputStream(f);
                    BufferedInputStream bst = new BufferedInputStream(fis);
                    int chunkMax = (int) Math.ceil((float) f.length() / (float)
                        (1000 * 16));
                    byte[] buff = new byte[1000 * 16];
                    int j;
                    int chunkNo = 0;
                    while ((j = bst.read(buff)) > 0) {
                        byte[] chunk = Arrays.copyOf(buff, j);

                        //putfile message
                        String header = "PUTFILE " + fileId + " " + storage.
                            repDegree.get(fileId) + " " + chunkNo + " " + chunkMax +
                             " " + Peer.port;
                        Runnable sendMessageThread =new SendMessageThread(header,
                            chunk, targetPort);
                        executor.execute(sendMessageThread);
                        chunkNo++;
                    }
                    Runnable sendMessageThreadRemoved =new SendMessageThread("
                        REMOVED " + fileId +  " " + (Peer.getStorage().
                        getReplicationsInSystem(fileId)-1), targetPort);
                    executor.execute(sendMessageThreadRemoved);
                    fis.close();
                    bst.close();
                }
                i++;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3 Concurrency Design

3.1 Thread Pools

To enhance the concurrency capability of the implementation *ScheduledExecutorService* was used to create a pool of threads that are then used to run different components of the program while ensuring concurrency such as:

- *ChordManager* - Class in charge of managing the *Chord* implementation.
- *SendMessageThread* - Class responsible for the sending of messages.
- *Receiver* - Class responsible for dealing with receiving messages.

```
public static ScheduledExecutorService executor;
////
executor = Executors.newScheduledThreadPool(100);
////
Runnable receiver = new Receiver(port);
executor.execute(receiver);
```

4 JSSE

The communication between peers in our project uses JSSE to increase the level of security of the connection between peers.

4.1 Receiver

When a peer creates its *Receiver* process said process will define the keystore and truststore and will provide the peer with its own server socket to listen for messages sent by other peers.

```
public Receiver(int port){
    this.port = port;

    System.setProperty("javax.net.ssl.keyStore", "server.keys");
    System.setProperty("javax.net.ssl.keyStorePassword", "123456");
    System.setProperty("javax.net.ssl.trustStore", "truststore");
    System.setProperty("javax.net.ssl.trustStorePassword", "123456");

    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();
    }
}
```

After being initiated the *Receiver* will listen for messages and when one is received it will redirect the message to the *MessageManagement* thread, which will deal with the information in the message it self.

```
public void run() {
    SSLSocket connectionSocket = null;
    while(true) {
        try {
```

```
            connectionSocket = (SSLSocket) serverSocket.accept();
        } catch (IOException e) {
            e.printStackTrace();
        }
        InputStream inFromClient = null;
        try {
            inFromClient = connectionSocket.getInputStream();
        } catch (IOException e) {
            e.printStackTrace();
        }
        DataInputStream in = new DataInputStream(inFromClient);
        if(connectionSocket != null){
            if(inFromClient != null) {
                byte[] buffer = new byte[64000];
                byte[] data = new byte[64000];
                try {
                    int readsize = in.read(buffer);
                    data = Arrays.copyOfRange(buffer, 0, readsize);
                } catch (IOException e) {
                    e.printStackTrace();
                }
                Peer.executor.execute(new MessageManagement(data));
            }
        }
    }
}
```

4.2 SendMessageThread

The last component of JSSE communication cycle is the *SendMessageThread* class, responsible for the sending of messages by connecting to the target *ServerSocket*, doing the *SSLSocket Handshake* and starting the transfer of data.

```
public void run() {
    if(port==Peer.port) return;
    SSLSocketFactory socketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault
        ();
    SSLSocket clientSocket = null;
    try {
        clientSocket = (SSLSocket) socketFactory.createSocket(InetAddress.
            getLocalHost().getHostAddress(), port);
        clientSocket.startHandshake();
        OutputStream outToServer = null;
        try {
            outToServer = clientSocket.getOutputStream();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        try {
            DataOutputStream out = new DataOutputStream(outToServer);
            if (this.chunk != null) {
                byte[] c = new byte[message.getBytes().length + chunk.length];
                System.arraycopy( message.getBytes(), 0, c, 0, message.getBytes().
                    length);
                System.arraycopy(chunk, 0, c, message.getBytes().length, chunk.
                    length);
                out.write(c);
            }
            else{
                out.write(message.getBytes());
```

```
            }

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    } catch (IOException e) {
        System.out.println("Port not available to receive message");
    }
}
```

5 Scalability

In this section we address scalability, in specific we address our use of *Chord* to ensure our system is scalable.

5.1 Chord

Chord was the way we found best to implement this project in a decentralized way. We found while reaserching that the implementation of *Chord* is mainly comprised of three parts these being: *ChordManager*, *FixFingers* and *Stabilize*.

ChordManager is the class in charge of handling the creation of a key for the peer and the start of the Chord protocol, this class is created when a peer is created and creates the thread of the *FixFinger* class as well.

```
private void setChord() throws UnknownHostException {
    String [] params = new String[] {String.valueOf(Peer.port), InetAddress.
        getLocalHost().getHostAddress()};
    ChordManager.peerHash = encrypt(params);
    System.out.println("Peer hash = " + peerHash + "\n");

    (new File("fileSystem/"+Peer.getID()+"/backup")).mkdirs();
    (new File("fileSystem/"+Peer.getID()+"/restore")).mkdirs();

    FingerTable();
    printFingerTable();

    FixFingers ff = new FixFingers();
    Peer.executor.scheduleAtFixedRate(ff,0,500, TimeUnit.MILLISECONDS);
}

public static String calculateNextKey(BigInteger hash, int index, int m){
    BigInteger add = new BigInteger(String.valueOf((int) Math.pow(2, index)));
    BigInteger mod =  new BigInteger(String.valueOf((int) Math.pow(2, m)));

    BigInteger res = hash.add(add).mod(mod);
    return res.toString();
}
```

The *FixFingers* class is the one responsible for updating the finger table, taking the identity of the peer in a finger table entry and verifying if it is already a successor or unknown with the help of *ChordManager*, if said peer isn't a successor the peer will proceed to look for said table entry by sending a LOOKUP message.

Seaching for peer as successor:

```
public static String searchSuccessor(String senderInfo){
    BigInteger successorKey = new BigInteger(fingerTable.get(0).split(" ")[0]);
    if(numberInInterval(peerHash, successorKey, new BigInteger(senderInfo.split(" "
        )[0]))){
        return "SUCESSOR " + fingerTable.get(0).split(" ")[0] + " " + fingerTable.
            get(0).split(" ")[1] +
                " " + fingerTable.get(0).split(" ")[2] + " " +senderInfo.split(" ")
                    [0] + " "+
```

```
                senderInfo.split(" ")[1] + " " + senderInfo.split(" ")[2];
        }
        else {
            for(int i = fingerTable.size()-1; i >= 0; i--){
                if(fingerTable.get(i).split(" ")[0] == null){
                    continue;
                }
                if(numberInInterval(peerHash, new BigInteger(senderInfo.split(" ")[0]),
                     new BigInteger(fingerTable.get(i).split(" ")[0]))) {
                    if(fingerTable.get(i).split(" ")[0].equals(ChordManager.peerHash)){
                        continue;
                    }
                    return "LOOKUP " + senderInfo.split(" ")[0] + " " +senderInfo.split
                        (" ")[1] + " " + senderInfo.split(" ")[2] + " "
                    + null + " " + fingerTable.get(i).split(" ")[1] + " " + fingerTable
                        .get(i).split(" ")[2];
                }
            }
            try {
                return "SUCESSOR " + ChordManager.peerHash.toString() + " " +
                    InetAddress.getLocalHost().getHostAddress() +
                        " " + Peer.port + " " + senderInfo.split(" ")[0] + " "+
                            senderInfo.split(" ")[1] + " " + senderInfo.split(" ")[2];
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
```

Updating the finger table:

```
    public void run() {
        index++;
        if(index == ChordManager.getBits() ) {
            ChordManager.printFingerTable();
            index = 0;
        }
        String key = ChordManager.calculateNextKey(ChordManager.peerHash, index,
            ChordManager.getBits() );
        ArrayList<String> fingerTable = ChordManager.getFingerTable();
        if(index > (fingerTable.size() - 1)) {
            try {
                fingerTable.add(ChordManager.peerHash + " " + InetAddress.getLocalHost
                    ().getHostAddress()+ " " + Peer.port);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
        String msg = null;
        try {
            msg = ChordManager.searchSuccessor(new BigInteger(key).toString() + " " +
                InetAddress.getLocalHost().getHostAddress() + " " + Peer.port);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
        if(msg != null){
            if(msg.split(" ")[0].equals("SUCESSOR")) {
                fingerTable.set(index, ((msg.split(" ")[1] + " " + msg.split(" ")[2] +
                    " " + msg.split(" ")[3])));
```

```
        }else if (msg.split(" ")[0].equals("LOOKUP"))
        {
            Peer.executor.execute(new SendMessageThread(msg));
        }
    }
}
```

Finally the *Stabilize* class is in charge of checking periodically if the architecture of the system is being maintained, by asking the successors of a particular peer if their predecessor is also a successor of the same particular peer if it is then it tells the successor peer to update its predecessors.

This creates the structure of the *Chord* finger tables by making sure that if Peer 1 has both Peer 2 and Peer 3 as successors and Peer 3 has Peer 2 as a predecessor, Peer 3 can then just update to say it has Peer 1 as a predecessor.

```
public void run() {
        try {
            if(ChordManager.getFingerTable().get(0).split(" ")[2].equals(Integer.
                toString(Peer.port)) && ChordManager.predecessor != null){
                ChordManager.getFingerTable().set(0, ChordManager.predecessor);
            } else if(!ChordManager.getFingerTable().get(0).split(" ")[2].equals(
                Integer.toString(Peer.port))){
                Thread t = new Thread(new SendMessageThread("GETPREDECESSOR " +
                    null + " " + InetAddress.getLocalHost().getHostAddress() + " " +
                     Peer.port + " " +
                      null +" " +  ChordManager.getFingerTable().get(0).split(" ")
                          [1] + " " + ChordManager.getFingerTable().get(0).split(" "
                          )[2]));
                t.start();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

6 Fault-tolerance

Although we attempted to implement a fault-tolerance tool that determines when a peer was no longer online and updates the finger tables of other peers ultimately we were unsuccessful.

That being said due to the nature of the *Chord* implementation some of the peers that are created can fail without too many ramifications however the case also exists where a peer that bridges the gap between many peers is taken down and the system then becomes unusable or some peers become unreachable.