Mestrado Integrado em Engenharia Informática e Computação

Distributed Systems

Serverless Distributed Backup Service for the Internet

Grupo 22, Turma 2

29/05/2020

Professores:

- Pedro Alexandre Guimarães Lobo Ferreira Souto
- Hélder Fernandes Castro

Desenvolvido por:

- Henrique Maciel de Freitas, up201707046@fe.up.pt
- José Miguel Martins Gomes, up201707054@fe.up.pt
- Liliana Natacha Nogueira de Almeida, up201706908@fe.up.pt
- Mark Timothy Vasconcelos Meehan, up201704581@fe.up.pt

Index

Overview

This project was developed for the Course Unit of Distributed Systems and this report has the main goal of describing and explaining, in detail, the implementation of the system and the reason behind most design choices.

The implemented system is able to execute 5 main protocols:

- Backup - Send a file to the system and store it with a specified replication degree
- Restore - Retrieve a file from the system
- Reclaim - Set a Peer's max space to limit its maximum capacity (in bytes)
- Delete - Delete all copies of a file from the system
- State - Retrieve all relevant information about a Peer

The architecture on which the system was built is a Serverless Peer-to-Peer architecture based on the **Chord** algorithm along with its fault-tolerant features. All Peers and Clients communicate through TCP sockets using JSSE **SSLSockets** to ensure **security** and **reliability** and make use of threads to ensure maximum **scalability**.

Architecture

Chord

The overall system was designed around the **Chord** algorithm and all the design decisions were made taking that into account.

To store a Peer's information a **ChordInfo** class was created. This class stores an IP address, a Port and a hashKey that is generated by passing the IP and Port through a SHA-1 hashing algorithm. This class is used by every Peer to store their communication data and to store "references" to other peers that they might know about.

The main logic behind the Chord algorithm resides in a Message System as will be described in the next section but the classes **ChordNode**, **FixFingers** and **Stabilize** are responsible for initiating the sending of said messages or controlling some inner information required by Chord (e.g Finger Tables, Successor Lists, etc).

Message Structure

The communication between Peers is made, as previously said, by **SSLSockets**. As such, a Message class was created. This abstract class implements "Serializable" in order to be sent through the SSLSocket as a Java Object and has three main fields: the IP Address on which to connect to, the Port associated with the connection and a **ChordInfo** object that contains information about the Peer that sent the Message. The class has an abstract method of type *void* called **handle()**. This method is called by a Peer/Client upon receiving a message, making each message responsible for “handling itself”. As such, the majority of the system logic is contained within the messages themselves.

Each and every message that is sent between Peers or to and from the Client extends the Message class and contains its own logic. The sender is responsible for filling all the fields of the message with the necessary information and the receiver needs only to call the handle() method. The protocol section will better explain how some messages work.

```
public abstract class Message implements Serializable {

   private String ipAddress;
   private int port;
   private ChordInfo sender;

   public Message(String ipAddress, int port, ChordInfo sender) {
       this.ipAddress = ipAddress;
       this.port = port;
       this.sender = sender;
   }

   public abstract void handle();
```

Figure 1 - Excerpt of the Message Class

Protocols

Backup

For the Backup protocol, a Client requests for a file to be backed up with a certain replication degree by sending a **ClientBackupMessage** to the given Peer. This message, when created, is filled with the file *key*, the byte array with the file content and the desired replication degree. The file *key* is calculated with the same hashing algorithm as the Peer’s keys by passing it a string with the file name and some metadata.

When this message is handled by the Peer, in a new thread, it checks if it’s the one responsible for the *key* of the file or not. Since the Initiator Peer cannot backup any files requested to it, when that happens it saves the file *key* in its **forwarded array**, which keeps all file keys that the Peer was supposed to keep but did not - for whatever reason -, and sends a **Backup** message to its successor (containing the Client info for the return message, and all the other fields received in the **ClientBackupMessage**). When the Initiator Peer isn’t the Peer responsible for the file *key*, it uses a *Lookup* protocol similar to the one Chord uses to find a Peer’s successor, by sending a **FindBackup** (with the same fields as the **ClientBackupMessage**) message to its successor. This *Lookup* protocol uses the Peer’s

Finger Table to “iterate” through the ring in ***O(log N)*** time and finds the predecessor to a given *key* according to the Chord algorithm.

When a backup message is received, the Peer adds the file *key* to the forwarded array if it has no space to keep it, or requests the **FileManager** to write the file/backup, decreasing the replication degree otherwise. After that, the current replication degree is checked. If it is not already achieved and this message has not already been to every existing Peer (to avoid circulating the message around the ring infinitely) it is sent to the current Peer successor. A **BackupReturnMessage** is sent to the Client otherwise. This last message is filled with the current replication degree and, when the Client handles it, it prints a success or error message according to its value and the original one.

```
    public void handle() {
        if (Peer.maxSpace.get() < (Peer.usedSpace.get() + content.length)) {
            //If there is not enough space
            Peer.forwarded.add(key);
        }else{
            //Write to disk
            Peer.fileManager.write(Integer.toString(this.key), this.content);
            Peer.usedSpace.getAndAdd(content.length);
            this.repDegree -= 1;
        }
        //If replication degree not achieved
        if(this.repDegree > 0){
            //If has been to every Peer
            if (getSender().getHashKey() == Peer.chordNode.getNodeHash()) {
                //Return to client with replication degree not reached
                Message message = new BackupReturnMessage(client.getIp(), client.getPort(),
 Peer.chordNode.getNodeInfo(), this.repDegree);
                MessageSender sender = new MessageSender(message);
                sender.send();
                return;
            }
            //Replicate the file forward in the ring
            ChordInfo successor = Peer.chordNode.getSuccessor(0);
            Message message = new Backup(successor.getIp(), successor.getPort(), getSender(),
 this.client, this.key, this.content, this.repDegree);
            MessageSender sender = new MessageSender(message);
            sender.send();
        }else{
            //Return to client
            Message message = new BackupReturnMessage(client.getIp(), client.getPort(),
 Peer.chordNode.getNodeInfo(), this.repDegree);
            MessageSender sender = new MessageSender(message);
            sender.send();
        }
    }
```

Figure 2 - handle() method of the Backup message

Restore

In order to restore a file, a Client creates a **ClientRestoreMessage** object and fills its *key* field with the *key* of the file it wants to restore. This Message is then sent to a peer that launches a new thread to call the handle() message.

This message handler checks if the Peer that was contacted is the one responsible for the requested *key*. If it is not the one responsible for the requested *key,* it searches for the responsible Peer using a *Lookup* protocol similar to the one described in the *Backup* protocol, but with a **FindRestore** message instead. When the responsible Peer is finally reached, the Restore protocol truly begins and a few checks are made in order to ensure that the file is found.

First, if the file exists locally, a request to the **FileManager** class is made in order to create a **RestoreReturnMessage** object that contains the file's contents, it's *key* and a *state* field that is either 1 or -1 (to indicate success or unsuccess finding the file in the system - it will be 1 in this case). This message, after being constructed, is sent directly to the original Client that requested the restore and handled locally in the Client that proceeds to write the file to disk (with the handle() method of the **RestoreReturnMessage**).

On the other hand, if the file does not exist locally, it's *key* is searched for in the Peer's **forwarded array**. If a matching entry is found, the system knows the file is either in the Peer's immediate successor or that Peer's **forwarded array** contains an entry to it. So if an entry is found on the **forwarded array** of a Peer, a **RequestRestore** message is sent to its immediate successor with information regarding the Client that requested the operation and the file *key* solicited. The **RequestRestore** message contains about the same logic as the **ClientRestoreMessage** except for the, in this case unnecessary, *Lookup* protocol.

After localizing the responsible Peer, if a Peer neither contains a file nor information about it in its **forwarded array**, it is safe to say that the solicited file is not anywhere in the system and a **RestoreReturnMessage** is sent with an empty content and a status of -1.

```
    public void handle() {
        //If is the Peer responsible for the key
        if(Peer.chordNode.getPredecessor() != null &&
Utils.isBetween(Peer.chordNode.getPredecessor().getHashKey(),
Peer.chordNode.getNodeHash(),key,false)){

            if(Peer.fileManager.fileExists(Integer.toString(this.key))){
                // Send to client
                Peer.fileManager.readAndRestore(Integer.toString(this.key), getSender());
            }else if(Peer.forwarded.contains(this.key)){
                System.out.println("Sending request request to successor1");
                //else query successor
                ChordInfo successor = Peer.chordNode.getSuccessor(0);
                RequestRestore message = new
RequestRestore(successor.getIp(),successor.getPort(), Peer.chordNode.getNodeInfo(),
getSender(), this.key);
                MessageSender sender = new MessageSender(message);
                sender.send();
            } else {
                //Error message
                RestoreReturnMessage message = new RestoreReturnMessage(getSender().getIp(),
getSender().getPort(), Peer.chordNode.getNodeInfo(), -1, this.key, null);
                MessageSender sender = new MessageSender(message);
                sender.send();
                return;
            }
        } else {
            //Find Responsible
            ChordInfo n1 = Peer.chordNode.closestPrecedingNode(this.key);
            FindRestore message = new FindRestore(n1.getIp(), n1.getPort(), getSender(),
this.key);
            MessageSender sender = new MessageSender(message);
            sender.send();
        }

    }
```

Figure 3 - handle() method of the ClientRestoreMessage object

Reclaim

For a Reclaim operation to occur, the Client must send a **ClientReclaimMessage** to the Peer it wants to free space from. This message is sent with the desired maximum disk space and when the Peer handles it, it sets its maxSpace variable to the received value and deletes previously backed up files until the space used by currently backed up files is lower or equal than the new maximum space.

This operation is done by the **FileManager**, which goes through all the files in the peer folder and for each of them sends a **ReclaimBackupMessage** to the Peer's successor, after the file is read. This message is similar to the **Backup** message in the Backup protocol but doesn't have a replication degree and doesn't answer to the Client. After the message is sent, the file is deleted from the Peer and the file *key* is added to the **forwarded array**, so that when a Peer requests that file it "knows" it will be in one of its successors.

At last the **ReclaimReturnMessage** is sent to the Client with a success message, shown when the message is handled.

```
public void deleteUntilMaxSpace(int space) {
       File folder = new File(rootFolder);
       File[] files = folder.listFiles();
       //Iterate through all files in the local file system
       for (File file : files) {
           System.out.println("Peer: " + Peer.chordNode.getNodeInfo().getPort() +
                               "\nnew max Space: " + space +
                               "\nused space: " + Peer.usedSpace.get());
           //Break if desired usedSpace is reached
           if (space != 0 && space >= Peer.usedSpace.get()) {
               break;
           }

           Peer.usedSpace.getAndAdd((int) -file.length());
           //Backs up the file in the successor and Deletes the local copy
           String deleted = file.getName();
           this.readAndBackup(deleted);

           if (!Peer.forwarded.contains(Integer.parseInt(deleted)))
               Peer.forwarded.add(Integer.parseInt(deleted));
       }
   }
```

Figure 4 - deleteUntilMaxSpace() method called upon a reclaim request

```
  public void readAndBackup(String name ) {
        Path path = Paths.get(rootFolder, name);
        AsyncRead.read(path, new CompletionHandler<Integer,ByteBuffer>(){
            //If reading fails
            @Override
            public void failed(Throwable arg0, ByteBuffer arg1) {
                try { Files.delete(path);
                } catch (Exception e) {
                    System.out.println(e.toString());
                }
                System.out.println("Failed backup, deleted " + name);
            }
            //If reading succeeds
            @Override
            public void completed(Integer arg0, ByteBuffer arg1) {
                ChordInfo successor = Peer.chordNode.getSuccessor(0);
                //Backup file in successor
                Message message = new ReclaimBackupMessage(successor.getIp(),
 successor.getPort(), Peer.chordNode.getNodeInfo(), Integer.parseInt(name), arg1.array());
                MessageSender sender = new MessageSender(message);
                sender.send();
                //Delete the file
                try { Files.delete(path);
                } catch (Exception e) {
                    System.out.println(e.toString());
                }
                System.out.println("Successful backup, deleted " + name);
            }
        });
    }
```

Figure 5 - readAndBackup() method called by the deleteUntilMaxSpace(). This method does the actual maintenance of the replication degree and deletion of the files.

Delete

If a Client wishes to Delete a file from the system it can rely upon the Delete protocol. All a Client must do is create a **ClientDeleteMessage** object with the key to the file it wishes to delete and send it to any active Peer.

Upon receiving a **ClientDeleteMessage**, the Peer will either start the actual delete protocol (if it is responsible for the received key) or initiate a search for the responsible Peer using the same *Lookup* logic in the other protocols.

The responsible Peer will delete the file in its local file system by order of the **DeleteMessage** class handle() method. This class contains information about the Client that requested the deletion and the key to delete. Like the Restore protocol, the Delete protocol will iterate through the ring while it can find either local copies of the file or information in the **forwarded array** about the file to delete, sending **DeleteMessage** requests to the immediate successors of the Peer's it passes through. If a Peer neither contains a file nor information about it in its **forwarded array**, just like the Restore protocol, it is safe to say that the file was completely deleted from the system.

```
    public void handle() {

        if (Peer.fileManager.fileExists(Integer.toString(this.key)) ||
Peer.forwarded.contains(this.key)) {
            //If the file exists locally or its key is in the forwarded array
            Peer.fileManager.delete(Integer.toString(this.key));

            if (Peer.forwarded.contains(this.key))
                Peer.forwarded.remove(Integer.valueOf(this.key));
            //Send Delete Request to successor
            DeleteMessage message = new
DeleteMessage(Peer.chordNode.getSuccessor(0).getIp(),
                    Peer.chordNode.getSuccessor(0).getPort(), getSender(),
this.client, this.key);
            MessageSender sender = new MessageSender(message);
            sender.send();
        } else {
            // Return to client
            DeleteReturnMessage message = new DeleteReturnMessage(client.getIp(),
client.getPort(),
                    Peer.chordNode.getNodeInfo());
            MessageSender sender = new MessageSender(message);
            sender.send();
        }
    }
```

Figure 6 - handle() method of the DeleteMessage object

State

The state protocol is the simplest of all the implemented protocols. Its functionality aims to help understand the current Chord Ring structure and a Peer's important information.

This protocol is composed of only two very simple messages: a **GetStateMessage** and a **ReturnStateMessage.** In order to retrieve a Peer's information a Client must send a **GetStateMessage** to the Peer it wishes to retrieve information from and will eventually receive an instance of the **ReturnStateMessage**. The Peer that receives a **GetStateMessage** will then create a **ReturnStateMessage** and fill it with the following information information:

- Chord information
    - Each finger's information and key
- Predecessor
    - Predecessor's port, IP Address and key
- Successor
    - Predecessor's port, IP Address and key
- Application information
    - Max permitted space
    - Used space
    - Forwarded files

Upon receiving the **ReturnStateMessage** the Client prints all the data in a formatted, easy to read way and ends the protocol.

```
SSL Server Socket created

-------- Client Info --------
{ip='192.168.1.87', port='45331', hash='28'}


-------- Peer 175 State --------
{ip='192.168.1.87', port='3000', hash='175'}

--- Chord Information ---
Finger 0[176]: {ip='192.168.1.87', port='6000', hash='202'}
Finger 1[177]: {ip='192.168.1.87', port='6000', hash='202'}
Finger 2[179]: {ip='192.168.1.87', port='6000', hash='202'}
Finger 3[183]: {ip='192.168.1.87', port='6000', hash='202'}
Finger 4[191]: {ip='192.168.1.87', port='6000', hash='202'}
Finger 5[207]: {ip='192.168.1.87', port='8000', hash='246'}
Finger 6[239]: {ip='192.168.1.87', port='8000', hash='246'}
Finger 7[47]: {ip='192.168.1.87', port='5000', hash='106'}
```

```
Predecessor:{ip='192.168.1.87', port='7001', hash='139'}

Successor 0: {ip='192.168.1.87', port='6000', hash='202'}
Successor 1: {ip='192.168.1.87', port='8000', hash='246'}
Successor 2: {ip='192.168.1.87', port='4000', hash='36'}

--- Application Information ---
Max Space: 270000
Used Space: 188031

Forwarded Files:
        File: 159
        File: 74
```

Figure 7 - Example of a Client call to the State protocol

Concurrency design

Since in operating conditions a Peer needs to handle many operations simultaneously in order to properly function, that system was designed with high concurrency in mind.

One thing that drastically made the design concurrent was the use of a **ScheduledThreadPool** to read incoming messages. Every Peer and Client contains a **MessageReceiver** object that is responsible for reading incoming messages and runs on a separate thread. The **MessageReceiver** has its own **ThreadPool** which it uses to handle the received messages, thus every message read can be run simultaneously preventing any blocking operations.

The Chord algorithm's necessity for periodically running maintenance protocols along with the Application itself sending quite a few messages for every protocol made it mandatory to implement such a system to handle messages. The Stabilize and Fix Fingers operations are scheduled, in each Peer, to run at a fixed rate of 500 milliseconds using the same **TheadPool** as the one used to schedule the **MessageReceiver** object.

With such a concurrent design came the necessity to maintain the access to some common variables as an atomic operation. To achieve this a few **Thread-safe** containers were used like ***syncronizedLists*** in the **forwarded array,** the Peer's **Finger Table** and the Peer's **Successor List** and ***AtomicIntegers*** in the **maxSpace** and **usedSpace** variables.

Another way to further extend the concurrency design is to make use of **Non-blocking I/O**. For this, the system utilizes **Java NIO** for reading/writing/deleting files. This made possible to, for example, proceed with the rest of the Reclaim protocol while deleting a copy of a file.

```java
    public static FixFingers fixFingers;
    public static Stabilize stabilize;
    public static MessageReceiver receiver;

ScheduledExecutorService executor = Executors.newScheduledThreadPool(3);
```

```
        executor.schedule(receiver, 0, TimeUnit.SECONDS);
        executor.scheduleAtFixedRate(Peer.fixFingers, 0, 500,
TimeUnit.MILLISECONDS);
        executor.scheduleAtFixedRate(Peer.stabilize, 0, 500, TimeUnit.MILLISECONDS);
```

Figure 8 - Modified excerpt of the Peer class to demonstrate the use of the ScheduledThreadPool

JSSE

To ensure secure and reliable connections between all the Peers and in Client/Peer communications, the system uses Java's **SSLServerSockets** with client authentication **enabled**. This way, every entity (being it a Peer or a Client) needs to have the correct keys and truststore.

```
    private void createSocket() {
        SSLServerSocketFactory serverSocketFactory = (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();

        try {
            serverSocket = (SSLServerSocket)
serverSocketFactory.createServerSocket(this.port);
            this.port = serverSocket.getLocalPort();
            serverSocket.setNeedClientAuth(true);
            serverSocket.setEnabledProtocols(serverSocket.getSupportedProtocols());
            System.out.println("SSL Server Socket created");
        } catch (IOException e) {
            System.err.println("Error creating SSL Server Socket");
            e.printStackTrace();
        }
    }
```

Figure 9 - createSocket() method that initiates the SSLServerSocket field of the MessageReceiver class.

Scalability

It is known that a good concurrency design goes a long way in the scalability of each Peer execution, and the usage of **Thread-Pools** and **Java NIO** in this project helped a lot in this particular aspect. Aside from this, and for the overall system scalability, the **Chord Protocol** is used for managing all Peers that belong to the system, keeping them updated relative to the others, and also to help each one have a better time executing so many threads at the same time.

Thanks to Chord, when one Peer wants to communicate with another, both the **Finger Table** and the **Successors List** are consulted, ensuring that the search of the needed Peer is

O(log N) in time, which means that one message doesn't need to go through all nodes of the ring, if a Peer would want to communicate with a node that is far away from itself. This means that a lot of message sending is saved and fewer Peers need to receive, process and eventually send the same message, repeating these steps while the Peer isn't found.

Still related to the Chord algorithm, a reason why it is useful is that it is efficient when used in a high update rate ring, which means that Chord tries to fix issues with peers that are constantly joining and leaving the node's ring. As the **Stabilize** and **FixFingers** protocols need to be frequently run to keep every Peer updated, it almost makes Chord a problem-proof managing system, but if the joining-leave rate is relatively high and certain data needs to be backed up/replicated through several nodes, there might be times where the replication can't be kept because the **Stabilize** wasn't able to update it in time, and keeping a thread waiting for it to be updated is not very scalable.

One way in which the Chord was improved to handle this kind of problem is to have each Peer save a list of its direct successors, a **Successors List**. In this implementation, this list is updated in every run of the Stabilize algorithm, in order to keep it as true as it can be to the specific ring the Peer is included in. In the real world, this helps the execution of the protocols regarding the backup system, because if the successor happens to become unavailable in between two executions of Stabilize, probabilistically, there will always be at least one successor active in the Successors List.

Another aspect that helps the Peer-wise scalability is the fact that the quantity of information that each node stores in volatile memory is low and it doesn't depend on the size of the ring. The only variable that might eventually grow on memory usage is the **forwarded array**. This array could happen to be bigger when, for instance, one Peer has its internal storage full and the next files to be backed up can't be stored in such a Peer, so it only keeps its file key (one integer variable) for the sake of knowing that the successor might have it. Generally, this array will never be very big, considering also that its maximum size is the maximum number of nodes in the Peer's ring.

Fault-tolerance

The advantages the **Successors List** has related to the scalability are even more important and crucial when talking about the fault-tolerance capabilities of the application. Every time a certain Peer tries to access its successor, if it can't establish a connection, he removes it from the list and supposes that its next successor is the following entry in the Successors List. This process is repeated every time the successor needs to be contacted, and theoretically, it is almost impossible that all nodes in this list become unavailable between **Stabilize** protocol executions.

Another way to ensure the authenticity of the data related to the backup of the files is to keep all of the information as updated as the Chord allows it to be. Two protocols were implemented to keep the system fault tolerant:

- When a new Peer joins the ring, after the acknowledgement it sends to its successor so the later knows that a new predecessor appeared, it receives from the same successor a list of files that must be kept in the forwarded array. This means that the Peer has information about every file previously backed up that he would have been requested to store. This protocol makes the system simple, because the peers don't need to worry about keeping the desired replication degree and dealing too much with sending and receiving files, which also helps with the scalability of the system;

- If a Peer is voluntarily leaving the ring, it will invoke the reclaim protocol for the files it has stored, and the successor Peers will act as if it was a normal reclaim (in fact, it is not much different than a 0 bytes reclaim on a Peer).

On the application level, and somewhat inferred at this point, is the replication degree. This makes it so that if a Peer happens to leave the ring unexpectedly, its files are not lost forever and instead are available on some other Peer. The higher the replication degree, the higher the degree of fault-tolerance (but also the overall space occupied in the ring).

When a tentative to send a message in the Stabilize and Notify protocols is unsuccessful the sending Peer retries to send the same message to its next successor making the system capable of handling unexpected communication errors and other possible faults related to the active state of a Peer.

```
MessageSender sender = new MessageSender(message);
if (!sender.send()) {
     Peer.chordNode.setPredecessor(predecessor);
}
```

Figure 11 - Example of code related to the failure to communicate with some peer