Project 2 - Distributed Backup Service in P2P Network

Final Report

Mestrado Integrado em
Engenharia Informática e Computação

Sistemas Distribuidos

Turma 6 Grupo 02

Ângela Cardoso	Diogo Pereira	Gonçalo Lopes	Ivo Fernandes
up200204375	up201305602	up201303198	up201303199

May 30, 2016

Contents

1 Introduction

The objective of this project is to develop a distributed backup service for a *wide area network* **(WAN)** based on the *peer-to-peer* **(P2P)** application architecture. The idea is to make use of the free disk space of computers for backing up files from system users. The service is provided by peers who are part of a chord network and can be located in different *local area networks* **(LANs)** as long as the routers that connect those LANs to the outside allow *universal plug and play* **(UPnP)**.

In this report we will explain in detail the approach used to solve the problems related to the implementation of the project.

2 Distributed Hash Table - The chord

2.1 peer lookup

One of the most important issues that needs to be tackled when implementing a P2P application is managing and locating the peers. To achieve this, we follow the chord *Distributed Hash Table* **(DHT)**.

We will not enter into the specific implementation of the chord on this report, since it is described in [**?**]. The pseudo-code on this paper was used for our code implementation.

One of the most important characteristics of the chord is that to locate a peer with key `K`, on a chord with a maximum of `N` peers, a maximum of `log(N)` hops is required. For a large `N`, this drastically increases lookup performance.

A peer on the chord also keeps a pointer to its predecessor, and a list of pointers to its successors. We used these pointers to implement replication, as we discuss ahead.

2.2 peer join

In order to join the network, a peer needs to know an entry-point, the IP address of a peer that already belongs to the network.

When starting a peer, the user can either specify a key, or receive a random one. When specifying a key, if that key is already filled in by a peer on the chord, the user receives an error, saying the key already exists. If the user does not specify a key, the program will first connect to the entry-point and ask the max number of peers allowed on the network. Then, it generates an array of `N` integers, and shuffles the array. After that, the peer tries to connect to the network with one key at a time. If all keys are full, the program outputs a message saying the network is full. The array is shuffled so that peers are evenly distributed on the chord.

After joining, the peer uses `chordNotify()` (see [**?**]), notifying its successor that it is its new predecessor. The `stabilize()`, `fixFingers()` and `checkPredecessor()` functions (also specified in [**?**]) are periodically executed to fix possible inconsistencies with the fingerTable, successors and predecessor, caused by peer departures and failures. The function `chordNotify()` also returns the predecessor's successors, so that the peer joining the chord may update its list faster.

3 Replication

Before getting into the backup, restore and deletion sections, we will briefly explain how we tackled Replication, entering into details for each protocol on their specific sections.

Similarly to the first project, a file has a `file_id` and it is split into chunks. To know where a chunk is or is supposed to be located at, the chunk's corresponding peer key `K` needs to be cal-

culated. `K = (file_id + chunk_no) % N`. This ensures that chunks are evenly distributed across the network.

Locating chunks becomes an issue when peers do not have enough space. To prevent chunks from being unreachable, a peer stores information about a chunk being saved or not, so that it can forward the request about the chunk to its successor.

To replicate a chunk with replication degree `R`, the chunk holder makes a recursive call to its successor, asking it to store the chunk with replication degree `R - 1`. This process is repeated until the replication degree is either met, or there aren't enough peers (or enough peers with available space) to satisfy the replication degree.

To keep consistency, when a peer joins the network, it must copy the chunks of another peer. For example, suppose that a chord consists of peers 1 and 5 and that peer 3 enters the chord. Then peer 3 must receive from peer 5 all the chunks whose key is between 3 and 5. Hence, peer 3 makes a `releaseChunk(Chunk)` call, which recursively calls successors until it finds a peer that either:

- Doesn't know the chunk, and the call ends;
- Has the chunk, and the chunk's saved `rep_degree` is 1, which means that this is the last peer to have the chunk, and that the desired replication degree is met. This peer then deletes the chunk, because the chunk's current replication degree is larger than the desired replication degree. If a peer did release the chunk, when the recursive call travels back through the successor chain, all peers decrement their chunk's replication degree, to update their position on the chain.

When a peer receives a new successor, it also sends to it the chunks whose replication degree are higher than 1, because they should belong to the new successor. This corrects any error that might have occurred when the old successor sent its chunks to the new successor (because it is its predecessor).

Both of these actions utilize `putChunk()` instead of `storeChunk()`. The difference is that `putChunk()` retrieves the chunk's body from the network if it does not receive it. This covers the case when a peer joins the network between two peers that have no space for chunks, which means these peers will send the chunks without body, using the `putChunk()` call.

What if two peers join at the same time? Both `chordNotify()` and `setSuccessor()` are synchronized methods, so that the successor or predecessor receiving the chunks does not suddenly change. If the methods were not synchronized, this would mean that for two peers joining the network at the same time, connecting to the same peer, none of them would receive all the chunks, but only a part of them.

3.1 Failure

When a peer's successor or predecessor fails, chunk replication is maintained when the peer receives a new successor or predecessor.

If, for some reason, the chunks replication degree goes below the desired degree, and this isn't detected, perhaps because one of the chunks in the successor chain crashed while the recursive call after a peer departure is active, the `checkChunkReplication()` function, which is periodically executed, performs the same actions that are taken when a node joins or leaves. However, the actions are only called for chunks by the peers that have the chunks' bodies. Also, the chunks' `last\_check` must have occurred longer than the *check_delay*. The `last\_check` value is used to prevent a chunk from getting checked by multiple peers at the same time, which would unnecessarily flood the network.

The function `checkChunkReplication()` also releases chunks when they are above the replication degree.

3.2 Unreachable chunks

Seeing how the chord works, there's an issue with unreachable chunks. If a peer has a chunk that is unreachable, it should delete it.

How to detect a chunk is unreachable?

Periodically, a peer calls `checkUnreachableChunks()`.

1. If the chunk's desired replication degree is 1, it does not perform any check, and does not delete the chunk.
2. Calculates, the first peer has the chunk, the `Chunk Holder`.
    - If the `Chunk Holder` does not exist, this peer failed to receive the delete message for the chunk. The peer then deletes the chunk.
    - If the `Chunk Holder` exists and is not the peer checking the chunk, the peer asks its predecessor if it knows the chunk. If it doesn't, the peer deletes the chunk.
    - If the `Chunk Holder` exists, and is the peer checking the chunk, the chunk is not deleted.

If a peer deletes a chunk when it shouldn't have, maybe because its predecessor hadn't completely received its chunks, and did not yet know of the chunk, `checkChunkReplication()` will later fix the issue.

What if the peer just revived, and is the Chunk Holder but, the chunk had been deleted from the network? We fix this by having a different `checkUnreachableRevive()` method. This method is executed 2 times. The `checkChunkReplication()` and `checkUnreachableChunks()` methods only start being periodically executed after `checkUnreachableRevive()` is executed twice. If this does not happen, the peer will wrongly reintroduce an already deleted chunk to the network. Besides only starting the periodic checks after the revive check, the peer only sends chunks to successors or predecessors on `setSuccessor()` and `chordNotify()` if the peer has revived already.

3.3 The recursive calls

An important issue with the recursive calls is that, if the calls lap around the chord and go into an infinite loop. We fixed this issue by adding an `ArrayList<Integer>` to the recursive calls. The first action when executing one of these calls, is checking if the peer's `guid` is already on the received list. If it isn't, the peer adds its `guid` to the list, and makes the call to the successor. This prevents calls from looping.

What if peers crash or join the network while the call is active, and this prevents the call from reaching all successors? This is fixed by using the periodical `checkChunkReplication()` and `checkUnreachableChunks()` calls.

3.4 The network load

Since the chord needs to keep calling certain methods to keep balanced, and some of these methods have to send chunks through the network, the load might be too high. If there is knowledge about how often peers are expected to join/leave the network, and how high replication degrees are, the time intervals to call the "balancing methods" may increase or decrease, keeping consistency and changing network load accordingly.

4 File Backup

To backup a chunk, the chunk's corresponding key must be calculated, as explained in the Replication section. After that, the peer calling the backup for the chunk locates the peer corresponding to the chunk's key and asks that peer to store the chunk with a desired replication degree.

When a peer is asked to store a chunk, it checks if it has enough space. If there isn't enough space to store the chunk, the peer keeps information about the chunk, as well as setting the `status` value to 0 meaning it didn't store the chunk. The peer then forwards the request to its successor, keeping the chunk's `rep\_degree`.

If the peer has enough space to store the chunk, it will decrement the *rep_degree* and forward the request to its successor.

The backup ends when either:

- The `rep\_degree` value reaches 1 and is stored on a peer
- The first peer to ask for the chunk's backup receives the backup request for said chunk from its predecessor. This means there weren't enough peers on the network to store the chunk with the desired replication degree. The backup call must end, otherwise it becomes an infinite loop.

5 File Restore

To restore a chunk, after calculating the chunk's corresponding key and the peer corresponding to that key, a restore call is made.

- If the peer does not have any information on the chunk, the call stops. This prevents infinite loops.
- If the peer has information about the chunk, but does not have its body, it forwards the request to the successor.
- If the peer has the chunk's body, it returns its `peerInterface` through the recursive call. This is done so that the chunk does not have to be transferred through all the peers. After receiving the chunk holder's `peerInterface`, the client asks it for the chunk.

6 File Deletion

To delete a chunk, after calculating the chunk's corresponding key and the peer corresponding to that key, a delete call is made.

- If the peer has any information about the chunk, it deletes it and forwards the request.
- If the peer does not have any information about the chunk, it does not forward the request.

7 Security

7.1 User authentication

To ensure that only the user that inserted the file into the network can retrieve it we implemented a username-password authentication system, an usage example would be:

1. User registers into the network with username and password, his encrypted metadata is stored in the network with replication degree 3.
2. User backs up example.txt file
3. User asks to restore of example.txt file
4. User metadata is retrieved from network and decrypted using a key generated from username and password, it is verified if the file is registered to the user and if it is the restore process starts.

In the client metadata we store a hashmap that has the file name as the key and file metadata as the value.

The file metadata consists of the file identifier that is generated when the file is backed up, as well as the number of chunks that the file was split into.

By storing this information we ensure that only the user that backed up the file can retrieve it, the possible attacks are:

- trying to guess the username and password combo using brute force to access a specific user's files.
- reverse engineer the the communication between peer and Client to obtain file identifiers and then simulate the communication between peer and client.

Even if the attacker had the skills to reverse engineer the communication between the Client and the peer the file received would be encrypted and the decryption key is generated from the username and password of the user that backed up the file, turning all that data useless. The most damage that could be done would be for the attacker, after being able to manipulate communication, to delete files as only the file identifier is necessary.

7.2 Confidentiality

When a file is being backed up into the network all its chunks are encrypted using a key generated from the username and password of the user, this ensures that people with access to the file system of various peers would not be able to recreate any files without knowing the username and password of the user that backed them up.

8 Port Mapping

So participants in the system can communicate with each other when they are part of different LANs we use the library Cling which provides an API for UPnP services, the system creates a port mapping on all NAT routers on a network, the ports mapped are 1099 and 1100 that are used in RMI communication. For this method to work the network routers must permit UPnP. We successfully managed to implement UPnP port forwarding and are able to communicate with peers in different LANs and clients accessing through different LANs.

The only problem we could not get through was multiple peers in the same computer communicating with different LANs. To test WAN operability of our application it is necessary to only run one peer per computer.

We added a constant *LAN_MODE* that removes all the port forwarding and setting of the rmi server hostname system property, we submitted with code with this constant to true so that we are able to run the project in FEUP, to test in a WAN this constant must be false.

9 User Manual

So that you can be able to run our *Application*, you have to complete the following steps, depending on your Operating System (**Windows** or **Linux**).

First of all it is necessary to put all the *Cling* (*API* used to do *Port Mapping* and to implement *UPnP*) *jars* in the same folder of *src* packages (*test*, *client*, *peer* and *utils*). You can find it through our svn repository (`https://redmine.fe.up.pt/projects/sdis1516-t6g02-p2/documents`).

Next, depending on your OS, open the *command terminal* and navigate to *src* folder, where there are present all the Application packages and *jars*.

- **Windows**

  Because of our communication protocol being done in *RMI (Remote Method Invocation)*, firstly you have to start *rmiregistry*. This can be made like this:

  ```
  start rmiregistry
  ```

  – Then, to compile, run the following commands:

  ```
  dir /s /B *.java > sources.txt
  ```

  ```
  javac -cp ".;*" @sources.txt
  ```

  – Finally, to launch the Application, run the following command:

  ```
  java -cp ".;*" client.ClientStart
  ```

- **Linux**

  – To start *rmiregistry*:

  ```
  rmiregistry &
  ```

  – To compile:

  ```
  javac -cp ".:*" */*.java
  ```

  – To launch the Application:

  ```
  java -cp ".:*" client.ClientStart
  ```

From now onwards, everything is equal independently of your OS, except the `".:*"` token after *"java[c] -cp"*. If your OS is **Windows** use `".;*"`, else if is **Linux** use `".:*"`.

So upon launching the Application, a message *"Dear User, please enter the commands:"* will appear.
These commands could be:

- `<sub_protocol> <opnd_1> (<opnd\_2>)? (<access_point>:<access_point_key>)?`

  Where:
  [**sub_protocol**] is either *BACKUP*, *RESTORE* or *DELETE*.
  [**opnd_1**] is the filename to *BACKUP*, *RESTORE* or *DELETE*.
  [**opnd_2**] is the file's replication degree if the subprotocol chosen is *BACKUP*.

[**access_point**] is the internal host *IP address*.
[**access_point_key**] is the internal host *guid*.

- `<action> <username> <password> <access_point>:<access_point_key>`

  Where:
  [**action**] is REGISTER or LOGIN.
  [**username**] is the client's (files' owner) username.
  [**password**] is the client's password.
  [**access_point**] is the internal host *IP address*.
  [**access_point_key**] is the host *guid*.

But, before testing any *subprotocol* available (Backup, Restore or Delete), i.e, launching any subprotocol command-related referred above, you have to **create** a *chord* with many **peers** in order to test the *Protocols* implemented, like *Backup*, *Restore* or even *Delete* files.
Note: Each peer joining or creation must be done in separated *command terminals*.

To **create**, just type:

```
java -cp ".;*" peer.peerStart [name] create [max_peers] [space]
```

Where:

[**name**] is the peer's name and its folder name, where it will store files.
[**max_peers**] is the number maximum of peers of that chord. This number must be multiple of 2.
[**space**] is the peer's space available to store files.

If you would like to **join** a **peer** to a specific *chord*, you have two ways to accomplish that:

- *Join known chord and receive a random key*

```
java -cp ".;*" peer.peerStart [name] join [known_host] [known_host_key]
[space]
```

or

- *Join known chord with given key*

```
java -cp ".;*" peer.peerStart [name] join [known_host] [known_host_key] [key]
[space]
```

Where:

[**name**] is the peer's name and its folder name, where it will store files.
[**known_host**] is the *chord* access point.
[**known_host_key**] is the *chord* access point *guid (Global Unique Identifier)*, which is the remote object's name.
[**given_key**] is the *guid* to join (the *chord*) with.
[**space**] is the peer's space available to store files.

10 Effort Distribution

Ângela Cardoso Chord network and protocol implementation

Diogo Pereira Client side development

Gonçalo Lopes Chord network and protocol implementation

Ivo Fernandes Protocol implementation, port forwarding