Self-hosted cloud platform using decentralized downloads

Alberto Fernández
Director: Jordi Delgado
Specialization: Information Technologies

25 Octubre 2017

Abstract

In this project we develop an open source solution that can be deployed by anyone, and offers its administrator the ability to upload files and share them in a more distributed and efficient way. The multimedia files can also be streamed in the platform rather than being downloaded.

The common client-server paradigm has many flaws when it comes to efficiency of resources. The server is always sending the complete information to any user requesting it, ending up sending the same over and over again, consuming a huge bandwith, and limiting the users bandwith if the server is overloaded. When it comes to downloading large files instead of plain information, this problem is even more relevant.

To solve this, we aim to decentralize the downloads of files, using a peer to peer system, but also accessible and easy to use by making it compatible with the browser. The project has a strong open source focus, it delivers a free and open platform usable by anyone, and offers a technological solution that can inspire other software developers or companies that want to integrate a peer to peer downloading system in their solutions.

This project can help individual users and organizations reduce their use of resources and increase scalability, making an important impact on the costs of infrastructure, and delivering a better service to the users.

Resumen

En este proyecto se desarrolla una solución open source que puede ser desplegada por cualquier persona, y ofrece a su administrador la habilidad de subir ficheros y compartirlos de una forma mas distribuida y eficiente. Los archivos multimedia tambien pueden ser reproducidos en la propia plataforma sin necesidad de descargarlos completamente.

El común paradigma cliente servidor tiene algunos problemas en lo referente a eficiencia de recursos. El servidor esta siempre enviando cada información completa a cualquier usuario pidiéndola, y acaba enviando la misma información una y otra vez, consumiendo una gran cantidad de ancho de banda, y limitando a los usuarios si este ancho de banda del servidor esta saturado. Cuando se trata de descargar ficheros grandes, el problema se vuelve aún mas relevante.

Para solucionar esto, se tiene como objetivo descentralizar la descarga de ficheros, usando sistemas distribuidos peer to peer, pero manteniendolo accesible y facil de utilizar haciendolo compatible con los navegadores de internet. El proyecto tiene un fuerte enfoque de código abierto, ofrece una plataforma libre y usable por cualquiera, y también ofrece una solución tecnológica que puede inspirar a otros desarrolladores de software y compañias que quieren integrar un sistema peer to peer para descargar ficheros en sus soluciones.

Este proyecto puede ayudar a individuos y organizaciones a reducir su uso de recursos e incrementar la escalabilidad, haciendo un importante impacte en el coste de infrastructura, a la vez que ofreciendo un mejor servicio a los usuarios.

Resum

En aquest projecte es desenvolupa una solució open source que pot ser desplegada per qualsevol persona, i ofereix el seu administrador l'habilitat de pujar fitxers i compartir-los de forma més distribuida i eficient. Els fitxers multimedia també poden ser reproduits a la mateixa plataforma sense necessitat de descarregar-los completament.

El comú paradigma client servidor té alguns problemes en quant a eficiencia de recursos. El servidor està sempre enviant cada informació completa a qualsevol usuari que la demani, i acaba enviant la mateixa informació una i un altre vegada, consumint una gran quantitat d'ampli de banda, i limitant als usuarios si aquest ample de banda del servidor està saturat. Quan es tracta de descarregar grans fitxers, el problema es encara més rellevant.

Per solucionar això, es té com a objectiu descentralitzar la descàrrega de fitxers, utilitzant sistemes distribuits peer to peer, però mantenint-ho accessible i fàcil d'utilitzar fent-ho compatible amb amb els navegadors d'internet. El projecte té un fort enfocament de codi lliure, oferint una plataforma lliure i usable per qualsevol, i també ofereix una solució tecnològica que pot inspirar a altres desenvolupadors i companyies que vulguin integrar un sistema peer to peer per descarregar fitxers a les seves solucions.

Aquest projecte pot ajudar a individus i organitzacions a reduïr el seu consum de recursos i incrementar la escalabilitat, fent un important impacte en el cost de l'infrastructura, a la vegada que oferint un millor servei als seus usuaris.

Contents

List of Figures

List of Tables

List of source codes

Chapter 1

Introduction

Peer to peer networks have been around since the creation of Internet itself. The vision itself of the internet was thought in its early days as a peer to peer network, with equally privileged nodes sharing content with other nodes and contributing to the network. But quickly companies take control over the nodes and the network, and this vision, if ever implemented, ceased to be the model of internet.

Later on, decentralized and peer to peer networks gain popularity again, but not without controversy. It was in 1999, when Napster appeared and became popular. It was used to share media files like books or movies, most times copyrioghted material, creating a legal controversy that remains nowadays.

Putting aside these controversy, peer to peer networks still have a fundamental potential in delivering content to a lot of people. By empowering the users to use their bandwith to offer part of the content, a company can save big amounts of money in infrastructure. Some companies have tried this over the years, like Blizzard [1] with its game World of Warcraft [2][1], the downloader of the game had the option to enable peer to peer downloading.

But it wasn't enough to make it a popular option. Using peer to peer required a specific software to run, and could not be executed through the browser, which made that most of the people wasn't aware of the technology or its use.

This has the opportunity to change since WebRTC [4] came to life. It allowed peer to peer connections directly from the browser, which allows for all sort of peer to peer networks and algorithms on our daily life webpages. Big new companies like Netflix [5] are considering peer to peer options to distribute their content [6].

Moreover, other decentralized technologies are rising this past years, with a special focus on cryptocurrencies [7]. This shows a constant shift of paradigms from a centralized internet, with predomination of client-server protocols, to peer to peer protocols, that make use of the resources of each node, and make each node user and contributor at the same time.

This project will try to mix both of the worlds, offering a technology and a platform for decentralized downloads, while keeping privacy and access control on one entity.

[1] The peer to peer option was dropped around 2015 [3]

Chapter 2

Preparation of the project

This chapter describes the previous steps to start the development of the project in hand. First of all, the problem we are going to tackle is described in detail, with the necessary technical information needed to understand the problem.

After describing the problem, it is offered a first glance of what a solution could look like. Along with this possible solution, there are described the different actors that would be involved in a project like this, and the current state of the art of the different technologies involved in the solution. If there are any similar solutions, they will also be mentioned with a small description of what they do and how they relate or are similar to this project.

After analyzing the problem and coming up with a solution, we are going to plan how to design and develop that solution. This includes a temporal planning that will help achieve the results we want in the timeline we have, and an economical planning that would allow us understand the economical implications of the realization of the project.

There is also included a sustainability analysis to know the impact of this project on others, economically, socially and on the environment.

2.1 Formulation of the problem

The problem we are trying to tackle is the download inefficiency in file storage and sharing web applications. Usually, these applications come with http downloads in the classical client-server paradigm. That is fine if only one person will download the file, but becomes a problem if you want to share your files with other people.

On the other hand, peer-to-peer downloads have been around since the beginning of the century, but they could not be used with web technologies. Because of its decentralized nature, they always required a specific desktop application.

In this section we will cover both of the paradigms, analyzing their problems in more detail.

2.1.1 Client-server

In the client-server paradigm, there are two main agents involved in the exchange of information (or 'files', that will be the main units of information to download in the platform). These are the **client** and the **server**. The client starts the communication, sending a request for a specific file, and the server sends back the file to the client. Examples of protocols that follow a client-server paradigm are: **HTTP** and **FTP**. We will focus on HTTP in this project.

The main problem of this paradigm is that the server has to send the entire file to each of the clients requesting it, which makes it inefficient for many reasons:

- It heavily affects the network of the server, specially when serving large files to a large number of clients.
- The hard drive can become a bottleneck when many clients are requesting files, slowing down the whole server.
- As with all centralised applications, it is a single point of failure. If the server goes down, there is no other way to retrieve the file.
- Because the file will be stored in a hard drive, it is costly to scale for availability and redundancy. Some RAID solutions for hard-drives can be an alternative, or having backup servers. But it comes at a big cost.

But there are some important reasons why we cannot just drop HTTP from the table:

- It is the only protocol the main browsers use to access webpages. If we want our web platform to be accessible to a large audience, it has to be through HTTP (Or its secure version HTTPS).
- Because of its centralised nature, routing and locating the file and the server is more efficient, as we only have to go through a DNS table.

2.1.2 Peer-to-peer

When we talk about peer-to-peer, we are referring to the protocols where all the connected computers to the network can act as client and server at the same time. These computers are called 'peers'.

The most used peer-to-peer protocol nowadays is Bittorrent. This project will focus on it because it has the major adoption between peer-to-peer protocols. We will explain in more detail how Bittorrent works, but for now let's focus on the benefits and disadvantages it has. Although we will focus on Bittorrent, these benefits and disadvantages apply almost identically to other peer-to-peer protocols.

It's important to note that some properties of Bittorrent can be benefits in a healthy network (many peers, good bandwith) and be a disadvantage at the same time with a poor network (few or zero peers, poor bandwith), as a consequence some properties will appear in both lists.

These are the benefits, assuming the nework is healthy:

- Files can be served from any computer in the network, which removes the single point of failure that centralization involves.

- Download speed: The maximum download speed that can be achieved will only depend on the number of peers and its bandwith capacity in the network. When we are in a healthy network, it usually translates in reaching your maximum download capacity of your bandwith.
- Server cost: Because the distribution of files relies on peers that previously downloaded the file, your server load is minimum, leading to a big decrease in cost of infrastructure. Your server will only be needed when no peer has the file, but this is optional. No file server is needed for the network to work.
- Piece splitting: BitTorrent splits files into different pieces, and then each piece is retrieved from the same or different peers. This makes it possible to download the file from different peers at the same time, and allows for fast switch of peers if one is found unreliable.

And these are the disadvantages:

- Bittorrent needs a desktop application in order to work. This is because it uses TCP [8] and UDP [9] connection protocols to connect the peers, and these protocols cannot be accessed directly through a browser.
- Download speed: Like its benefits when there is a healthy network, it can also become a downfall when the network is unhealthy. The download speed can be very slow or zero in the worst case scenario, when there are no peers or only low quality peers.
- Piece splitting: In an unhealthy network, it could happen that no peer has a specific piece of the file requested, which makes impossible the download of the file until a peer connects to the network with the missing pieces.

2.2 Finding a solution

From what we have seen so far, HTTP seems the best protocol to access our platform. It will make it accessible to anyone with a web browser. On the other hand, for the specific functionality of sharing files with others, peer-to-peer protocols are the best option.

This is actually possible through WebRTC, a new web standard technology. WebRTC [4][10] its the acronym for Web Real Time Communication, technology developed by the W3C [11] (World Wide Web Consortium) that creates a peer-to-peer connection between two clients and can be used in the browser. The standard was initially launched in May 2011, and slowly became adopted by the different browsers. This opens up the possibility for a web platform to offer peer-to-peer downloads.

2.2.1 Platform proposal

The proposed solution is to create a self-hosted platform that would allow the administrator of the platform to share their files, using a peer-to-peer protocol for the downloads using WebRTC. This way, the downloads would be more efficient, and the server would have less load when different people is accessing at the same file. This platform would also ensure the complete availability of the file by offering the first peer with the complete file on the network. With this, we would have solved the main problems of HTTP and Bittorrent formulated in the previous section.

The platform would also have an intuitive interface, with information about the people downloading a file at a certain time, download speed and number of conexions.

It would also have an admin panel. From this panel, the admin could upload and edit files in the platform, and also invite people to register on the platform in order to download the files.

Theoretically, it could even be possible to stream the files that are media files supported by the new video and audio HTML5 tags, so the user do not need to download some files to their desktop in order to visualize it.

2.3 Actors involved

This project will not serve the necessity of any direct client, but instead will be offered open source, so anyone can adapt, use and deploy it in a server.

2.3.1 Target

The target of this project could be from companies to individual users, that could need a platform to privately share some files with a specfic audience.

Some examples of situations where this project could be used:

- Multimedia distribution: It could be used to share multimedia files like videos and photos. From sharing them with family or friends, to bigger communities that would find useful to have a specific platform where to store and share files.
- Document management: This project could be used by companies or departments to safely store their files, and invite only the authorized people to access them.

Although the platform might not be suitable for the following situations, the specific technology (peer-to-peer download integration into web pages) could be adapted to other kind of companies that would see a huge benefit in using it:

- Games downloads: Videogames are very heavy products, usually around the tens of gigabytes, so having a desentralised way to deliver them would cause a massive cost reduction in infrastructure.
- Multimedia distribution and streaming: Like mentioned before, but in this case oriented to commercial companies like Youtube [12] or NetFlix [5] that could see a huge cost reduction in infrastructure by using this technology to distribute their videos.

2.3.2 Who will use it and who will benefit from it

There are some different ways to use this project, and who will use it will in part depend on it:

- As it is: This project will have a basic user management system that will enable the administrator to invite users to register in the platform. Once registered, users will be able to see and download files that the administrator uploaded to the platform. For those where this system is enough, it can be used by having some system administration knowledge in order to self-host the platform. In this case, the administrator would benefit from the platform by saving costs in infrastructure, while the users invited by this administrator could benefit too from better download speeds.

- Building something on top: Being an open source project, anyone could use it as inspiration for another project, or use the project itself to build on top of it. This would be the case of companies with very specific business logic, where the default project would not be suitable. Programming knowledge in web technologies will be needed. in this case, those who adopt it will be who benefit from it, and their users who will use the technology.

2.4 State of the art

Although some important concepts have already been explained in previous sections, this section will have a more in depth description of the technologies that are available to develop this project. Also another technologies will be explained that might be used to deploy this project to a server.

2.4.1 HTTP

HTTP is the defacto protocol to download webpages and files from the internet. It works on top of TCP as transport layer. The current version is 1.1, from 1999. Version 2 [13] is being developed by a team of W3C, including in its specification other types of protocols like WebSocket [14].

As mentioned before, HTTP uses a client-server model paradigm, and no peer-to-peer options are contemplated at the moment. For each resource requested from the server, one TCP connection is open.

In order to specify the resource that is being requested, an HTTP request contains a path, specifying where to find it. Like file systems, it contains different parts divided by a slash (/). This allows for a better organization of HTTP resources. For instance, to access the friends of a user in HTTP, a possible path might look like this: */users/user12/friends*. In this case, we first go to *users*, we look for the *user12*, and then we retrieve its friends.

Also, the request contains some "Headers" that give more information about what and how should the request be resolved. For instance, the header 'Method' specifies what do we want to do with the resource, as we can get, modify or delete it. A request can also contain a body that contains the new information if we are modifying a resource or creating a new one.

For the responses, they have a status code that specifies if the request succeeded or failed, and contain also headers and body if needed. In responses with content, the content will come in the 'body' section.

The table 2.1 contains the most common HTTP methods.

Status codes are 3-digit numbers that specify if the server could answer the response or not and how. They are divided into 5 big categories, each one starting by a different first digit. E.g 2XX status code means the response could be fulfilled, while 4XX status code means an error. Table 2.2 contains a list of commonly used status codes. Other non-official status codes can be used, and their meaning is agreed by convention.

2.4.2 Bittorrent

BitTorrent is the most used peer-to-peer protocol at the moment to share files in a decentralized way.The first implementation of the protocol was developed by the programmer Bram Cohen [15].

Method	Description
GET	Only retrieves data and has no effect on the data
POST	Used to send information to the server, like uploading files or sending forms
HEAD	Like GET, but only transfers status and headers, without content
PUT	Stores the information sent under the URI specified. If the resource already exists, it will be overwritten, otherwise a new one will be created
DELETE	Remove the representation of the target resource
OPTIONS	Describe the communication options for the specified resource

Table 2.1: HTTP Methods

Status code	Description
100 Continue	Means that the request should proceed. E.g when used with POST request, the client could wait for a 100 response before sending the body of the request.
101 Switch protocols	The client has requested to switch protocols and the server has agreed to do so.
200 OK	Standard status code for successful responses.
201 Created	The request has been fulfilled and as a result a new resource was created.
202 Acepted	The request has been accepted for processing, but the processing has not finished yet.
204 No content	The request has been fulfilled, but the server is not sending any content as a response.
206 Partial content	The server is responding only with part of the resource (byte range).
301 Moved permanently	The request has permanently changed URI, and the following requests should go to the new URI.
304 Not modified	Indicates that the resource has not been modified since the last time it was requested. Useful for client-based cache.
400 Bad request	Standard response for an invalid request.
401 Unauthorized	Used when the client is not authorized to access the requested resource.
404 Not found	When the requested resource could not be found in the server.
500 Internal server error	There was an internal server error and the request could not be processed.
501 Not implemented	The server does not recognize the request method or does not know how to fulfill it.

Table 2.2: HTTP Response status codes

When sharing a file through Bittorrent [16], first a ".torrent" file has to be generated. This file is then shared through a conventional network, like a webpage or email. This torrent file contains metadata about the file to download, like the number of pieces the file has been split to, a cryptographic hash of the pieces to ensure they are not tempered when downloading, or the tracker to access to look for other peers of the file.

When a torrent file is opened with a Bittorrent client, it knows how to read the torrent's metadata, and then starts connecting to other peers and requesting different pieces from the file until the download is completed.

The following list contains important concepts and vocabulary used in Bittorrent:

- Peers: Each client connected to the network is a peer.
- Leechers: The clients that are downloading a file but do not have completed it.
- Seeds: Clients that already have the complete file downloaded, and are only sharing the file with others. It is important that any file shared through the network has at least one seed. Without it, it could happen that nobody has a certain piece of the file, hence the file cannot be downloaded completely.
- Swarm: The network created between all the peers that are downloading the same file.
- Tracker: Server that allows peers to know about the networking information of other peers in order to start a connection with them. It also contains statistics about the swarm created for a specific file, like number of seeds and leechers.
- DHT: Stands for distributed hash table, that contains network information of other peers. It is a decentralized alternative to the tracker to discover peers of a certain file.

Bittorrent specification is subject to change, and can also include different extensions over time to adapt to future improvements to the network. These extensions are called Bittorrent Extension Protocol (BEP) [17].

2.4.3 Integrating HTTP and Bittorrent through WebRTC

At this point, seems logical to create a protocol based on Bittorrent that would work over the WebRTC network as a substitute for TCP and UDP protocols in the original Bittorrent. Some important changes would need to be addressed in order to have an easy to use and reliable platform.

- No need for a .torrent file to start the download. Being a web platform, relying on external files to start the downloads would only complicate things. Ideally the information stored in a .torrent file could be stored somehow without physically storing the file itself.
- As mentioned before, the protocol would use WebRTC as communication layer between the peers, being the only one supported in the web ecosystem.
- When using a Bittorrent application, the user that creates a torrent from a file, becomes a seed of that file (has the complete file in disk). In this case, the administrator would upload the file to the platform, and the platform has to become the seed of the file somehow, in order to be downloadable if the administrator disconnects from its computer.

At this point, looks like a Bittorrent-like protocol implemented for WebRTC is needed. After some search, I found WebTorrent [18], which is an implementation of Bittorrent using WebRTC, what I was looking for. We could use directly Webtorrent, or develop a new solution inspired on Webtorrent. This is a list of advantages and disadvantages of using Webtorrent.

- It uses the original Bittorrent specification.
- Already functional and actively developed.
- Similar performance to Bittorrent

As for any disadvantage, the only one is that Webtorrent still requires the .torrent file to work, so a workaround will be needed to use Webtorrent without storing the .torrent files directly.

After considering the advantages, a prototype will be developed to ensure that the functionality that we are looking for can be implemented.

After implementing the prototype [19], we see Webtorrent can be used for our platform. We can adjust to the mentioned changes by means of specific workarounds using Webtorrent. The change about WebRTC is already solved by the package.

- The .torrent file could be stored as a byte Buffer [20] in a database like MongoDB [21][22], and then sent to the client as a byte list. This way, the .torrent file never needs to be stored manually, and does not virtually exists as a file, only as data in a database.
- For ensuring that the file is always accessible, we can use a BEP (Bittorent extension protocol) called Webseed [23] that allows the download of a file through HTTP, thus complementing the peer-to-peer network. In this case, the HTTP server acts as another peer that contains the full file.

Now that we have seen tht we can use Webtorrent for our needs, let's see what exactly is Webtorrent and where it came from.

2.4.4 WebTorrent

WebTorrent[18] is an implementation of the Bittorrent protocol initially developed by Feross Aboukhadijeh, open sourced through the platform Github [24].

It does not only work in the browser, but also implements the classical Bittorrent through TCP and UDP. This way, if the package is used from the browser, it will use WebRTC, while if it's used from a computer through a Node.js environment [25], it will use TCP and UDP connections like a normal Bittorrent client.

It is important to mention here that the peers connected through TCP and UDP are not able to connect directly to WebRTC peers [26]. Because WebRTC peers are not yet specified as Bittorrent extension protocols, classical Bittorrent clients do not implement WebRTC peers, and for WebRTC peers it is impossible to connect to TCP and UDP peers. Webtorrent has an issue open [27] to make other popular Bittorrent clients also implement WebRTC peers.

The project is already functional, although not in a stable state. Lots of issues are still open, and a number of contributors are working to make Webtorrent more stable. The first release of the project was launched in December 2013.

2.4.5 Docker

Another technology it is important to mention in this project, although not directly related to Webtorrent, is Docker. It is important because it will be the technology used to deploy the applications in the cloud. Besides, the platform will be adapted to ease the process of deploying it.

Docker [28] is a virutalization technology that emerges from another technology called LXC Containers [29]. What Docker does is to create a virtual system that shares memory spaces with the host machine, but contain its own software, library space and file system. Docker creates virtual images that can later be deployed into containers.

This way applications can be updated dynamically, without affecting the host machine or its libraries and software. They are also portable between the different operating systems that support Docker technology. When encapsulating applications with docker, we can also deploy more than 1 container of the same application, even in different servers with an external load balancer, which creates endless possibilities regarding scalability and reliability.

Big companies like Paypal, Ebay or Spotify [30] have their applications deployed under Docker. It is actually in version 1.8 and is in constant development and with frequent updates [31]

This technology will be used after the platform is developed to deploy it live and configure the production environment. Although the technology won't be strictly needed to deploy the platform, it will be the recommended technology for an easy deployment.

2.4.6 Similar solutions

In this section we will cover different solutions that have already been developed or are in development that solve a similar problem. These solutions will be divided in two categories. One category will contain the solutions that are cloud platforms, self-hosted and that might have multimedia streaming, but usually do not include decentralized downloads, while the other category includes the platforms that have specifically targeted decentralized distribution and are potentially using Webtorrent or any other technology to offer content, but might lack the functionalities of a self-hosted storage platform.

The list of the first category that we describe include platforms like ownCloud, Seafile, and more described below, and are similar solutions to the platform we want to build:

- **ownCloud**: It is one of the most used self-hosted clouds in the market. They're code is open source, and can be deployed by anyone with minimum knowledge about system administration. With ownCloud you can share your files, calendar, contacts and mail from any device with your team. It has a marketplace of third party applications to extend its functionality, like media players, polls, etc...
- **Seafile**: Similar to ownCloud, it is a self hosted application that can be accessed from anywhere with its cross platform client applications. It has interesting features like version control of your files, file locking and online editing and co-authoring of the files.
- **Filecloud**: Another platform, similar to ownCloud or Seafile, in this case it offers the option to be self hosted, or hosted by the company. It has integration with Active Directory and NTFS. It also integrates with Office and Outlook, and (like Seafile) allows for remote editing of the files in the cloud.

In the second category are included some solutions that use decentralized downloads or synchronization, but might lack the functionalities of self-hosted storage platform:

- **Resilio**: Formerly known as Bittorrent Sync, and developed by the original authors of Bittorrent. It uses Bittorrent to synchronize different files over private networks. It can be used to distribute updates to vehicles, synchronize servers, or distribute software faster. The software comes in different packages, from Home use to teams or enterprise solutions.
- **instant.io**: This project is a simple project that demonstrates the use of Webtorrent to distribute files over the web. The webpage allow us to upload a .torrent file or paste a magnet link to download the torrent file, or upload a file of our own and it will generate the .torrent file to distribute that file.
- **Fastcast**: Web application that distribute different videos in high quality by harnessing Webtorrent as download technology. It is also open source, and therefore can be self hosted.
- **GitTorrent**: Peer-to-peer network to host git repositories over a Bittorrent network.

2.5 Project scope

This project is divided into different self-contained units. This means that when a unit is finished, the project will be functional and can be used. In the first part, a MVP (Minimum Viable Product) [**MVP**] will be created. From that point, the following parts will add value to that MVP. Since the project has a specified timeline to be finished, we can ensure that a functional project will be delivered, even though some of the final units can be missing.

2.5.1 Project units

This section will describe the different units mentioned before and a small briefing of what they will be. In further sections, timeline and cost of each of the units will be specified.

2.5.1.1 Minimum viable product

The minimum viable product contains the following components:

- **Admin panel**: Separated webpage or section inside the platform to manage the files that will be displayed for visualization or download. Other component that might exist in this part is the user management, to create or invite other users to join our platform in a private way.
- **Downloads panel**: This webpage or section of the platform will show all the files available for download, and will contain the webtorrent client to download them. If streaming is finally solved and included, it will also be in this panel.
- **Tracker**: A private Bittorrent tracker to connect the different peers of the networks created by our own files. It can be just a simple tracker with no business logic, or it can contain rules to allow or reject different users and torrent identifiers.
- **Webseed**: When we upload files to the platform, they will be publicly served through HTTP. As explained before, this will allow a 100% percent of availability of the file, even when no other peers are connected to offer other pieces of it.

- **Database**: All files stored in the platform will contain its metadata in a database, to be able to discover which files are available and the torrent information needed to start a bittorrent network to download that file. It can also be used to store other information related to the platform, such as users, invitations or folders.

This unit contains the biggest amount of work. After this milestone is complete, we will have a working platform that can be used. But developing the following units will help adoption of the project and a better distribution and deployment of it.

2.5.1.2 Docker production environment

Because this project contains different standalone services, Docker technology is ideal to deploy them independently. It can allow for massive flexibility regarding scalability and failure resistance, as different services can be deployed in different servers.

Docker also allows us to redistribute images of the different parts of the platform through Docker hub [32], making them publicly available to use directly from a Dockerfile or the docker command line interface.

2.5.1.3 Simple portability

Once the different parts are available through Docker, it can be adapted to be easily configured to match any server environment.

In this unit of the project, the goal is to put into enviromental variables everything that can be configured regarding deployment, from different companies with different domains, to also different developer environments like development or testing.

2.5.1.4 Project webpage

After finishing the portability with docker, everything regarding the platform is done. In this unit, a landing webpage with information of the project will be developed.

In this static web page, some information like the features of the platform, or how to deploy it will be present. With this information, we try to improve the adoption of the project by the public.

2.5.1.5 Webtorrent.io collaboration

As a final unit in the project, it would be ideal to contribute to the webtorrent project with code or issues.

2.5.2 Risks involved

To develop this project there are different risks that we have to assess before starting the development, in order to prevent them in case something goes wrong.

2.5.2.1 Webtorrent is not in stable version

The webtorrent library is not yet in a stable release. It can contain bugs or could not be suitable for production environments with many users. We are limited by what the library can or cannot do.

In the case we find bugs that affect us, one solution can be to try and create a fix and get the fix into the library's official repository. This way the next release of the library will contain the bugfix.

It could happen that we need more important fixes or changes in the library, that are more difficult to get into the original library because they modify the way the library works or are not suitable for everybody. In this case we could create our own fork of the library, and start developing things for our platform on top of that fork.

2.5.2.2 Difficult distributed testing environment

The nature of the project is to decentralize downloads. But to test this property is a difficult task, because Bittorrent scales better the more peers are on the network, and this is difficult to test in a local environment without real network conditions.

It can even be difficult to test decentralization locally, because the WebRTC peer-to-peer connections might not be adecuate for internal LAN or computer connections.

Ideally, we could test at least that two computers in different networks are able to share a single file without requesting the whole file from the webseed. This would mean that Webtorrent works decentralized, and we can assume it would work with more peers connected.

2.5.2.3 Webtorrent software integration

Because this project has many different components that have to coexist with each other, it could happen that some part does not work as we expected or contains bugs that force us to take a different path in development.

Specifically, different bittorrent related projects will be used, like torrent-create [33] or the bittorrent-tracker [34]. Also the integration with the webseed can be tricky, because Webtorrent executes the webseed requests with the range header, and we need a webseed server able to respond to those headers.

2.5.2.4 Environment not mature enough

All the development tools we are going to use were developed in the recent years or even months. This has its benefits and its disadvantages. It usually makes the life of the developer easier by developing faster and with more robust patterns, but on the other hand it can be risky as those tools might still have pending issues to solve.

In this category we can include complementary tools to Docker, web development tools like webpack, front end frameworks like Angular or backend frameworks like Koa, and the different plugins and tools built around them.

2.6 Methodology

The project has different units, so I think the methodology that best fits the development of the project is an Agile [35] methodology, more specifically SCRUM [36].

SCRUM is normally used in small teams, so in this case it will be adapted to the needs of this project, with project units that are already defined, and with only one developer.

2.6.1 Agile approach

Like in SCRUM, this project will be developed in different **sprints**, that will have a fixed duration depending on the quantity of work specified for each one.

In this section we will describe some concepts needed to understand the methodology and structure, most of them inspired in SCRUM or other agile methodologies.

- **User story**: The basic unit of sprint planning. It contains a certain functionality to be developed, a bug to fix or other self contained tasks (Like generating documentation) that add value to the project. It may group other smaller tasks related to each user story together.
- **Backlog**: The backlog is the list of user stories that are defined to be picked in any sprint. This is a wide list with all the user stories of the project updated. They are usually specified by the project manager, and then the developers decide how much time and effort every one needs. In this project, the backlog is already splitted for every sprint, but this does not need to be always the case.
- **User points**: It's a score given to each user story. It has a direct relationship with the quantity of work needed for its development, and can be used directly with a 1 hour = 1 user point relationship, or also adapted to more complex planning.
- **Sprint planning**: Phase of the sprint where a selection of user stories are picked from the backlog to be developed during the current sprint. During this phase the developers consider how many user points correspond to eash user story, in order to know if they will be able to carry on with all the user stories.
- **Sprint retrospective**: At the end of each sprint, a sprint retrospective is where it is analyzed how the sprint went. What was done properly, and what can be improved in the following sprints is pointed out, as well as if the time estimation for each of the tasks was the appropriate one. This gives a flexibility to the team to adapt from one sprint to another on the methodology or the way to work, in order to try and improve on each sprint iteration.

Unlike classical SCRUM, where there is no final planification for the whole project on each sprint, we will have a final planification split into sprints, so instead of having a sprint planning each beginning of sprint, all sprints will be already planified and assigned tasks, and the sprints will be adapted while the project is being developed according to the results met at the end of each sprint.

In this case, our backlog instead of containing the full list of tasks needed to end the project, will be adapted to contain tasks that are found during the development of the project, but that still don't have a specified sprint assigned.

2.6.1.1 Evaluation system

At the end of each sprint, a sprint retrospective will be done to analyze the work accomplished. This analysis will contain things that have been done properly, and other things that are considered to be improved in the next sprint.

Also, after finishing each of the standalone units of the project, a review will be done to ensure it meets the expectations and describing the problems found and the developed solutions that may appear while developing the project.

2.6.1.2 Project tracking

The Agile organization of the project will be tracked in the Taiga [37] platform.

The code will be offered open source through my Github profile [38], and whatever is done will be always available, regardless of the state of the project.

2.7 Project planning

2.7.1 Calendar

The project has a total duration of about three months and a half, from end of September to mid January, when the defense of the project is set. The project will start at the last week of September.

Because Christmas is happening just before delivering the project, the following plan tries to finish the project by december, to spend the rest of the time in the dissertation and the preparation of the presentation. Thirteen weeks of work are planned, leaving January as margin, to finish the dissertation and presentation.

The actual implementation of the project could change depending on the situation.

2.7.2 Project units specification

This section contains each of the different units of the project, and its sprint planning with User stories and user points assigned.

2.7.2.1 Preparation unit

This unit contains some aspects of the project that are necessary before starting the development of the project. It consists of one sprint of one week duration, described below.

Sprint 0 - Environment preparation and investigation -*duration: 1 week*

The main objective in this sprint is to setup the tools needed to work on the project, and create a diagram and a prototype that will help us understand the technologies to use and make sure we are on the right way and the project is feasible.

- **External services**: In this user story are included tasks to preparare the external services that we will need to track the project and the code, along with other services like infrastructure. Github and Taiga will be created for the tracking, and DigitalOcean and Mongolab accounts will be created for the server infrastructure and database. *3 UP*
- **Platform diagram**: In this task, a diagram of the overall architecture of the platform will be created, in order to have a general view of the different components and how they will connect to each other. *5 UP*
- **Prototype**: Prototype to integrate webtorrent into a webpage, where a file can be uploaded to a server, generate a torrent file (in bytes, not stored directly) and check if that file,can be downloaded. It is critical to solve this part, to make sure we are on the right way with the project, and that it is technologically possible to do what the platform needs. *20 UP*

Total UP: 28

After this unit, if everything goes well and the prototype works as we expect, then we will move forward to the development of the minimum viable product.

2.7.2.2 Minimum Viable Product

The minimum viable product planning contains three different sprints, that contain the different parts we specified before.

Sprint 1 - Admin panel and webseed -*duration: 2 weeks*

The main objective for this sprint is to create the administrator platform, that will allow the administrator to upload files, and manage the rights and settings of the platform. Also, we will configure the ngnix to serve the files through HTTP (webseed).

- **Admin REST API**: Create an API rest for the admin panel to upload the files and manage different users and settings. *20 UP*
- **Admin Frontend**: Design and implementation of the admin panel frontend. *30 UP*
- **Nginx configuration**: Configure an nginx to serve the RET API and the frontend files. Configure the same nginx or a different one to also offer the files to be downloaded as a webseed. *10 UP*

Total UP: 60

Sprint 2 - Downloads frontend - *duration: 2 weeks*

For this sprint, the main objective is to develop the downloads platform, from frontend to back-end, and also setup the tracker that will allow us to connect our users to create a bittorrent network.

- **Simple tracker**: A tracker to test the platform, without complex functionalities like filters for files or connections, just a deployed bittorrent tracker. *5 UP*
- **Downloads REST API**: A REST API that allows the downloads platform to list the different files available, along with structure into folders and user management. *10 UP*
- **Downloads with webtorrent**: Configure and develop the services needed to download the files via webtorrent. *5 UP*

- **Downloads frontend**: Design and implementation of the web application to download file. *15 UP*
- **Connect to REST API and tracker**: Connect the frontend to our API and the tracker to display information about the torrent network of the files. *10 UP*
- **Progress module**: A separated module to display the download progress of the files. *20 UP*

Total UP: 65

Sprint 3 - Review of the platform - *duration: 1 week*

In this sprint, the main objective is to review what we have done until this point, finish anything that could not be finished on time, add anything to the project that we might find interesting. In case nothing extra needs to be done, there are some tasks to improve and properly test the platform.

- **Advanced tracker**: Take advantage of other functionalities that can be developed on the tracker, like filter the files requested to only the ones available in our platform. *5 UP*
- **Different seeds testing**: Test that the platform works properly with an actual network of different peers connected from different networks.*10 UP*
- **Review server configuration**: Make sure all the configuration is optimal, and review the security of the platform. *5 UP*
- **Automated testing**: Develop an automated testing suite for the platform, to make sure everything works properly, and also to help future developments test the platform for potential bugs introduced. *15 UP*

Total UP : 35

After this unit is finished, we should have a working platform to try locally and start thinking how to deploy, distribute and document it.

2.7.2.3 Production server with Docker

This unit contains the work needed to deploy the platform to a production environment.

Sprint 4 - Docker configuration - *duration: 1 week*

The main objective for this sprint is to start using docker. The docker images will be created for each of the components of the platform needing one, and any configuration or development needed to deploy these images will be created.

- **Install and configure Docker**: Download the necessary packages to run a docker environment, locally to test and in the server for production. Configure the docker client if needed. *5 UP*
- **Create docker images**: To create an image, a Dockerfile needs to be created first. The Dockerfile specifies the steps to create an image. In this task, all the Dockerfiles for each component of the platform will be created.*10 UP*

- **Configure docker network**: Look for and implement a technology to communicate dockers with each other internally, in order to have more efficiency on the communication. It will also allow to distribute docker containers to different servers with an abstraction that connects them independently of being on the same or different servers. *5 UP*
- **Deployment scripts**: If necessary, create scripts to deploy the different containers. *5 UP*
- **Testing**: Test that the production deployment works properly and apply any needed bugfix. *5 UP*

Total UP: 30

After this unit is finished, we should have a working production environment that would allow us to deploy the project and use it, but it is not yet adapted to be distributed and portable to different environments.

2.7.2.4 Platform portability

In this unit the platform will be adapted to be more portable. This means that everything related to the infrastructure, like URLs, credentails or ports, will be abstracted into a configuration that will allow other users to easily deploy the platform into their infrastructure.

Sprint 5 - Platform portability - *duration: 1 week*

The tasks for ths sprint are not yet properly defined, as it will depend on the development done until this point, and also of the requirements necessary to make the platform more portable. Because of this, an analysis of what is needed will be done before anything else to know what needs to be done.

- **Requirements analysis**: First of all an analysis of the current development needs to be done to assess which parts need to be adapted or changed in order to facilitate the portability.
- **Implement portability**: Work on the changes found on the analysis previously done.
- **Distribution**: After the portability issues have been solved, it is time to distribute de platform through docker hub and github.
- **Documentation**: Some documentation will be needed to explain how to achieve the portability and deploy in any infrastructure.

There is not enough information to know the user points needed for this user stories, but the intention is to keep it a normal 1 week sprint of around 30 to 35 user points.

2.7.2.5 Landing page

In this unit, a landing page for the project will be created. This will help to let know people about the project and also to host some basic documentation. As well as the previous unit, the tasks cannot be properly defined yet, until the project reaches this point and an analysis of functionality is done for this landing page.

Sprint 6 - Platform landing page - *duration: 1 week*

- **Functionality design**: Analysis and design of the functionalities and requirements of the landing page.
- **Implementation**: Implement the solution proposed by the previous task.
- **Deployment**: Deploy the landing page to a server, with its own domain.

Around 30 to 35 UP

2.7.2.6 Webtorrent collaboration

After finishing all the previous tasks, if there is time left and all the implementation went according to the planned timeline, contributions to the webtorrent project will be done.

Inititally, the methodology for this unit is not yet specified. Depending on the contributions that are chosen to be made, it could be feasible to organize them in a SCRUM way, like the rest of the project, to better organize the work to do.

The duration of this unit will be the 4 weeks remaining

The contributions can be of different ways, these are the most common ones:

- **Code contributions**: Through 'pull requests', we can offer changes to the original code of the project, that will be reviewed by the collaborators of the project, and they could ask for changes before accepting the new code into the project's original code.
- **Issues**: By creating or commenting on issues. If a bug is found it is a good practice to open an issue, to let everyone on the community know about that bug. By commenting on issues, we can sometimes offer help to others using the project.
- **Gitter**: Webtorrent has its own Gitter channel with people talking about the project, by being there we could ask for help or offer help to other developers.

2.7.3 Action plan and alternatives

There are some elements in the planning that allow for certain flexibility, allowing the delivery of a finished project, even though if some of the timelines are not strictly met. This is the list of these flexibility measures:

- **Unit organization**: As decribed below, the project is divided into different self contained **units**, that allows for the delivery of a functional project even if not all the units are realized.
- **Agile planning**: With sprints and the different user stories, if one sprint is finished before the expected date, some tasks of the next one can be moved to the current one, speeding up the development. On the other hand, if a sprint cannot be finished on time, the tasks and user stories on it can be moved to the following sprint, deciding if any task can be discarded.
- **Collaboration unit**: The last unit of the project is the most flexible one. In the worst scenario, it could be taken to finish user stories of the previous sprints if considered that those are more important than any collaboration to the platform.
- **Sprint margin**: Most of the sprints do not arrive to the maximum UP proposed by working 7h a day. This is 35 UP for a one week sprint, or 70 UP for two weeks sprint. This gives some extra time to solve any problem that may come up during that sprint's work.

2.7.4 Gantt diagram

See figure 2.1

2.8 Resources

In this section are documented the resources that will be needed for the realization of the project.

2.8.1 Physical machines and servers

- Laptop Macbook Pro 2015 [1] as main development tool, and to connect to any other external service or server.
- Virtual server in DigitalOcean, with 1GB RAM, 1 CPU, 30GB of SSD amd 2TB of network transfer per month.

2.8.2 Developer tools

- Git, as code version control system tool.
- VisualStudioCode, as main text editor to develop.
- Google Chrome Developer Tools, to make tests and debug the code while it is being developed.

2.8.3 Cloud tools

- Github to remotely host the git repositories of the platform, and share them with the community.
- Taiga, tool used for the tracking of Agile projects.
- Docker Hub, that will host our Docker images created to use and share them.
- Sharelatex to create the documentation of the project.

2.8.4 Communication tools

- Email, mainly for the communication with the project director, but can always be used to communicate with other people that may relate to the project in the future.
- Github is also considered a communication tool, because it connects developers with each other, and a lot of discussion is created through issues and pull requests.
- Gitter, chat tool used by the webtorrent project along with their IRC channel.

[1] CPU Intel Core i7 and 16GB of RAM

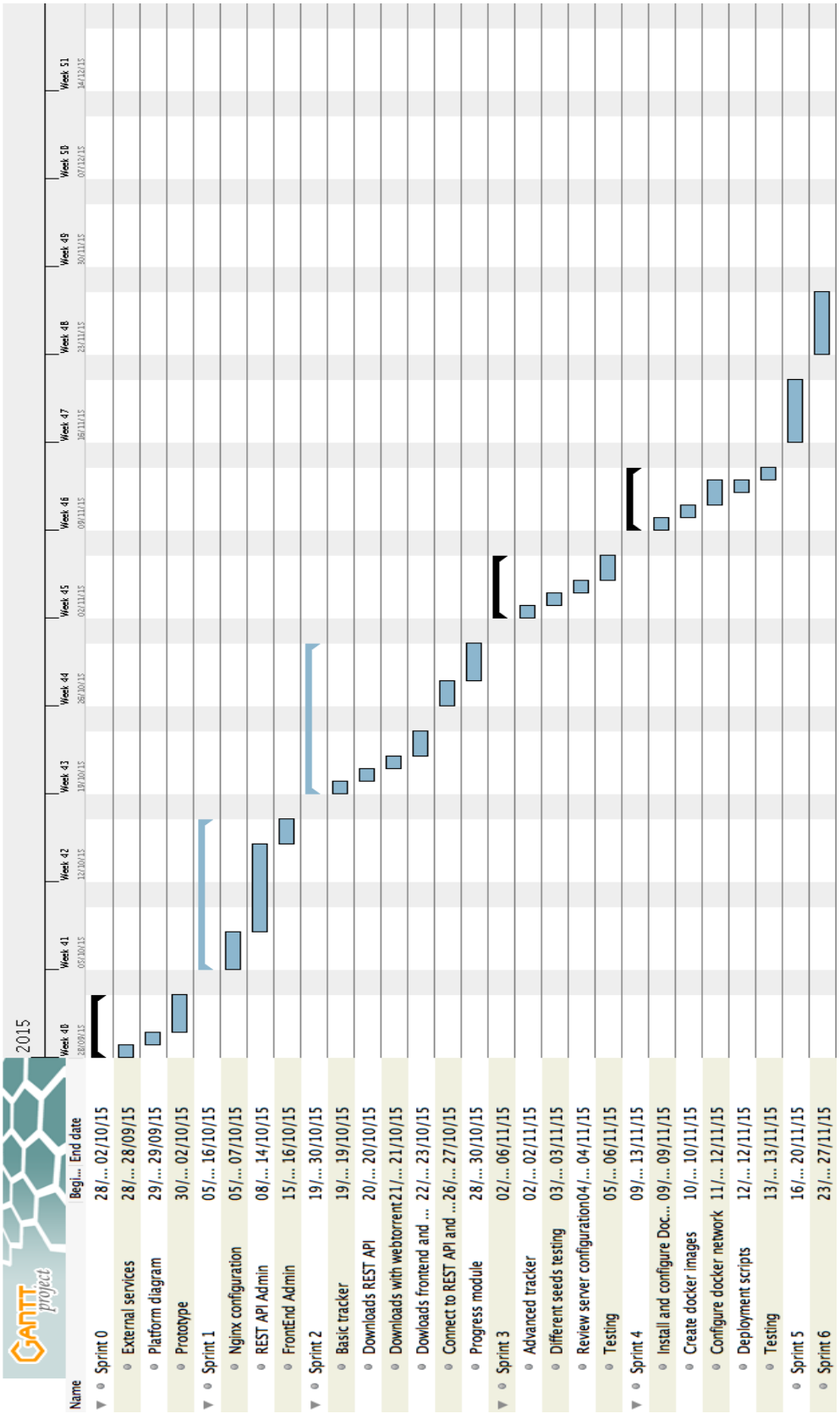

Figure 2.1: Gantt diagram

2.8.5 Other tools

- draw.io, to generate diagrams.
- GanttProject, used to create the Gantt diagram of the planning.

2.9 Cost estimation

This section identifies and describes the different costs of the project. As specified in the planning, the project has a duration of thirteen weeks, with a weekly work hours of 35 (7h a day for five days a week). There are different roles involved in the realization of the project, and the amount of time needed of each of them may vary each unit, so the personal costs will be divided by project units.

As mentioned in the planning, each sprint has some hours left that can be used for potential modifications on the realization of the tasks, extracting from these hours the contingency plan.

There is also included the Scrum Master role, that will take roughly 1 hour from each sprint.

The project is realized by 1 person only, that will do all the different roles specified in the project, and that will not take any money from it. The prices specified are only an approximation of a real life situation.

2.9.1 Roles

These are the descriptions of the different roles mentioned before:

- **Scrum Master**: This role's job is to do the planning of the project and the necessary modifications. The User stories chosen for each sprint, and the user points assigned are part of the Scrum Master's job. Any modification needed in the planning to adapt to the speed of development is also part of this role's responsability. *Cost: 25 €/ h*
- **Developer**: The main role of the project, responsible of programming the platform. We assume the developer also has analysis and architecture skills, so it can design the technological solution almost entirely, without needing other roles like software architect. *Cost: 20€/ h*
- **System administrator**: Another important role in the project, specially in the third and forth unit of the project, in the integration with docker and the production environment. It also gives support to the developer in other phases, in parts were the development needs to adapt to the infrastructure. *Cost: 20€/ h*
- **Tester**: Role that is responsible to test the functionalities of the platform, manually and programming automated test suites. It creates and makes sure the platform achieves the quality standards. *Cost: 20€/ h*
- **Project director**: This role, although being very important, has few hours in the planning. Its main objective is to supervise the whole development of the project. It creates the cost and time plans and setups the external services. It also makes sure the changes and modifications made by the Scrum Master are applied to the actual development of the project. It also creates the documentation for the project. *Cost: 40€/ h*

2.9.2 Personnel costs

This section describes the personnel costs per each of the sprints planned for the project, in detail with each of the user stories. Also, the margin hours of each sprint are also accounted for, as part of the contingency plan. These sprints and user stories are directly related to the Gantt diagram in figure 2.1

Aside from the user stories described before, the personnel costs considers also print planning and Documentation as other tasks that need to be done in each sprint.

The following tables contain the breakdown of costs for each sprint:

Sprint 0 - Preparation unit						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Sprint planning		1				25 €
Documentation	2					80€
External services	3					120€
Platform diagram	5					200€
Prototype			15	5		400€
Total						825€
Total hours				Unassigned		
31				4		

Table 2.3: Sprint 0 costs

Sprint 1 - Admin panel and webseed configuration						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Sprint planning		1				25 €
Documentation	2					80€
REST API Admin			20			400€
FrontEnd Admin			30			600€
Nginx configuration				10		200€
Total						1325€
Total hours				Unassigned		
63				7		

Table 2.4: Sprint 1 costs

Sprint 2 - Downloads frontend						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Sprint planning		1				25 €
Documentation	2					80€
Basic tracker			5			100€
Downloads REST API			8	2		200€
Downloads integration			5			100€
Files list design and implementation			15			300€
Connect to REST API and tracker			10			200€
Progress module			20			400€
Total						1405€
Total hours				Unassigned		
68				2		

Table 2.5: Sprint 2 costs

Sprint 3 - Review of the platform						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Sprint planning		1				25 €
Documentation	2					80€
Advanced tracker			5			100€
Different seeds testing				2	8	200€
Review server configuration				5		100€
Automated testing					15	300€
Total						805€
Total hours				Unassigned		
38				-3		

Table 2.6: Sprint 3 costs

Sprint 4 - Production server with Docker						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Sprint planning		1				25 €
Documentation	2					80€
Install and configure Docker				5		100€
Create docker images				10		200€
Configure docker network				5		100€
Deployment scripts				5		100€
Testing					5	100€
Total						705€
Total hours				Unassigned		
33				2		

Table 2.7: Sprint 4 costs

In the sprints 5 and 6, we will consider that we need all the unassigned hours after assigning sprint planning and documentation, even if it's not clear yet what the tasks will be.

Sprints 5 - Portability						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Scrum planning		1				25 €
Documentation	2					80€
Other				32		640€
Total						745€
Total hours				Unassigned		
35				0		

Table 2.8: Sprint 5 costs

Sprints 6 - Página web						
User stories	Director (40€)	Scrum Master (25€)	Developer (20€)	System admin (20€)	Tester (20€)	Total €
Scrum planning		1				25 €
Documentation	2					80€
Other				32		640€
Total						745€
Total hours				Unassigned		
35				0		

Table 2.9: Sprint 6 costs

In the unit of contributions to webtorrent, the remaining 4 weeks of the project, it is the developer

on its own who handles all the work and communication needed, so it will not need of scrum master or project director.

Webtorrent contributions		
User stories	Developer	Total €
Contributions	140	2800

Table 2.10: Webtorrent collaboration costs

2.9.3 General costs

For the general costs, we are going to assume that we work on the coworking space of Espai Emprèn, inside the campus of the Polytechnical University of Catalonia, with a mensual fee of 20€for companies with just 1 developer. There is also included the cost of the personal computer, servers and transport. Everything else is free for use or in trial period.

General costs		
Description	Monthly fee	Total €
Personal computer		1850
Coworking space	20	80
Transport	100	400
Servers	10	40
Total	130	2370

Table 2.11: General costs breakdown

2.9.4 Maintenance

The project will require some maintenance, divided and described below into two different fields:

- **Code**: The code maintenance, new features, bugfixing and open source collaborations. As an open source project, the maintenance will be done by the developer in its own free time, and by any other developer that wants to contribute to the project.
- **Server**: The server in DigitalOcean where the platform will be initially hosted for demonstration. Initially, we have 100$ credit thanks to Github Student Pack. After this credit is used, either the developer or other contributors will pay the server, or a free alternative will be used.

2.9.5 Potential unexpected events

During the development of the project, some unexpected events might happen that affect the planning of the project, either temporally or economically. These are some of the unexpected events we may encounter:

- **Library bugs**: The libraries we want to use can contain bugs that prevent us from doing something important to the platform. This will cause that more hours are needed for a certain

task, either because we have to fix the bug in the library, or change the library for another one.

- **Design errors**: The infrastructure we want to develop is complex and with lots of elements. Something could not work as we expect it to, involving a redesign that would cost a certain amount of hours, and possibly a redistribution of some tasks or even planning.
- **Delays**: Other unforeseen factors could affect the planning causing some delays.

2.9.6 Contingency plan

To assume the possible unexpected events, the contingency plans to implement the following measures to ensure the realization of the project in the given time:

- **Unassigned hours**: We have some hours left in some sprints described before, that will be included in the economical plan at a price of 30€/h, a value between the best and worst paid roles.
- **10% added**: There is a 10% added on top of the personnel costs, to account for extra hours needed to solve the unexpected events that might happen, or extra services that need to be added to the project.

2.9.7 Total budget

The following table sums up the total costs of the project.

Total costs	
Costes de personal	9355€
10% Contingencia	935€
Unassigned hours	360€
Costes generales	2370 €
Total	13020€

Table 2.12: Total cost of the project

2.9.8 Amortization

This project is not planned to get any economical benefit or amortization from it, not directly at least. Indirectly, some contributors to the project could donate to pay the developers working on it, or the developer could be hired as a consultant to setup the platform or create a customized version of it.

2.10 Management control

In each of the sprint summaries, there will be a recap of all the hours each of the role has dedicated to its tasks. This way, we can control the deviations in each sprint, if any.

In the webtorrent collaboration unit, we will account for all the expenses of doing the contributions, if any. It could be attending conferences, additional software or courses.

2.11 Sustainability

2.11.1 Economical

The project is not directly sustainable economically, but it has an economical impact on th users that decide to use it. Because it encourages all connected computers to share the bandwith to distribute files, instead of having a central server, the more users and traffic the company has, the bigger impact on reducing of costs.

The collaboration with another Open Source project can also make an impact on the users of that project, as it enforces decentralization.

We believe the competitive advantage that the platform and technology can offer, by means of reduction of infrastructure costs, compensates the implementation costs.

Score: 7

2.11.2 Social

Nowadays, the commonly used paradigm for downloading files is client-server, but this project enforces a more decentralized way of sharing the files, and, by this, the resources and network use is more decentralized.

This way, the network tariff of the users would be used to distribute files, which reduces the number of servers needed by the companies, and the costs associated with it.

The companies that offer servers and cloud services might be affected by this project, reducing their income because the companies that adopted this project would need much less servers. This companies offer a wide range of cloud services, and only a few of them would be affected.

Score: 8

2.11.3 Environment

The enviromental impact of this project is minimal. It would only affect the machine it will be developed with, and the energy consumption of this machine and the production server.

Also, this project is highly reusable and adaptable, so companies could use it to reduce their ecological footprint. With open source, the companies that use it can also contribute to its development and adoption.

Score: 8

2.11.4 Sustainability table

Economical	Social	Environment	Total
7	8	8	23

Chapter 3

Realization of the project

This chapter describes the development process of the project. Starting from the already seen prototype, it goes through the overall architecture designed, and then explaining the design and development of all the different components.

Each component contains a retrospective that explains how it adapts to the plan previously defined, and the modifications that have been done, if any, with its justification.

These components might not follow the same separation that was specified in the previous chapter, due either to modifications on the final implementation, or for clarity purposes when following the dissertation.

3.1 Prototype

The prototype, found as a task in sprint 0 at 2.7.2.1, is a general proof of concept for the technology. It has been done previous to any further work, in order to make sure that the process we are planning can be executed with the current state of the technologies that are going to be used.

The code of this part can be found at [39].

3.1.1 Objective

The objective of the prototype is to prove that the following steps can be developed and work as expected:

1. From the webpage, a file can be uploaded to the server.
2. From the server, this file will be stored in disk.
3. From the server, a torrent file will be generated for the uploaded file.
4. From the webpage, I can get the torrent file generated by the server.
5. From the webpage, the torrent file prevously downloaded can be used with the Webtorrent package to download the file at hand.

The code for all of the steps is shown, as these steps are crucial to make the project work, and it's important how code is gluing the different components and functionalities together.

3.1.2 Architecture and technologies used

The overall architecture of the prototype is fairly simple. It consists of a server process, and an static webpage that connects to the server with Ajax [1] requests. The code of both components is inside of the same repository. In the folder *client* of the repository we find the code for the webpage, and the code for the server can be found at the file *index.js*.

The client is a simple html webpage, that uses some utility libraries like jquery [2] (to setup ready event on Javascript) and stream-http [3] (to make http requests to the server). It also uses the Webtorrent library to test that the download of the file from the torrent works properly.

The server is a web server created with the library koa [4] and some helper libraries related to koa that can be found in the code. Another important library used in the server is create-torrent [5] to generate the torrent file.

3.1.3 Development

For the different steps of the objective of this prototype, it is showed here the part of code that handles that task, and a description if needed.

3.1.3.1 Upload file to server

To upload a file to a server, there is code needed on the webpage and on the server.

The part of the client is split into the HTML part and the Javascript part. The HTML part contains a form with an input of type *file* and a button to trigger the upload. See the listing 1. The Javascript part (listing 2) adds a callback for the submit of the form (triggered when clicking the button inside the form) to do an AJAX request to the server of type **POST** to the endpoint /**upload**/ with the **FormData** of the element (in this case, contains the file to upload).

```
<form id='uploadForm' name='uploadForm'>
  <input type="file" name="fileToUpload" id='fileToUpload'>
  <!-- <input type='submit' value='Upload' id='uploadButton'/>-->
  <button id="uploadButton">Upload</button>
</form>
```

Listing 1: Prototype: Upload a file on the client HTML

[1] Ajax [40] is a way to make HTTP requests to a server from Javascript and the browser
[2] Jquery [41] is a Javascript library that contains many helpers to simplify our work using Javascript.
[3] stream-http (https://www.npmjs.com/package/stream-http) is an implementation of the http module of node.js to be used on the browser.
[4] Koa [42] is a webserver library for node.js that allows to easily setup endpoints and be accessed through http
[5] create-torrent [33] is a package that allows to generate a torrent file from the file at the given path

```
$('#uploadForm').submit(function (e) {
  e.preventDefault()
  $.ajax({
    type: 'POST',
    url: '/upload/',
    data: new window.FormData($(this)[0]),
    processData: false,
    contentType: false,
    success: function () {...}
  })
})
```

Listing 2: Prototype: Upload a file on the client javascript

Regarding to the server, the function that handles the upload of the request can be found in listing 3. It uses a library called co-busboy [6] to help with the upload of the files. We use a node.js stream [44] to put the contents of the file in the disk.

```
let parse = require('co-busboy')

router.post('/upload', function * (next) {
  // multipart upload

  var parts = parse(this)
  var part

  // Each part is just an entry in the form. Allow multiple files
  part = yield parts
  while (part) {
    var stream = fs.createWriteStream('./files/' + part.filename)
    if (part) part.pipe(stream)
    //
    // ...Here goes the part to create the torrent file
    //
    this.status = 200
    part = yield parts
  }
})
```

Listing 3: Prototype: Receive file and save on disk

3.1.3.2 Create torrent file

As mentioned before, we are using the package create-torrent to generate the torrent metadata for the file that is uploaded. The options that are needed to create the torrent file are specified in listing 4.

Aside from some metadata like *name* or *createdBy*, there are other properties of this torrent file

[6] co-busboy [43] is a wrapper of busboy to be used with Javascript generators. Busboy (https://www.npmjs.com/package/busboy) is a package that handles multipart POST requests, that are commonly used to upload files. Generator functions are a specific type of function in Javascript that creates iterables. See more at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators

that affect the behaviour of the downloads that are *announceList*, *private* and *urlList*.

AnnounceList is the url of the tracker used to discover other peers of the network. In our case, this will be a private tracker, but a tracker is not used in this prototype. *private* specifies that this torrent is a private one, this prevents other torrent clients to share information about this torrent with other trackers that are not the specified on this file. And the *urlList* is the webseed for this file, the most important property in this case as we want to prove that the file can be downloaded entirely and that the webseed works.

```
let downloadUrl = {
  protocol: 'http',
  hostname: 'localhost',
  port: 3000,
  pathname: path.join('/files/' + part.filename)
}

var opts = {
  name: part.filename,
  createdBy: 'TEST',
  creationDate: Date.now(),
  private: true,
  announceList: [['http://localhost:3000/tracker/']],
  urlList: url.format(downloadUrl)
}
```

Listing 4: Prototype: Options passed to the create-torrent function

After the torrent file is created, it is stored in memory in a variable to be returned to the client when it is requested.

3.1.3.3 Get torrent and download file

Now that the torrent file is created and its content stored into memory (it is not needed to store the contents in a file, as we can send them binary to the client), we can download the file. This involves three different parts of the code:

- Get the torrent file from the server, the one generated in the previous section.
- Start downloading with Webtorrent on the client.
- Serve the file from the server, as a static file.

The code for all these parts can be found in listing 5, that contains pieces of code from the server and the client to do the steps described above.

```
// server routes to get the torrent file and to download the file (this endpoint
↪  acts as webseed)
router.get('/torrent', function * (next) {
  this.body = torrentCache
})

router.get('/files/:file', range, function * (next) {
  this.body = fs.createReadStream(path.join('./files', this.params.file))
})
// client part instead of using jquery AJAX for the http request, we use
↪  stream-http, that works better when receiving binary data, that we convert as
↪  Buffer on the client to be compatible with Webtorrent
http.get('/torrent/', function (res) {
    var data = []
    res.on('data', function (chunk) {
      data.push(chunk) // Append Buffer object
    })

    res.on('end', function () {
      data = Buffer.concat(data)
      client.add(parseTorrent(torrent), function (torrent) {
        // the file has now been added to the torrent client
      })
    })
})
```

Listing 5: Prototype: Get torrent file from server

3.1.4 Retrospective and deviations

In this part of the project, we have achieved our goals in the timeline and budget specified in the plan.

No major problems have been found developing this section. Some small problems include the use of stream-http instead of the jquery ajax request, changed after not getting to work the binary download in bytes with the jquery request library.

Having proved that the technology we want to use seems to work for our objectives, the next step in the project is to design an architecture for the whole platform, and start implementing it.

3.2 Platform architecture

This section describes the overall design and architecture of the platform. It is defined as a task in Sprint 0 2.7.2.1.

3.2.1 Objective

The objective of this unit is to give a clear vision of the elements that form our platform and how are they connected to each other.

3.2.2 Infrastructure diagram

The diagram 3.1 has an overall design of the different components that will be part of the platform with the type of connections to other components.

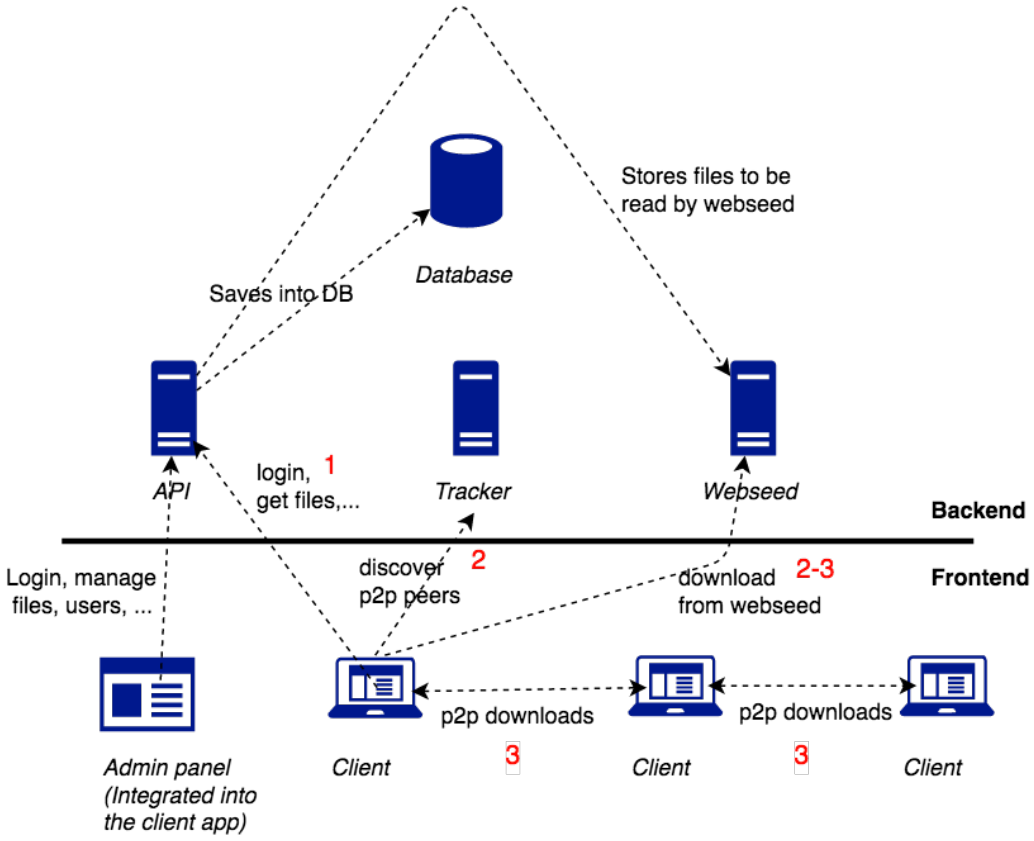

Figure 3.1: Infrastructure diagram

The diagram is split into **Frontend** and **Backend**, that differentiate the software that will run on the users computer from the one that runs in the platform's administrator servers.

Although in the tasks list both the admin panel and the downloads panel (See tasks of Minimum Viable Product in 2.5.1.1) are different elements, finally both have been merged as a single element that contains the admin functionalities behind the login of an administrator user. This has been done to simplify the architecture, leaving only one API and one client to be developed, instead of two clients and two API. The client and the API handle administrator rights.

On the frontend part, the diagram still separates the clients in admin panel and downloads client, to clarify the different connections that each part needs to do, but they are developed into the same application.

For the backend part, there are in total four elements. Tracker, webseed and Database have been described in Project scope (section 2.5.1.1), and the API is part of the admin panel and the downloads panel, now merged into one component.

3.2.2.1 Download process

The diagram also describes some of the connections that are needed when downloading a file, the numbers in red represent that process.

After the administrator has uploaded a file and has been stored to be served by the webseed, these are the steps done when downloading a file from the webtorrent client:

1. **Login and get files**: The first step for the user to start downloading is to login into the platform and get the list of available files. After choosing one file, the torrent metadata is sent by the API.
2. **Discover p2p peers and webseed download**: With the torrent metadata received from the previous steps, we have information about the tracker and the webseed. The download can start from the webseed, and, at the same time, connect to the tracker to notify about ourselves to the network of peers, and receive other peers that are connected to the network.
3. **p2p and webseed downloads**: After connecting with the tracker, if there are other peers connected the webtorrent client will start downloading pieces from them. The webseed process does not stop, as both types of downloads will be running for different pieces of the file.

3.2.3 Retrospective and deviations

In this unit there have been important deviations for the rest of the project from what was planned. As mentioned in the previous section, the design of the architecture has changed. Admin panel and downloads panel are now in the same application. This change implies we only need one API now and one frontend client.

The temporal planning to realize this unit has also changed. Instead of being done as part of the sprint 0 2.7.2.1, it has been done after all the other work.

During the development of the project, other things have been dynamically changing due to temporal restrictions or a change of motivations (in the features to do), so this diagram has been done after the project has been finished with the accurate representation of the finalized application. See more of the timeline in which the project has been done at 5.2.

3.3 Software models

This unit describes the different software models that the application will handle, and the relations between them. It has not been planned on any sprint, although it is part of the necessary documentation of the project.

3.3.1 Data models

These are the different data models involved in the application:

- **File**: The most important model for this project. It contains some metadata about the file like name, size, creation date. It also contains all the torrent metadata as a Buffer, and some encryption information needed to decrypt the file. More on the encryption on the security section 3.6.
- **Folder**: Inspired by the most common file systems, there are also folders where we can put the files on, for better organization. The folders form a tree of folders, where there is one root folder that can contain other folders or files.
- **User**: To be able to use the platform you have to be registered on it. For this purpose, there is a User model that will hold the email and password of login of each user, along other information of the user, like the name.
- **Invitation**: Instead of having an open registration to create users on the platform, the registration can only be done through an invitation from the administrator. This invitation contains the email that is able to register, and a specific token to be used on the moment of registration.

3.3.2 Diagram

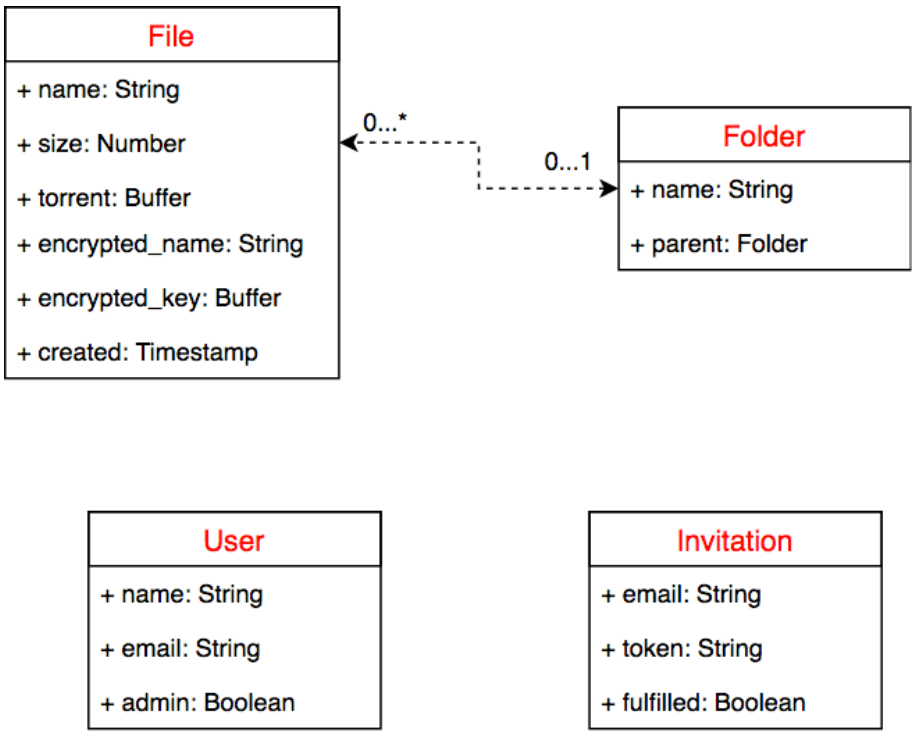

Figure 3.2: Data models

File and Folder are related in a *hasMany* relationship from Folder to File, as a Folder can contain 0 or more files. A file can pertain only to one Folder, or to zero, in which case it is considered to be in the root.

3.4 Backend

In this section we will go through the development of the Backend components. The complete code of the backend can be found at [45]. The tasks in the planning that belong to this section are:

- **Admin REST API and Downloads REST API**: Found at sprints 1 and 2 on Minimum Viable Product 2.7.2.2, are now merged into one single API.
- **Simple tracker**: Belonging to sprint 2 on 2.7.2.2. Remains unchanged and has been developed as planned.
- **Nginx configuration**: Belonging to sprint 1 on 2.7.2.2. This task has been changed, now the webseed is a node.js program instead of an nginx configuration.
- **Automated testing**: Found on Sprint 3 on 2.7.2.2.
- **Portability**: Most of the portability work is done by changing the way the backend is developed, taking into account variables that affect the different infrastructures. The portability is planned at Sprint 5 2.7.2.4

3.4.1 API

The API is a node.js server that accepts different combinations of HTTP verbs and paths (see 2.4.1), and returns a specific response for each of them. It will store and serve the information we need to manage the different entities of the platform (files, folders, users...).

3.4.1.1 Endpoints and functionalities

The following tables contain the different endpoints that the API has. The list is divided by the models described before in 3.3.1, and a special endpoint to authenticate users.

This is the list of columns:

- **Path**: The path where to reach a specific resource. Can be divided into different columns to reuse parts of the path that are common to different endpoints.

  When a part of the path it is preceeded by a colon (:), e.g. */users/:id/*, it means the path is not literally /users/id/, but a parameter to be replaced by an actual id (identifier), e.g. *507f1f77*. If followed by a *?*, means the parameter is optional.
- **Verb**: One of the verbs specified in 2.1.
- **Authenticated**: ✓if the endpoint requires authentication to be used, or ✗if it does not.
- **Admin**: ✓if the endpoint requires the administrator to work, ✗if it does not.
- **Same user**: Only in users table, it refers to resources where the user authenticated has to be the same that the one in the path or the administrator. E.g. if you need to access */users/5778*, you need to be authenticated as user with id 5778, or to be the administrator.
- **Description**: Brief explanation of the endpoint.

Path		Verb	Authenticated	Admin	Description
/api/files	/	POST	✓	✓	Upload file.
	/:id	GET	✓	✗	Get all the information of a file.
		PUT	✓	✓	Modify file. Only name can be modified.
		DEL	✓	✓	Delete file. It deletes metadata in the database, and the file in disk.

Table 3.1: Files endpoints

Path		Verb	Authenticated	Admin	Description
/api/folders	/	POST	✓	✓	Create new folder.
	/:id?	GET	✓	✗	Get folder with specified id. If not :id, the root folder is returned. Contains list of files and inner folders.
	/:id	PATCH	✓	✓	Modify folder. Only name can be modified.
		DEL	✓	✓	Delete folder. Recursively deletes all inner folders and files.
	/tree	GET	✓	✗	Get tree of folders from the root.

Table 3.2: Folders endpoints

Path		Verb	Authenticated	Admin	Same user	Description
/api/users	/	GET	✓	✓		Returns the list of users in the system
		POST	✓	✓		Creates a new user in the system as an administrator.
	/register	POST	✗	✗		Used to register a new user, without being an administrator.
	/password	PUT	✓	✗		Changes the password of the user currently authenticated.
	/:id	GET	✓	✗	✓	Get personal information about the user.
		PUT	✓	✓	✗	Modify user information.
		DEL	✓	✓	✗	Delete user.

Table 3.3: Users endpoints

Path		Verb	Authenticated	Admin	Description
/api/invitations	/	GET	✓	✓	Get invitations
		POST	✓	✓	Invites a new user to register and notifies by email.
	/:id/	DEL	✓	✓	Delete the specified invitation.
	/:id/resend	GET	✓	✓	Resends the notification email to register.

Table 3.4: Invitations endpoints

Path	Verb	Authenticated	Admin	Description
/api/authenticate	POST	✗	✗	Checks user and password sent. Returns token to use in *Authorization* header.

Table 3.5: Authenticate endpoints

3.4.1.2 Technologies used

Table 3.6 contains the most important technology choices made for this part of the project. Other small libraries and utilites will be mentioned in the Development 3.4.1.3 section. See 5.1 for the list of criteria considered when chosing a technology.

Programming language	Javascript. Has been chosen mainly for item 1 of criteria list, although it also complies with 2,3 and 5.
Runtime	Node.js, as Javascript runtime. It is the most used, and the one with the biggest community (See its repository [46]).
Web framework	Koa [42], chosen mainly for item 4, but complies with 2,5 and 6.
Database	MongoDB. There are various reasons to choose it: • Its document JSON data structure, it is easy to use along Javascript. • Has good support to store Buffer data, needed to store the torrent file binary data. • It is already known for the developer (Item 1).
Object document mapping (ODM)	Mongoose [47]. Used mainly because of criteria 1,2,3 and 5.

Table 3.6: Technology choices table

3.4.1.3 Development

This section has a brief description of how each component inside the API is programmed, explaining the software patterns that have been used, and how the technology is used to develop the different layers (From processing a request to write something on the database).

Koa middleware system

Koa (and other node.js frameworks) use the concept of *Middleware*. It is nothing else than a function that is executed for every request, or when it meets a certain criteria. Middleware can be stacked, creating an ordered list of functions to execute on a request.

A middleware function takes two parameters:

1. **context**: It contains all the information about the request, and is also used to build the response.
2. **next**: Function that calls all the following middlewares in the chain. If next is not called, then the following functions will not be executed.

See the example 6 of a middleware function in use, which creates a new Koa app and adds the request logger as first middleware. The requestLogger was taken from the Backend, and can be found in [48].

```
const app = new Koa(); // Create new koa application.

const requestLogger = async (ctx, next) => {
  const timestamp = new Date().getTime()
  let message =
    `REQUEST:: url: ${ctx.request.url} - method: ${ctx.request.method} `
  try {
    await next()

    // After next() is called, all the following middlewares have been executed,
    // and we can do something with the context, that has been modified by the other
    // middlewares (Like logging the status code sent)
  } finally {
    message += `RESPONSE:: status: ${ctx.response.status} `
    if (ctx.status >= 400 && ctx.status <= 500 && ctx.body) {
      message += `error: ${ctx.body.message} `
    }
    message += `TIME:: ${new Date().getTime() - timestamp}ms`
    log(message)
  }
}

// Later on, add a middleware to the application
app.use(requestLogger); // Use requestLogger as first middleware
```

Listing 6: Middleware function example

Routing

The routing system of Koa (separated in a different repo [49]) allows us to specify a set of middleware functions to execute for a certain HTTP path and verb, or all the verbs of a certain path, like in listing 6 using *app.use(middleware)*. The router can be a nested tree of routers, so for instance, we could define a router for the */api/* path, that is, at the same time, a combination of routers for each subpath, like */api/files* or */api/folders*.

This is the case in our project, where the router for each model is inside a global router for the whole application. The main router can be found at the path */api/modules/apiRouter.js* inside the repository [45], while the router for each model is inside its folder in */api/services/...*.

As an example, see listing 7, what configures the router for the */api/files* endpoint. See how the rights level (authenticated, admin) are configured as middlewares before the function that does the actual endpoint work, and they will throw an error if the rights are not met.

```
filesRouter.get('/:id', isAuthenticated, getFile)
filesRouter.del('/:id', isAuthenticated, isAdmin, deleteFile)
filesRouter.patch('/:id', isAuthenticated, isAdmin, updateFile)

filesRouter.post('/', isAuthenticated, isAdmin, uploadFiles)
```

Listing 7: Routing example of Koa applications

Authentication and Authorization

The authentication of the API is done via json web tokens [50]. The process to authenticate and consume protected resources is:

1. Browser sends a POST request to the server route to authenticate (*/api/authenticate* in our case), and if user and password correct, the server generates a token with user information by signing it with a secret. For the signature, the koa-jsonwebtokeb [51] package is used. See */api/services/authentication/middleware.js* in [45].
2. The token generated is sent to the browser along with other user information in the response. The browser is responsible of keeping this token to use it later.
3. Each time the browser needs to access a protected resource, it will send the token on the *Authorization* header of the request, preceeded by the type of token and a blank space, in this case "Bearer ".
4. When a request is done, the middleware of authentication verifies the signature of the token previously generated, with koa-jsonwebtokeb [51], and puts the information previously signed into the *context*. This way, we know in the following middlewares the user that has done the request.
5. When we have the information of the user, then we can check if it has the rights to access the resource its trying to access.

See listing 8 for the function handler of the authenticate endpoint. Errors and other non-important code has been removed for clarity. Listing 9 contains the middlewares *isAuthenticated, isAdmin and isSameUserOrAdmin*.

```
async function authenticate (ctx) {
  const {email, password} = ctx.request.body

  try {
    // Find user in DB
    const user = await User.findOne({email})
      .select('email password admin')
    if (!user) {
      throw new Error('UserDoesNotExist')
    }
    // Check password. See User model for implementation of this function
    if (await user.checkPassword(password)) {
      delete user.password
      // Information inside the token
      const token = {
        id: user._id,
        admin: user.admin
      }
      // Constructs the body of the response. This is what the client receives
      ctx.body = {
        token: sign(token, secret, { expiresIn: '7d' }),
        id: user._id,
        email: user.email,
        admin: user.admin
      }
    } else {
      throw new Error('IncorrectPassword')
    }
  } catch (e) {
    // ...error handling
  }
}
```

Listing 8: Authentication endpoint handler

```
// isAuthenticated middleware, using the same secret it was used to sign.
// It uses koa-jsonwebtoken module.
const jwt = require('koa-jsonwebtoken').default;
isAuthenticated: jwt({ secret, extractToken: koaJwt.fromAuthorizationHeader })

async function isAdmin (ctx, next) {
    if (ctx.state.user.admin) await next()
    else ctx.throw(401, 'You need to be admin to view this resource')
}

async function isSameUserOrAdmin (ctx, next) {
    if (ctx.state.user.admin || (ctx.state.user.id === ctx.params.id)) {
        await next()
    } else {
        ctx.throw(401, 'You cannot access this resource')
    }
}
```

Listing 9: Middlewares for authentication and authorization

Model-View-Controller architecture

Except authentication endpoint, the handlers of the other endpoints have a specific architecture. This handler is usually the latest function in the chain of middlewares of an endpoint, after the authentication and authorization middlewares, and is the one responsible of building the response for the client.

To build such responses, a Model-View-Controller pattern is used. These are the three layers of this pattern and how they have been used in this project:

- **Model**: The model is the part of code that directly interacts with the data. In this case, the code that connects to the database and implements the models defined in 3.3.1 with the Mongoose (see 3.6). It is usually a definition of properties of the data model, like in listing 10, the File model. In other cases, like User [52], it can contain functions like *checkPassword*.
- **View**: The view is the layer that constructs the response in the HTTP way, is responsible of choosing status code and formats the data (if needed) before appending it to the response body. Listing 11 contains an example of view function, from *GET api/files/:id* endpoint.
- **Controller**: It joins both of the parts, view and model, together. It usually handles the logic of joining models together and propagate other side effects. Listing 12 is the controller function used by the previous view to get the data.

In the project file structure [45], models are inside each of the services by the name *model.js*. The view functions are inside the *middleware.js* file of each service. This is because the view is the function that connects directly to Koa and its routing and middleware architectures. The controller functions are inside *controller.js* on each service.

And this covers the main development strategy of the API. Other sections like Security 3.6 and Portability 3.7 will explain other aspects of the backend in more detail.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const fileSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  size: {
    type: Number,
    required: true
  },
  torrent: {
    type: Buffer,
    required: true
  },
  folder: {
    type: Schema.Types.ObjectId,
    ref: 'Folder'
  },
  created: {
    type: Date,
    default: Date.now
  },
  encrypted_name: {
    type: String,
    required: true
  },
  encryption_key: {
    type: Buffer,
    required: true
  }
})

const File = mongoose.model('File', fileSchema)
module.exports = File
```

Listing 10: File model implementation

```
async function getFile (ctx, next) {
  try {
    const file = await fileController.getFile(ctx.params.id, FILE_SCREEN)
    if (!file) throw new Error('FileNotFound')
    ctx.body = file
  } catch (e) {
    log(e)
    if (e.name === 'CastError') {
      ctx.throw(400, 'Invalid ID')
    } else if (e.message === 'FileNotFound') {
      ctx.throw(404, E.FILE_NOT_FOUND_ERROR(ctx.params.id))
    } else ctx.throw(500, 'Unknown error')
  }
}
```

Listing 11: getFile view function implementation

```
async function getFile (_id, fields = '_id name size torrent encryption_key') {
  return await File.findOne({ _id }).select(fields)
}
```

Listing 12: File controller implementation

3.4.2 Tracker

The tracker is a node.js server too. It uses the module bittorrent-tracker[34], which already contains all the development of the tracker. In this case the only thing needed was to configure and deploy it. The configuration contains options like the type of server supported, like UDP, HTTP or WebSockets.

The filter function, called with the infohash of the current torrent request, can filter the torrents that the tracker operates. Ideally, this would filter by the torrents belonging to our platform, but for now it is a normal tracker that operates on any torrent. Listing 13 contains the actual config of the tracker server, and the whole can be found in the tracker folder of the backend repository [45].

```
const Server = require('bittorrent-tracker').Server
const debug = require('debug')('arxivum:tracker')

const server = new Server({
  udp: true, // enable udp server? [default=true]
  http: true, // enable http server? [default=true]
  ws: true, // enable websocket server? [default=true]
  stats: true, // enable web-based statistics? [default=true]
  filter: function (infoHash, params, cb) {
    debug('tracker received request for infoHash', infoHash)
    cb(true)
  }
})

server.listen(PORT) // Starts the tracker on port PORT
```

Listing 13: Tracker initial configuration

3.4.3 Webseed

The webseed is an HTTP server that serves the files that have been uploaded to the server.

This project contains a simple implementation of this webseed server in Node.js with Koa, but other web server implementations, like Nginx [53] or Apache HTTP Server [54], should work too if they have the correct configuration. It has to be able to accept and process Range header requests. The range header specifies a range of bytes of the resource to retrieve.

Listing 14 has the implementation of webseed server used in this project.

```
koaRouter.get('/:file',
async (ctx, next) => {
  ctx.set('Cache-Control', 'no-cache, no-store')
  ctx.set('Pragma', 'no-cache')
  var range = ctx.header.range

  if (!range) {
    ctx.status = 416
    return
  }
  await next()
},
range, async (ctx, next) => {
  ctx.body = fs.createReadStream(path.join(WEBSEED_ROOT, ctx.params.file))
})
```

Listing 14: Webseed example implementation

3.4.4 Automated testing

The API and webseed have an automated testing suite to ensure its correct functionality. The type of testing chosen for the backend is API endpoint testing. This type of testing initializes the server and does API requests like any other client would, to test that the responses are the ones expected.

The code for the testing suite can be found at */test/* folder in [45]

3.4.4.1 Testing tools

The tests are written in Javascript and executed on Node.js too, like the API. But it uses some libraries specific to writting tests:

- **Mocha**: Testing framework that runs on Node.js and the browser. See 3.4.4.2 for how Mocha is used.
- **Chai**: Assertion library, to check the received results with the expected ones.
- **Supertest**: Library to execute requests to a Node.js server, like an HTTP client.

3.4.4.2 Test development

When writing mocha tests, there are some reserved words that we can use to configure how the code of the tests is going to be executed.

This is a list of the reserved words:

- **describe**: It defines a group of tests, joining them together semantically. Accepts a callback function as argument, and the individual tests must be defined inside that block. **describe**s can be nested, having one inside another.
- **it**: Defines an individual test. Accepts a message as first argument, and a callback function that contains the necessary expectations to run the test.
- **before**: Specifies code to be executed before executing the tests in the same describe the *before* is defined.
- **beforeEach**: Specifies code to be executed before each of the tests in the current describe.
- **after**: Code to be executed after all the tests in the current describe (and all inner describes) have been executed.
- **afterEach**: Code that is executed after each of the tests (or describe block) has been executed.

Listing 15 contains an example from the test suite, formatted for clarity.

The **before** clause configures some enviromental variables for the tests (port numbers, debug, and ore), it drops the database to start clean, and initialized database and server to prepare it to receive requests.

In the **after** claure, the DB is disconnected and the server process is killed. In almost all the other test files this is the procedure, with small variations (When the webseed is also started, there are 2 server apps instead of one, for instance).

The test shown tests that the admin user is already initialized in the application after we initialize it, without needing to manually create it.

```
const expect = require('chai').expect
const utils = require('./utils')
const req = require('supertest')
const killable = require('killable')
const http = require('http')

describe('Authentication and authorization tests', function () {
  let server
  before(async function () {
    this.timeout(5000)
    utils.configTest()
    await utils.dropDB()
    utils.configDB()

    const app = require('../api/app')
    const db = require('../api/modules/database')
    await db.connect()

    server = http.createServer(app.callback()).listen(process.env.API_PORT)
    server = killable(server)
    // Let server create the admin user and password
    await utils.tick(1500)
  })

  after(done => {
    server.kill(() => {
      utils.disconnectDb()
      done()
    })
  })

  it('When initialized, I can directly authenticate with admin user', async () => {
    const res = await req(server)
      .post('/api/authenticate')
      .send({ email: 'admin@admin', password: 'admin' })

    expect(res.status).to.equals(200)
    expect(res.body).to.have.property('token')
    expect(res.body).to.have.property('admin', true)
  })

  // ... more tests on the original file
})
```

Listing 15: Testing example from auth.spec.js

3.4.5 Retrospective and deviations

There have been some deviations from the original plan on the backend. As mentioned earlier in 3.1, there is only one backend (and one platform) instead of two (Admin and Downloads).

Also, the MVP 2.7.2.2 on Sprint 1 contains an nginx that will host both the files of the deployed

frontend, and the webseed. The nginx configuration for the frontend files has been done inside Docker, see 3.8 for more. And the webseed is not an nginx server, but instead has been coded manually as example.

Something that is not mentioned at all in the planning is that the files are encrypted on the backend when saved. See 3.6 for a description of the security decisions taken around the platform.

Finally, the advanced tracker task, mentioned on Sprint 3 of 2.7.2.2 has not been done, and the tracker remains for now a simple tracker. This is something that could be improved.

In terms of timeline, it has been developed during several months over the last year. See 5.2 for more information about the timeline. The change in costs is almost impossible to calculate, as the hours spent on this backend have not been tracked.

3.5 Frontend

This section describes the Frontend of the platform. The code of this unit can be found at Github [55].

The number of tasks from the project planning that correspond to this section are:

- **Admin Frontend and Downloads Frontend**: Found on sprints 1 and 2 of 2.7.2.2, are now a single frontend. This task includes screens design and basic functionalities.
- **Downloads with webtorrent**: Found on sprint 2 of 2.7.2.2, integrates the webtorrent client with the frontend.
- **Progress module**: Also on Sprint 2, it would be on this section, although the progress is finally not a separate module but part of the same frontend project.

There are other functionalities that have been added to the project after it was planned.

3.5.1 Screens

Table 3.8 contains the list of screens that will be needed to cover all the functionalities, with the level of authentication needed to access them, and its url.

Screen	URL	Anonymous	Authenticated	Admin	Description
Login	/login/	✓	✗	✗	Screen with login form.
Register	/register/	✓	✗	✗	Screen with register form.
Folders and files list	/files/list/	✗	✓	✓	List to navigate through the folders tree and files of each folder.
Downloads	/files/downloads	✗	✓	✓	List of currently downloading files.
Uploads	/files/uploads	✗	✗	✓	List of currently uploading files. Only admin can upload.
User info	/userinfo/	✗	✓	✓	Screen with user information. Also has change password form.
Player	/player/	✗	✓	✓	Screen with the player in case we are streaming a media file.
Users	/users/registered	✗	✗	✓	List of registered users. Needs admin.
Invitations	/users/invitations	✗	✗	✓	List of sent invitations. Needs admin.

Table 3.7: Frontend screens table

3.5.2 Technologies used

Table 3.8 contains the technologies used on the Frontend, using the criteria defined in 5.1.

Programming language	Typescript [56]. Has been chosen for reasons 2 to 6. Also, as it compiles to Javascript, it can be used in the browser.
Runtime	Client's browser. It only supports Javascript language. [46]).
Frontend framework	Angular [57], complies with elements 2 to 6 of the list of factors.
Resource bundler	Webpack [58]. Used to join together the different files of the project for an efficient delivery to the client. It also compiles Typescript to Javascript during the process.

Table 3.8: Technology choices table

3.5.3 Development

This section describes the overall architecture and development process followed to create the frontend of the platform. Before going with the actual code there are some concepts that need to

be explained as they are used on all the frontend: the Observable pattern 3.5.3 and the Angular architecture 3.5.3.

Angular architecture introduction

This section contains a small description of how Angular works. Angular [57] is written in Typescript, and enforces the use of typescript to develop frontend applications.

In Angular, every entity is developed in a Typescript class, and is decorated with an Angular decorator that specifies the type of entity it is. Angular has a main file where we put all our entities together for bootstrapping before our application starts. This way, we can do Dependency Injection [59] between all the entities. See the example 17.

It has two main entities of operation: components and services, along other more specific entities.

Component

A component is an element that has its own HTML template and CSS styling, and has also a typescript class that handles its logic. It is useful for separation of concerns [60], as each different element of the webpage that has an independent logic will be developed inside its own Angular component. Each angular component has its own selector or tag[7].

Components are organized in a tree. There will be an *App* component, that has inside other components, and then until the *leafs* of the tree, that are components that only use primitive html tags and not other components.

Listing 16 and 17 are an example of how to declare a Component in Angular with Typescript and Decorators [61]. The HTML file is not a normal HTML file, but instead has some Angular keywords that link the HTML to our Typescript code. See the official documentation for Structural directives [62] for an explanation of the custom syntax used by Angular to dynamically modify the HTML. The *@Input* decorator is used to accept an attribute when used (In this case, path: <ax-folders-breadcrumb path='variableContainingPath'>)

```
<ul *ngIf='path'>
  <li *ngFor='let folder of path; let i = index' class='folders-breadcrumb-item'>
    <clr-icon *ngIf='i !== 0' shape='collapse right'></clr-icon>
    <a
      class='nav-link'
      *ngIf='i < path.length - 1'
      (click)='navigateTo(folder)'
      [class.pointer]='i < (path.length - 1)'
      [class.breadcrumb-link]='i < (path.length - 1)'>
      {{folder.name}}
    </a>
    <span *ngIf='i === path.length - 1'>{{folder.name}}</span>
  </li>
</ul>
```

Listing 16: Angular component example: HTML file

[7] Like an html tag. For instance, <my-app-component> could be a valid component selector

```
@Component({
  selector: 'ax-folders-breadcrumb',
  templateUrl: './folders-breadcrumb.component.html',
  styleUrls: ['./folders-breadcrumb.component.scss']
})
export class FoldersBreadcrumbComponent implements OnInit {
  @Input('path') path: Observable<Array<any>>;

  constructor(
    private router: Router
  ) { }

  ngOnInit() {}

  navigateTo (folder) {
    const args: any[] = ['/files/list'];
    if (folder._id) args.push({ id: folder._id});
    this.router.navigate(args);
  }
}
```

Listing 17: Angular component example: Typescript file

Service

Services use the decorator *Injectable*. They are normal classes that have no HTML or CSS related to it. The dependency injection system ensures to initialize them for us, so we can use the instance in memory directly. Like in listing 17, where the Router is injected in the constructor of the function, means that the Router class is an Injectable too.

A service can be something that holds specific logic for the application or the architecture, a class to connect to the API, or anything else that we need that is not directly connected to an HTML element.

Routes and router

In Angular, we can list a route (path and params of URL) to a specific Component, so each time we change the path the Angular will dinamically change the contents of the page. To specify where to locate the components, we use the *<router-outlet>* component. Nested routes have nested *<router-outlet>*s. Listing 18 contains the configuration of routes of the application, directly connected to table 3.7

We then use

```
this.router.navigate(['/userinfo'])
```

to navigate to any of the routes specified.

```
export const routes: Routes = [
  { path: 'login', component: LoginPageComponent, canActivate: [LoginPageGuard]},
  { path: 'register', component: RegisterPageComponent, canActivate: [LoginPageGuard]},
  { path: 'files', component: FilesPageComponent, canActivate: [FilesPageGuard], children: [
    {
      path: '', redirectTo: 'list', pathMatch: 'full'
    },
    {
      path: 'list', component: ListPageComponent
    },
    {
      path: 'downloads', component: DownloadsPageComponent
    },
    {
      path: 'uploads', component: UploadsPageComponent, canActivate: [UploadsPageGuard]
    }
  ]},
  { path: 'player', component: PlayerPageComponent, canActivate: [PlayerPageGuard] },
  { path: 'users', component: UsersPageComponent, canActivate: [], children: [
    {
      path: '', redirectTo: 'registered', pathMatch: 'full'
    },
    {
      path: 'registered', component: RegisteredPageComponent
    },
    {
      path: 'invitations', component: InvitationsPageComponent
    }
  ]},
  {
    path: 'userinfo', component: UpdateUserPageComponent, canActivate: [FilesPageGuard]
  },
  {
    path: '**',
    redirectTo: 'login',
    pathMatch: 'full'
  }
];
```

Listing 18: Routes configuration in Angular

Observable pattern

Angular adopts the Observable pattern [63] almost everywhere in the code. The typescript library used for this pattern is rxjs [64]. For instance, the changes to the parameters of the URL are an Observable of those changes, to which we can subscribe to react to those changes, like in listing 19.

Because observables are composable elements through different operations, we can get observables of another observables, like listing 20 that receives the changes in the uploading files list, and creates an observable of either *badge-success* or *badge-orange*, to use as class in the icon of uploading. In this case we use operation map. See observable docs [65] for a list and description of operations. Section 3.5.3 contains more about where this downloading list comes from.

```
this.route.params.subscribe(params => {
  this.filesPageService.goToFolder(params['id']);
});
```

Listing 19: RxJS Example: Router

```
uploadsBadgeColor$ = this.filesPageService.uploadingList$
    .map(uploading => uploading.progress)
    .map(progress => progress === 100 ? 'badge-success' : 'badge-orange');
```

Listing 20: RxJS Example: Badge class

NGRX patterns

Ngrx [66] is a set of libraries that enforce a specific state management solution for frontend applications. The pattern it enforces is similar to the Redux pattern [67]. The main elements of the ngrx pattern are:

- **Store**: The store has the current state of the application. This state is an Observable of JSON Object. The Observable emits a new value each time the Store changes. From the store, we can select pieces of the store as observables. For instance, if the Store object has inside a *currentFiles* property, we can get an observable of only the changes to the *currentFiles* property with

  `store.select(state => state.currentFiles)`

- **Actions**: Actions are just an object that contains a property *type*, and optionally can contain more data related to the action. An action could be *AddFile*, that could have *ADD_FILE* as type, and the new file. The actions are then dispatched to the reducers.

- **Reducer**: A reducer is a pure function [68], that is executed for every action that is dispatched and returns the new state of the store (or piece of store). Usually reducers operate only on a portion of the state, and are then combined into a single reducer that contains all the others.

- **Effects**: Because reduers are pure functions, side effects cannot happen inside them. We cannot send any HTTP request, save to localStorage, or anything that is considered a side effect. Effects allow us to react to an action with a side effect, and dispatch another action after the side effect.

Image 3.3 summarizes this workflow.

State management

This section describes how the NGRX pattern described before is used to handle the state management of the frontend application. First of all, let's start by describing the shape of our Store object (see code 21), with a brief description of each property. All the code regarding state management can be found at folder */src/app/core/* inside the frontend repository [55].

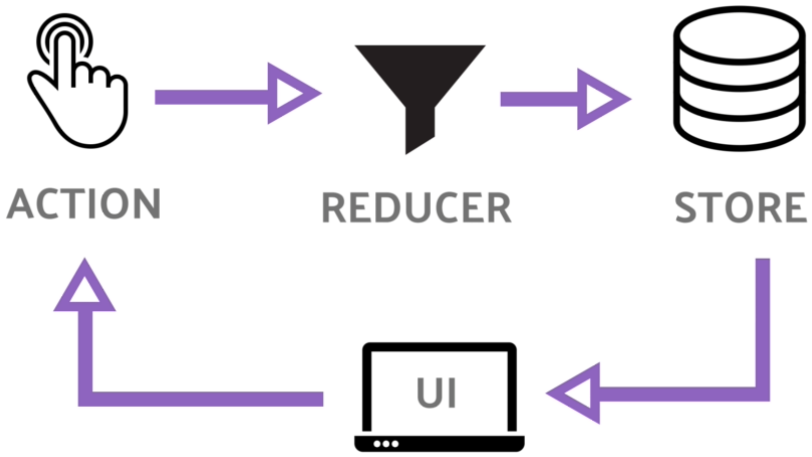

Figure 3.3: NGRX workflow, taken from https://angularfirebase.com/lessons/angular-ngrx-redux-starter-guide/

```
export interface AppState {
    // Authenticated state contains the current authenticated user. This property
    // is used to know if we are logged in or not, and to use its content to put
    // in the authentication header for each time we want to do a request.
    authenticated: AuthenticationState;
    // When we are navigating through the list of files and folders,
    // This property holds the current folder and file we are
    currentFolder: CurrentFolderState;
    // uploading and downloading contain the list of files that are currently
    // being uploaded and downloaded. It, although, does not contain the
    // percentages of upload or download on real time. This is specified
    // at uploadData and downloadData.
    uploading: UploaderState;
    downloading: DownloaderState;
    // When the admin is in the screen of users, this property holds the list
    // of users.
    users: UsersState;
    // Keeps real time progress for downloads and uploads
    // separated from the list itself.
    // It is more efficient when rendering the list and the percentages.
    // Here instead of a list, we have an object where keys are ID of file.
    downloadData: DownloadDataState;
    uploadData: UploadDataState;
    // Used to keep state of open modals on the whole application
    modals: ModalsState;
    // To use in the player, the state of the current file playing
    player: PlayerState;
    // List of invitations when admin is in this list
    invitations: InvitationsState;
    // Other common properties for the whole app, like app-wide notifications.
    common: CommonState;
};
```

Listing 21: Interface of Store object

Now let's take the example of currentFolder state. Listing 22 contains the actions that have been identified that modify our state somehow:

```
// The go to folder action is triggered when the route changes,
// and we want to go to a different route. There is a side effect
// that handles this action
export const GO_TO_FOLDER = '[Folders] Go to folder'
export class GoToFolder implements Action {
  readonly type = GO_TO_FOLDER;
  constructor(public id) {}
}

// This is the action that will directly modify the state of the store,
// putting the payload that contains the list in the store
export const UPDATE_FOLDER_LIST = '[Folders] Update folder list';
export class UpdateFolderList implements Action {
  readonly type = UPDATE_FOLDER_LIST;
  constructor(public payload) {}
}

// Reloads the current folder from the backend. Used after doing
// changes to the current list.
export const RELOAD_LIST = '[Folders] Reload list';
export class ReloadList implements Action {
  readonly type = RELOAD_LIST;
}
```

Listing 22: currentFolder actions

All these actions are used either by the reducer to modify the current state of the store, or by side effects to trigger some action and then return a different action. Listing 23 and listing 24 contain the code for the reducer and the side effects of the currentFolder.

The reducer operates mainly in the *UpdateFolderList* action, that is the one that contains the new content of the folder, either if we came here initially, by navigating, or by refreshing the current folder due to other changes (like changing folder names).

As shown in the side effects code, the *GoToFolder* action and *ReloadList* end up dispatching *UpdateFolderList* in the end, with its own operation to get the content (Actually, *ReloadList* dispatches a *GoToFolder* action with the current folder as parameter).

```
// Interface of currentState property of the store
// The path is used in the breadcrumb.
export interface CurrentFolderState {
  _id: string;
  name?: string;
  files?: IFile[];
  folders?: IFolder[];
  path?: IFolder[];
};

const initialState: CurrentFolderState = null;

// This function is called for each action dispatched to the store,
// it performs operations only on the actions that change currentState
// or returns the current state otherwise (no changes).
export function foldersReducer (state = initialState, action) {
  switch (action.type) {
    case FoldersActions.GO_TO_FOLDER:
      // Put the id
      return {_id: action.id }
    case FoldersActions.UPDATE_FOLDER_LIST:
      const {_id, name, files, folders, ancestors} = action.payload;

      const path = ancestors ? [...ancestors] : [];

      if (_id) {
        path.push({_id: _id, name: name});
      }

      // Add arxivum as first item in path
      path.unshift({name: 'Arxivum'});

      return {files, folders, path, _id, name};
  }
  return state;
}
```

Listing 23: currentFolder reducer

```
@Injectable()
export class FoldersEffects {
  private currentFolder$ = this.store.select(state => state.currentFolder);

  @Effect()
  reloadList$ = this.actions$
    .ofType(FoldersActions.RELOAD_LIST)
    .withLatestFrom(this.currentFolder$)
    .map(([_, currentFolder]) => new FoldersActions.GoToFolder(currentFolder._id))

  @Effect()
  goToFolder$ = this.actions$
    .ofType(FoldersActions.GO_TO_FOLDER)
    .switchMap((action: FoldersActions.GoToFolder) => this.foldersApi.getOne(action.id)
      .map(payload => new FoldersActions.UpdateFolderList(payload))
      .catch(error => Observable.of(new AppError('Could not load folder')))
    );

  constructor(
    private actions$: Actions,
    private store: Store<AppState>,
    private foldersApi: FoldersService
  ) {}
}
```

Listing 24: currentFolder side effects

Connecting state to components

With all our state management completed, there is only one thing left that is to connect this to our HTML. Imagine we have an Observable of current folder, declared like

```
currentFolder = this.store.select(state => state.currentFolder)
```

then we can iterate over the list of files to display it with

```
*ngFor='let folder of (currentFolder  async)?.folders'
```

This way, anytime the list changes, the HTML will be automatically updated to match the new state of the store. This is how the UI can react to whatever changes happen on the store by side effects or user interactions, without to manually propagate these changes on to the UI level. See [62] for the different ways to link HTML and Typescript code.

3.5.4 Retrospective and deviations

The case of the frontend in terms of deviations from the plan is similar to the backend. In contrast to what was planned in 2.7.2.2, there is only one frontend application, that contains a private admin section to administrate the platform, instead of having two different applications.

The tasks specified in Sprint 2 of 2.7.2.2, except of one, have been all realized. The task that is not done is the *Progress module*. It was planned that this part was a separate module, but has been finally integrated into the app itself, after being considered not a necessity to do it as a separate module. Furthermore, there are other tasks not specified in the planning that have been done. These are:

- **Streaming**: It has been added the possibility to stream media files (video, audio) or display images on the application itself, while it is being downloaded, without needing the complete file or having to download it to the computer at all.
- **Crypto**: Because of the streaming, it was needed to create a library to partially decrypt AES-CBC encryption on the fly, to be able to stream from partial file downloads that were encrypted. More about this will be explained at 3.6.

Like the backend, it has been developed during several months over the last year. See 5.2 for more information about the timeline. The change in costs is almost impossible to calculate, as the hours spent on this frontend have not been tracked.

3.6 Cryptography

This section covers some cryptography and security decissions that have been done during the development of the project. It covers how the security and cryptographic protocols have been used and implemented, why it was needed to implement them, and the payoffs that originate.

It is important to mention that this project was never planned as a complete cryptographically secure platform, and should not be considered like that. Instead, cryptoghraphy has been used to patch some concerning flaws that could be easily exploited.

3.6.1 File encryption

The files are encrypted when they are stored in the server, so the webseed can send them when there are no peers connected. The webseed is a public server, so anyone could ask the files we have stored in our platform. This is the reason behind encrypting the files in the backend, so the contents are not public to anyone arriving at the specific url of the file. This decission came after not figuring out how to add authentication headers to the webtorrent request that uses a webseed as a peer in the whole swarm created.

When the encryption is done in the Backend, the key to decrypt it and a random name are generated and stored in the database. The random name is used for the encrypted file on disk. This way, anyone with authorization access can ask for the key of the file to decrypt it.

The encryption is made with the algorithm AES [69] (Advanced Encryption Standard) in CBC mode [70] (Cipher Block Chaining). Listing 25 contains the lines of code that encrypt the file in the backend when it is received:

```
const qcrypto = require('crypto-promise')
const crypto = require('crypto')
const streamToPromise = require('stream-to-promise')
...
// Following code is inside the 'UploadFiles' controller.
/** Generate crypto properties */
const uuidName = `${uuid()}.enc`
let encryptKey
try {
    encryptKey = await qcrypto.randomBytes(256)
} catch (err) {
    throw new Error(E.GENERATE_CRYPTO_KEY_ERROR)
}

/** File model */
let model = {
    encrypted_name: uuidName,
    encryption_key: encryptKey,
    encrypted_file_path: path.resolve(WEBSEED_FOLDER, uuidName),
    file
}

/** encrypt and save to filesystem */
const encryptCipher =
crypto.createCipher(ENCRYPT_ALGO, model.encryption_key)

const writeFileStream = model.file
    .pipe(encryptCipher)
    .pipe(fs.createWriteStream(model.encrypted_file_path))
try {
    await streamToPromise(writeFileStream)
} catch (err) {
    log(err)
    throw new Error(E.ENCRYPTION_ERROR(model.file.filename))
}
```

Listing 25: Code that encrypts the file

3.6.2 Streaming encrypted files

One of the main challenges in this project has been to stream media files when they were coming encrypted from the webtorrent client (through the webseed or other peers). Webtorrent handles the downloads so the first pieces come first, in streaming order. Webtorrent gives access to an asynchronous stream of the contents of the file, encrypted. The streaming library needs a stream of the file, but decrypted. The solution developed here involves getting a stream of an encrypted file, and return an stream of the file decrypted, considering *start* and *end* positions.

How it works

When using AES-CBC-256 bits, the content is split in blocks of 16 byte length. Each block of data, is first XOR'ed with the Initialization Vector (if it's the first block), or the previous block

encrypted if its not the first block, and then this result is encrypted with the key. With this mode, the encryption cannot be parallelized and needs to be sequential, as each encrypted block is needed to XOR the contents of the following one. See image 3.4 for a visual representation of this process.

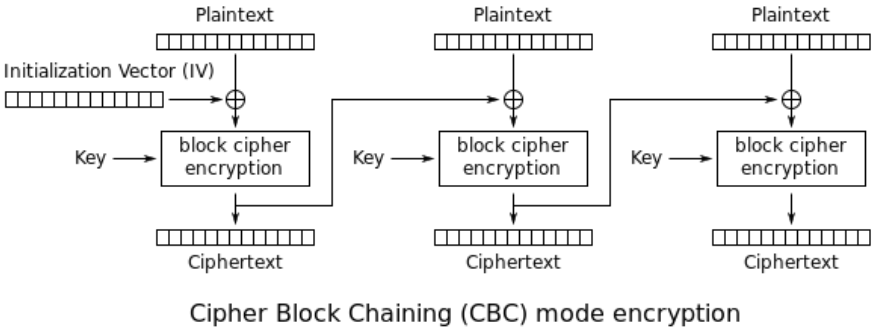

Figure 3.4: CBC mode encryption

And image 3.5 describes the process of decryption. As can be observed from the image, the only thing needed to decrypt a block in CBC is the key and the previous encrypted block (or IV if it's the first). So, for instance, to decrypt the third block, we would only need the encrypted second block and the decryption key to decrypt it. This way, we can achieve a partial decryption that is compatible with what is needed to stream the files.

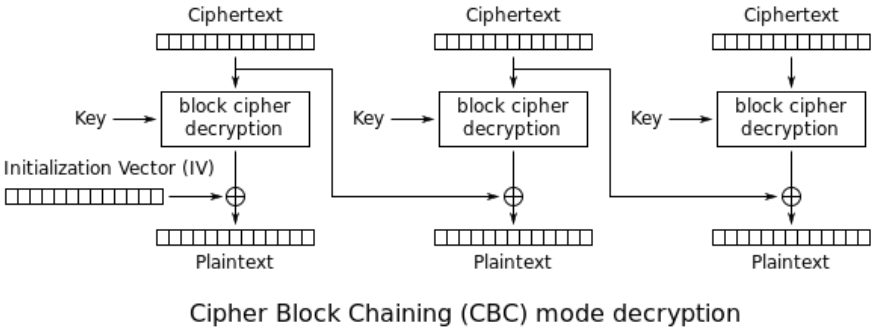

Figure 3.5: CBC mode decryption

** Both images taken from https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation)*

Repository

The code for the library developed on this chapter can be found at [71].

3.6.3 Decrypting files for download

The other situation is when we don't need to decrypt for streaming, but instead to decrypt the whole file to download it to the computer. In this case, we are doing a normal decryption, listing 26 shows

how. It is implemented as an NGRX Effect, triggered by the action of 'SAVE_FILE', dispatched when the user clicks download.

```
const DECRYPT_ALGO = 'aes-256-cbc';
...
@Effect()
saveFile$ = this.actions$
    .ofType(DownloaderActions.SAVE_FILE)
    .do(action => this.store.dispatch(new StartDecrypting((<SaveFile>action).file._id)))
    .switchMap(action => {
      const file = (<SaveFile>action).file;
      return this.filesService.getOne(file._id)
        .pluck('encryption_key')
        .switchMap(key => {
          const decipher = crypto.createDecipher(DECRYPT_ALGO, key);
          const stream = file.torrent_file.createReadStream();
          return Observable.fromPromise(streamToPromise(stream.pipe(decipher)));
        })
        .switchMap(decrypted => {
          return Observable.fromPromise(blobUtil.arrayBufferToBlob(decrypted))
        })
        .do(blob => {
          const link = document.createElement('a');
          link.target = '_blank';
          link.download = file.name;
          link.href = blobUtil.createObjectURL(blob);
          link.click();
        })
        .map(() => new FinishedDecrypting(file._id))
        // .catch(err => Observable.of(new ))
})
```

Listing 26: NGRX Effect that decrypts the file and open prompt for download

3.6.4 Potential flaws

There is, at least, one security concern about the files decryption, that mainly affects streaming. Because the file is being decrypted partially and *on the fly*, there isn't any hash verification that what has been downloaded is the right content. A hash verification would need to be done for the whole file, so while doing streaming, there is no such verification.

On the other hand, webtorrent already does a hash verification of each of the pieces it downloads, so we are partly covered against content tampering while downloading.

3.7 Portability

This section discusses how portability has been achieved, and how other users that decide to use the platform can have their own configuration for their own backend infrastructure.

This unit of the project was planned in 2.7.2.4.

3.7.1 Description of the problem

When using this platform, there are many different applications that depend on reaching each other in order to work. The backend needs to know where the webseed, database and tracker are from a public perspective, when creating the torrent file that the frontend will use to download the file. The same way, the frontend needs to know where the backend is (under which URL, protocol, etc).

3.7.2 Solution for Backend

In the backend, the solution was to make the variables that make up portability into a config file. This file is, by default (See listing 27), taking everything from environment variables from the operating system. This way, there are two ways to change and adapt the platform:

1. By modifying the config file corresponding to the environment we are configuring.
2. By setting up all the environment variables to match our deploy infrastructure, and let the default config file take them.

3.7.3 Solution for frontend

In the frontend, the solution is similar because it uses environment variables too, but has small differences. In the frontend, we actually don't have environment variables directly (as it is being executed in a browser), but because the FrontEnd code has to be built to get the Javascript code from Typescript, we can use environmental variables in the code, and in the moment of building, these variables will be replaced by whatever environment variables are in the machine building it.

The only problem we have with this solution is that the user has to build the Frontend before using it for its own environment. Ideally, the code would already be built, and then some configuration takes place to put these configuration variables.

3.7.4 Deviations

There have been some deviations with this unit. Specially regarding the timeline, although this unit was planned to be done after the deployment unit, it has been one before the deployment. This way, while developing the deployment unit, this same unit was being tested.

Regarding time spent and costs, it has not deviated much from what was planned, but it has been integrated in the programming of the backend itself, and not done as a standalone unit.

3.8 Deployment

This section describes the different options of deployment that the platform has, and also the preferred one that will be used for the demonstration of the platform.

This unit is planned in 2.7.2.3.

```
const config = {
  // URL of the front end
  PUBLIC_URL: env.PUBLIC_URL,

  // Database options
  DATABASE_URL: env.DATABASE_URL,

  // Initial admin email for the database.
  ADMIN_EMAIL: env.ADMIN_EMAIL,
  ADMIN_PASSWORD: env.ADMIN_PASSWORD,

  // Security options
  JWT_SECRET: env.JWT_SECRET,

  // Webseed path for the api
  WEBSEED_FOLDER: env.WEBSEED_FOLDER,

  // Webseed folder where to look for files, and API to put them.
  WEBSEED_ROOT: env.WEBSEED_ROOT || './files',

  // Public url where to access this api
  PUBLIC_API_URL: env.PUBLIC_API_URL,

  // Where the tracker is (WebSocket tracker)
  PUBLIC_TRACKER_URL: env.PUBLIC_TRACKER_URL,

  // Where the webseed is
  PUBLIC_WEBSEED_URL: env.PUBLIC_WEBSEED_URL,

  // Nodemailer options
  // For now only this ones are supported
  // https://nodemailer.com/smtp/well-known/
  // EMAILER_SMTP_SERVICE: env.EMAILER_SMTP_SERVICE,
  EMAILER_PORT: env.EMAILER_PORT,
  EMAILER_HOST: env.EMAILER_HOST,
  EMAILER_PUBLIC_EMAIL: env.EMAILER_PUBLIC_EMAIL,
  EMAILER_AUTH_USER: env.EMAILER_AUTH_USER,
  EMAILER_AUTH_PASSWORD: env.EMAILER_AUTH_PASSWORD,

  // Private PORTS
  API_PORT: env.API_PORT || 3000,
  TRACKER_PORT: env.TRACKER_PORT || 4000,
  WEBSEED_PORT: env.WEBSEED_PORT || 5000
}
```

Listing 27: Default config file for production

3.8.1 From source code

3.8.1.0.1 Backend

As many other open source projects, there is the option to just download the source code, play with it, and deploy it. To deploy from the source code, one needs to download the two main repositories

```
export const environment = {
  production: process.env.NODE_ENV === 'prod' ? true : false,
  api_url: process.env.API_URL || 'http://localhost:3000/api/',
  tracker_url: process.env.TRACKER_URL || 'http://localhost:4000/tracker/'
};
```

Listing 28: Environment variables in the Frontend

of the platform (arxivum-web [55] and arxivum-api[45]).

First, the backend needs to be deployed, in order for the frontend to have an API where to connect and login with the admin user. To deploy the backend from source code, you can either deploy the 3 modules separated, or you can use the code on *index.js* on the root folder of the backend. It is a small node.js code that creates a proxy for the other services under different public ports. To simply run this proxy, you do

```
node .
```

while inside the root folder of the backend.

In the *package.json* of the project there are other scripts to manually start each of the modules with the debugging messages:

```
npm run start-api, npm run start-tracker, npm run start-webseed
```

The API source code has two ways of configuration, based on configuration or environment variables, described in 3.7.2.

3.8.1.0.2 Frontend

Before deploying the frontend, it needs to be built. The building process of the FrontEnd generates the Javascript, html, css and other resources that are to be sent to the client. This differs with the actual code written more and more each year, with the incorporation of different languages (Typescript, Clojurescript, Coffescript,...) that compile and optimize Javascript. All the *compiled* resources are stored under the *dist* folder when building. To build the project, do in the root folder of the frontend:

```
npm run build
```

While building is when the environment variables are replaced by the actual values in the generated code. Look at 28 for the variables to replace to adapt it to any environment. Another option is to just replace the file with the actual values instead of reading from environment variables.

There is another command that is useful to build and deploy on the same command:

```
npm start
```

3.8.2 From docker images

When deploying from docker images, the first requiremet is to have Docker installed. When having docker, we run a docker container by running *docker run*. See the reference [72] for all the information on how to use this command. Listing 29 is an example of bash command to deploy the API in docker.

```
docker run -p 3000:80 \
    -v ./files:/app/files
    -env NODE_ENV=prod \
    -env PUBLIC_URL=https://arxivum.bertofer.me \
    # ...other envs
    -d albertoferbcn/arxivum-api
```

Listing 29: Command to deploy a docker container from an image

With this solution we can deploy each of the modules with just a command, without need to download the source code, or even knowing where the docker images are. The docker command line automatically will fetch the images needed for us, as they are uploaded to Docker hub.

Another solution to deploy docker images without running manually all the commands is to use a docker orchestration software [73]. This is probably the better solution for complex infrastructures where container orchestration for reliability and scalability is needed.

3.8.3 Configuring the stack in docker-compose

For the purpose of this project, docker-compose is going to be used to deploy the whole stack with a single command. Docker-compose is a tool to deploy multi-container applications. It allows us to configure the whole stack in a single file, that then can be deployed by docker-compose (to deploy in a single machine), or docker swarm [**docker-swam**] (multiple machines).

In our deployment stack, aside from the 4 modules we have for the platform, we are using 2 more modules inside their own container:

- **Proxy**: When using docker, no more than one container can be linked to the same port. This means we cannot have the different modules of the project listening on port 80 for HTTP (or 443 for HTTPS). This is a problem, as all of them use HTTP/S to retrieve the contents. The solution is to use a proxy, that is connected to ports 80 and 443 in the server, and proxies to the corresponding module. In this case, it knows where to proxy given the subdomain. For instance, *arxivum-api.bertofer.me* would redirect to the API, while *arxivum.bertofer.me* redirects to the Frontend.
- **Database**: There is configured a database too, for test purposes. In a real production environment, it is not the best solution for databases to be containerized.

Listing 30 shows the structure and contents of the *docker-compose.yml* file that is used to configure the whole stack. It is not in the repository, but instead in the server I am hosting the platform, because it is specific for my infrastructure, and not something related to the project itself.

```
version: "3"
services:
  nginx:
    image: albertoferbcn/nginx-proxy:latest
    ports:
      - 443:443
      - 80:80
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
      - /root/certs:/etc/nginx/certs
  mongo:
    image: mongo
    ports:
      - 27017:27017
    environment:
      - VIRTUAL_HOST=mongo.bertofer.me
  api:
    image: albertoferbcn/arxivum-api
    volumes:
      - ./files:/app/files
    links:
      - mongo
    environment:
      - VIRTUAL_HOST=arxivum-api.bertofer.me
      - NODE_ENV=prod
      - PUBLIC_URL=https://arxivum.bertofer.me
      - DATABASE_URL=mongodb://mongo/arxivum-prod
      - ADMIN_EMAIL=email@bertofer.me
      - ADMIN_PASSWORD=bertofer.me
      - JWT_SECRET=randomS3cret
      - WEBSEED_FOLDER=/app/files
      - PUBLIC_API_URL=https://arxivum-api.bertofer.me
      - PUBLIC_TRACKER_URL=https://arxivum-tracker.bertofer.me
      - PUBLIC_WEBSEED_URL=https://arxivum-webseed.bertofer.me
      - EMAILER_PORT=587
      - EMAILER_HOST=smtp.upc.edu
      - EMAILER_PUBLIC_EMAIL=alberto.fernandez.cubero@est.fib.upc.edu
      - EMAILER_AUTH_USER=alberto.fernandez.cubero
      - EMAILER_AUTH_PASSWORD=***********
      - DEBUG=arxivum:*
  tracker:
    image: albertoferbcn/arxivum-tracker
    environment:
      - VIRTUAL_HOST=arxivum-tracker.bertofer.me
      - DEBUG=arxivum:*
  webseed:
    image: albertoferbcn/arxivum-webseed
    volumes:
      - ./files:/app/files
    environment:
      - VIRTUAL_HOST=arxivum-webseed.bertofer.me
      - DEBUG=arxivum:*
  front:
    image: albertoferbcn/arxivum-front
    environment:
      - VIRTUAL_HOST=arxivum.bertofer.me
      - NODE_ENV=prod
      - API_URL=https://arxivum-api.bertofer.me/api
      - TRACKER_URL=https://arxivum-tracker.bertofer.me
```

Listing 30: Docker compose configuration file for the whole stack

With this file, we are able to use the command *docker-compose up* to launch all the services at the same time.

The VIRTUAL_HOST environment variable is what the proxy uses to redirect the different subdomains to the specific container.

3.8.4 Deployment security

When deploying this project in production, it is recommended to use HTTP/S for all communications, specially the connection with the api, because it sends email and passwords over the Internet.

For the purpose of showing this project and having a deployed version of it, I have my own HTTPS certificates for the **.bertofer.me* domains. This certificates are shared with the nginx proxy, through a docker volume, as can be seen in listing 30 line 10. The nginx proxy handles automatic redirect to HTTPS option when available.

3.9 Landing page

A landing page has also been done for visibility of the project, and to act as an entry point to it.

It's planning is defined in 2.7.2.5. The repository can be accessed in Github [74]. The webpage is hosted under the url *arxivum-landing.bertofer.me*.

3.10 Documentation

Both of the repositories (backend [45] and frontend [55]) have a small public documentation in the *README.me* that explains the technologies that have been used to develop the project, how to run tests if any, and how to deploy the project.

The Landing page also contains a small introduction to get started deploying the project.

Chapter 4

Conclusions

There is a need of decentralization of the current Internet. The servers of the bigger companies of the world cost millions and millions, while users have a huge amount of spare bandwith that they are already paying for. Decentralized solutions offer efficiency and engage people to participate in the network.

This project offers an already made platform to upload files and do file-sharing, using decentralized protocols, while keeping the files private and only accessible to those with access. But equally important, it serves as an inspiration to adapt other file sharing solutions to use decentralized downloads, heavily reducing infrastructure costs by using the users bandwith.

It has a strong focus on delivering a good user experience. From the users point of view, the frontend is well designed and uses new libraries with strong focus on design and accessibility. Being a single page application, there are no loads between different pages, and the whole platform works smoothly. From the administrators point of view, the platform has been designed to be easy to deploy while keeping flexibility of use, to adapt it to other infrastructures and deployment processes.

The platform is now working, but still lacks more support and more tests to be a solid solution. Being an open source project, it is important to engage other people and companies to participate, either by reporting issues, solving them, or with discussion and implementation of new features. This is the objective for the following months after delivering the first version of the platform.

Chapter 5

Appendix

5.1 Technology choice criteria

When choosing a technology or programming language there are various factors to take into account, that are summarized in the following list:

1. The technology is already known to the developer team? This speeds up the beginning of the development.
2. The language or framework offers everything you need in a simple way, without much effort from the developers.
3. There is an active community behind the technology? this ensures issues get solved quickly and it is constantly evolving.
4. When choosing frameworks or libraries, consider the features of the programming language they use (In the case of Javascript, does it use ECMAScript 2015?[75] Async functions?[76])
5. Is it open source? This makes easier to contribute, and is usually free to use.
6. Does it enforce good programming practices? This is, specially on the mid and long term, a very important consideration for maintainability of the project.

This list is mentioned through the project when technology choices have to be considered, mainly in the Frontend and Backend parts.

5.2 Follow up at 16 December 2018

This section has a sum up of the follow up report with the parts that are not directly into other sections of this thesis.

5.2.1 Introduction

The project was first started in autumn 2015, but will be finished long after, by winter 2018.

Before enrolling into the project, I was accepted to go on Erasmus in the second quarter of the 2015-2016 academic year. The city I was going to is quite expensive, so I looked for a job there, which I found in November 2015 (as full stack web developer). That was months before the initial delivery date of the project (January 2016). It turned out to be a quite stressful startup, we had very tight deadlines and my colleagues and I had to work extra hours to meet the expectations of the company. Because of this, the initial project was postponed to be delivered in April 2016, instead of January.

In January 2016, I started the university in the foreign country. I could manage to do some less hours at work, but handling three things at once (Foreign university, Work, Degree final project) was impossible. I had to choose and finally decided to let go of the degree final project.

In June 2016, I finished my exams in the university. I also quit my job, as the levels of stress were beyond what I was willing to accept. At that moment, I wanted to stay abroad to improve my English and get more professional and personal experience.

Found my second job in October 2016 (also as web developer), and when I was established in the new company I decided to start the project again. Because technologies and ideas had evolved, I decided to start from scratch the platform, instead of reusing what I did the first time. I had a full-time job and other personal things to care about, so I started developing it around November 2016 in order to have it finished to deliver in January 2018.

5.2.2 Current status

The project is almost finished. This is the list of project units:

1. Minimum viable product
2. Docker production environment
3. Portability
4. Project landing page
5. Webtorrent collaboration

Of those, the first three are already finished. The landing page is still pending, and will be done during the remaining weeks of the project, while the webtorrent collaboration won't be done, although other open source collaborations have been done along the realization of the project.

5.2.3 Work plan

The work plan, because of the explained at 5.2.1 has been changed. Some features were added while the project was still in development, mentioned below:

- It was not planned to encrypt files on the server, but was considered a security necessity if the webseed was going to be a public HTTP server serving the files. This way, if anyone acces those files, will find encrypted bytes.
- Streaming of the media files has also been achieved. For this a small utility [71] had to be written and has been made open source. It helps decrypt on real time while the file is being downloaded.

The timeline for the project has also been completely changed. The planning of sprints has not been followed. Instead, the project has been done on free time during one complete year, with very irregular intervals, almost impossible to track down.

Thus, the personal costs for the project are completelly changed. Other costs have not been affected. The work plan for the remaining 2 weeks is to finish the thesis, the landing page of the project and the documentation.

5.2.4 Methodology and rigour

The methodology has been changed. It was planned as Sprints split by weeks, but has not been followed or tracked that way. Instead, the project has been on development for almost a year, just focusing on the technology and solving the problems that appeared, no matter the timeline spent on it. After the project was almost finished, it has been then enrolled to be delivered.

5.3 Open source collaborations

During the development of this project, there have been some contributions to open source projects that were not planned, but necessary or interesting to do. This section has a brief listing of these contributions.

5.3.1 Chromium cache bug

While developing the streaming on the frontend, I found a weird behaviour with the way the range requests were being handled. After some research and debugging to get as close as possible to the bug, I opened an issue in the Chromium repository to notify the issue

5.3.1.1 Description of the problem

I first discovered something was wrong when the partial requests that the webtorrent package was doing, when, instead of receiving the byte range requested, the whole file was being sent, making the webtorrent package fail to process the file, as it was expecting a specific length of bytes that was not matched.

After debugging through the frontend and the backend, I realized, through the Chrome developer tools, that those requests that were failing didn't actually have the range header on it. Weird, because going throught the webtorrent code, the range request is always put on the code ([77]). Even the underlying library handling http requests (simple-get [78]) didn't show any sign of being the problem.

Next step was going to the *chrome://net-internals* page of Chrome, which shows a log of networking events. By tracking the requests, I figured out that, when trying to access cache for those range requests, the request to the cache had the range header, but when trying to retrieve the original content after not finding it in cache, the header was lost.

After this, I tried to disable the cache in the developer tools, and then everything worked as expected, which confirms that the problem was in how Chromium is handling the caching.

5.3.1.2 Submissions

After realizing where the source of the bug was, I opened an issue in the chromium bug reports system [79] explaining the problem. A small project reproducing the problem was also developed [80]

5.3.2 Koa range

The koa-range package enables a backend to properly handle range requests of files, but by the time I needed it, it did not have koa version 2 compatibility. I opened a pull request to add this compatibility [81].

5.3.3 Mongoose path tree

This package allows for tree relations in Mongoose schemas for MongoDB. The package was working for an older versin of mongoose, so I forked it and adapted the code to newer versions of some of its dependencies. The project still uses my fork instead of the original one.

This is the changes made to make it work [82]. It mostly changes the dependencies versions and refactors some lines.

Bibliography

[1] *Blizzard official page*. URL: https://www.blizzard.com/es-es/.

[2] *World of warcraft webpage*. URL: https://worldofwarcraft.com/es-es/.

[3] Vexis. *Forum comment abput dropping support*. URL: https://us.battle.net/forums/en/bnet/topic/16283439122#post-2.

[4] Google. *WebRTC Webpage*. URL: http://www.webrtc.org/.

[5] Netflix. *Netflix*. URL: https://www.netflix.com/es/.

[6] *Bittorrent*. URL: https://arstechnica.com/information-technology/2014/04/bittorrent-netflix-should-defeat-isps-by-switching-to-peer-to-peer/.

[7] *What are cryptocurrencies*. URL: https://blockgeeks.com/guides/what-is-cryptocurrency/.

[8] University of Southern California Information Sciences Institute. *Transmission Control Protocol*. Sept. 1981. URL: https://tools.ietf.org/html/rfc793.

[9] J. Postel. *User Datagram Protocol*. Aug. 1980. URL: https://www.ietf.org/rfc/rfc768.txt.

[10] Voxeo Cullen Jennings Cisco Anant Narayanan Mozilla (until November 2012) Adam Bergkvist Ericsson Daniel C. Burnett. *WebRTC 1.0: Real-time Communication Between Browsers*. Feb. 2015. URL: http://www.w3.org/TR/webrtc/.

[11] W3C. *World Wide Web Consortium*. URL: http://www.w3.org/.

[12] Google Inc. *YouTube*. URL: http://www.youtube.com.

[13] R. Peon Google Inc M. Thomson Ed. Mozilla M. Belshe BitGo. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. May 2015. URL: https://tools.ietf.org/html/rfc7540.

[14] A. Melnikov Isode Ltd. I. Fette Google Inc. *The WebSocket Protocol*. URL: http://datatracker.ietf.org/doc/rfc6455/?include_text=1.

[15] From Wikipedia. *Bram Cohen*. Retrieved December 2017. URL: https://en.wikipedia.org/wiki/Bram_Cohen.

[16] Bram Cohen. *The BitTorrent Protocol Specification*. Jan. 2008. URL: http://www.bittorrent.org/beps/bep_0003.html.

[17] David Harrison. *Index of BitTorrent Enhancement Proposals*. Jan. 2008. URL: http://www.bittorrent.org/beps/bep_0000.html.

[18] Feross Aboukhadijeh. *WebTorrent.io*. URL: http://www.github.com/feross/webtorrent.

[19] Alberto Fernandez. *Arxivum prototype*. URL: https://github.com/bertofer/arxivum-prototype/.

[20] Node Foundation. *Buffer Documentation*. URL: https://nodejs.org/api/buffer.html.

[21] MongoDB Inc. *MongoDB*. URL: https://www.mongodb.org/.

[22] MongoDB Node.JS Team. *Binary() Documentation*. URL: https://mongodb.github.io/node-mongodb-native/api-bson-generated/binary.html.

[23] Michael Burford. *WebSeed - HTTP/FTP Seeding (GetRight style)*. Feb. 2008. URL: http://www.bittorrent.org/beps/bep_0019.html.

[24] Github Inc. *Github*. URL: http://www.github.com.

[25] Node.JS Foundation. *Node.JS*. URL: https://nodejs.org/en/.

[26] Feross Aboukhadijeh. *Webtorrent FAQ*. Sept. 2015. URL: https://webtorrent.io/faq.

[27] Feross Aboukhadijeh. *Add WebRTC support to popular torrent clients*. 2015. URL: https://github.com/webtorrent/webtorrent/issues/369.

[28] Docker Inc. *Docker*. URL: https://www.docker.com/.

[29] Canonical LTD. *Linux Containers*. URL: https://linuxcontainers.org/.

[30] Docker Inc. *Docker Customers*. URL: https://www.docker.com/customers.

[31] Docker Inc. *Docker Source*. URL: https://github.com/docker/docker.

[32] Docker Inc. *Docker Hub*. URL: https://hub.docker.com/.

[33] Feross Aboukhadijeh. *Create Torrent Module*. URL: https://github.com/feross/create-torrent.

[34] Feross Aboukhadijeh. *BitTorrent Tracker Module*. URL: https://github.com/feross/bittorrent-tracker.

[35] Kent Beck Mike Beedle Arie van Bennekum Alistair Cockburn Ward Cunningham Martin Fowler James Grenning Jim Highsmith Andrew Hunt Ron Jeffries Jon Kern Brian Marick Robert C. Martin Steve Mellor Ken Schwaber Jeff Sutherland Dave Thomas. *The Agile Manifesto*. URL: http://www.agilemanifesto.org/.

[36] Ken Schwaber & Jeff Sutherland. *The Scrum Guide*. July 2013. URL: http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf#zoom=100.

[37] LLC Taiga Agile. *Taiga.Io*. URL: https://taiga.io/.

[38] Alberto Fernández. *Bertofer Github Profile*. URL: https://github.com/bertofer.

[39] Alberto Fernandez. *Arxivum prototype code*. URL: https://github.com/bertofer/arxivum-prototype.

[40] *Ajax*. URL: https://www.w3schools.com/xml/ajax_intro.asp.

[41] *https://jquery.com/*. URL: jQuery.

[42] *Koa - next generation web framework for node.js*. URL: http://koajs.com/.

[43] *co-busboy package*. URL: https://www.npmjs.com/package/co-busboy.

[44] *Node.js - Streams*. URL: https://nodejs.org/api/stream.html.

[45] Alberto Fernandez. *Arxivum Backend*. URL: https://github.com/bertofer/arxivum-api.

[46] NodeJS Foundation. *Node.js JavaScript runtime*. URL: https://github.com/nodejs/node.

[47] Automattic. *mongoose*. URL: http://mongoosejs.com/.

[48] Alberto Fernandez. *requestLogger.js file*. URL: https://github.com/bertofer/arxivum-api/blob/develop/api/middleware/requestLogger.js.

[49] Alex Mingoia. *koa-router*. URL: https://github.com/alexmingoia/koa-router.

[50] J. Bradley Ping Identity N. Sakimura NRI M.Jones Microsoft. *JSON Web Token (JWT)*. RFC. May 2015. URL: https://tools.ietf.org/html/rfc7519.

[51] Sid Jain. *koa-jsonwebtoken*. URL: https://github.com/f0rr0/koa-jsonwebtoken.

[52] Alberto Fernandez. *User model implementation*. URL: https://github.com/bertofer/arxivum-api/blob/develop/api/services/users/model.js.

[53] *Nginx webpage*. URL: http://nginx.org/.

[54] *Apache HTTP Server webpage*. URL: https://httpd.apache.org/.

[55] Alberto Fernandez. *Arxivum Frontend*. URL: https://github.com/bertofer/arxivum-web.

[56] *Typescript - JavaScript that scales*. URL: https://www.typescriptlang.org/.

[57] *Angular*. URL: http://angular.io/.

[58] *Webpack*. URL: https://webpack.js.org/.

[59] *What is DI?* URL: https://angular-2-training-book.rangle.io/handout/di/what_is_di.html.

[60] derekgreer. *The Art of separation of Concerns*. 2008. URL: http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/.

[61] *Decorators - Typescript*. URL: https://www.typescriptlang.org/docs/handbook/decorators.html.

[62] *Angular - Structural Directives*. URL: https://angular.io/guide/structural-directives.

[63] *Observer pattern*. URL: http://www.oodesign.com/observer-pattern.html.

[64] *RxJS API*. URL: http://reactivex.io/rxjs/.

[65] *RxJS Observable documentation*. URL: http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html.

[66] *Ngrx Repository*. URL: https://github.com/ngrx/platform.

[67] *Redux webpage*. URL: https://redux.js.org/.

[68] *Three principles of Redux*. URL: https://redux.js.org/docs/introduction/ThreePrinciples.html.

[69] *AES Development*. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development.

[70] *Block Ciphers Modes of Operation*. URL: http://www.crypto-it.net/eng/theory/modes-of-block-ciphers.html.

[71] *cbc-partial-decrypt*. URL: https://github.com/bertofer/cbc-partial-decrypt.

[72] *Docker run reference*. URL: https://docs.docker.com/engine/reference/run/.

[73] *The ultimate guide to docker orchestrators*. URL: https://www.twistlock.com/2017/05/22/container-orchestrators/.

[74] *Arxivum landing page repository*.

[75] ECMA International. *ECMASscript 2015 Language Specification*. URL: https://www.ecma-international.org/ecma-262/6.0/.

[76] Mozilla and individual contributors. *Async functions*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.

[77] *Webtorrent code - sending of Range request header*. URL: https://github.com/webtorrent/webtorrent/blob/master/lib/webconn.js#L124.

[78] *Simple-get repository*. URL: https://github.com/feross/simple-get.

[79] *Byte range requests sometimes omits range header if cache is enabled*. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=775014.

[80] *chromium-byte-range-bug repository*. URL: https://github.com/bertofer/chromium-byte-range-bug.

[81] *koa-range - Pull request 11 - Added compatibility with koa2*. URL: https://github.com/koajs/koa-range/pull/11.

[82] *Mongoose path tree fork - commit 7418785697e65f2ce1f86f3d10cb8263f6fa171d*. URL: https://github.com/bertofer/mongoose-path-tree/commit/7418785697e65f2ce1f86f3d10cb8263f6fa171d.