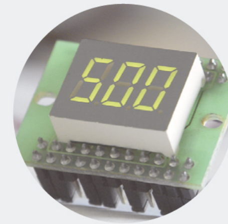
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

isep



Multimedia Content Distribution Management Using a Distributed Topology

PEDRO MIGUEL MOURA ZENHA
Setembro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Multimedia Content Distribution Management Using a Distributed Topology

Master in Electrical and Computers Engineering

Pedro Zenha

Guidance
Vitor Manuel Rodrigues da Cunha

Academic year: 2019-2020

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Eletrotécnica
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

Acknowledgements

First of all I have to thank Eng. Vitor Cunha for his time and dedication to this dissertation, his help was indispensable.

I also need to thank Nelson Novais for the opportunity and trust placed in me. And also to José Moreira for all the help and knowledge he gave me during this project.

Nonetheless, to my family and girlfriend for all the support.

Abstract

Advertising plays an important role in order for many companies to promote their products and services. It can be expensive to place advertisements with no guarantees that the message will reach the intended persons. In this field, targeted advertising is the mainstream strategy to captivate the potential consumer. People are used to see advertisements everywhere they go in many different forms. One of those is the use of screen displays that are believed to make the ads more engaging. However, using digital screens to advertise may lead to some issues, like down times or unwanted error messages from the device that controls the screens. This can cause a bad experience for both the target audience and the advertiser.

This thesis was developed within the scope of a project called Vixtape. It's a platform with the goal of turning any public screen into an ads displaying device and in the process reward the screen owner by exposing ads to the target audience. It also has the mission of giving the end user a optimal technological experience, no flaws and highly efficient. All these characteristics are accomplish by the use of a new open source technology called Interplanetary File System (IPFS), that allow devices to share content between them in a Peer-to-Peer (P2P) topology. This content distribution method saves Internet bandwidth to the end user (i.e., the Vixtape service client) and also enables the devices to work offline in case their Internet connection drops. This will greatly reduce the common problems seen with ads screen, thus giving a better experience to both the audience and the end user.

By the end of this document one can see that, adding a distributed topology to the Vixtape platform increased the Internet usage efficiency of the ads devices by always having up-to-date content available. This avoids that a device unnecessarily requests content from any of the other devices that had previously requested it. Additionally, a strategy to target a given audience was employed in order to chose the right ads to play. This further increases the maximum potential consumers the advertisements are shown to.

keywords: Target Advertisement, Digital Out Of Home, P2P, IPFS, Distributed, Decentralized, Web Development

Contents

Acknowledgments **iii**

Contents **i**

List of Figures **v**

List of Tables **vii**

Glossary **ix**

1 Introduction **1**
1.1 Contextualization 1
1.2 Problem definition and proposed solution 2
1.3 Thesis structure 5

2 State of the Art **7**
2.1 Historical background 7
2.2 Targeted advertisement and brand building 8
2.3 Digital Out Of Home 9
2.4 The problem with online advertisements 10
2.4.1 Online advertisements effectiveness 11
2.5 Current state of the centralized Internet 12
2.6 Decentralized Content Distribution Network overview 13
2.7 Distributed content distribution 15
2.7.1 BitTorrent 15
2.7.2 Blockchain 16
2.7.3 Hadoop 17
2.7.4 InterPlanetary File System 18
2.8 Discussion 20

3 InterPlanetary File System **21**

3.1 Architecture Overview . . . 21
3.2 Content addressing . . . 22
3.3 InterPlanetaty Name System . . . 23
3.4 Merkle DAG . . . 23
3.5 Version Control System . . . 26
3.6 Distributed Hash Tables . . . 27
3.6.1 Kademlia DHT . . . 27
3.6.2 Coral DSHT . . . 29
3.6.3 S/Kademlia DHT . . . 29
3.7 BitSwap protocol . . . 30
3.8 IPFS Cluster . . . 32
3.8.1 CRDT . . . 33
3.8.1.1 Operation based . . . 34
3.8.1.2 State based . . . 35
3.8.2 Raft consensus . . . 36
3.8.3 Consensus comparison . . . 37
3.8.4 IPFS Cluster Application Program Interface . . . 39
3.9 IPFS Application Program Interface . . . 39

4 Project Architecture and Technologies 43
4.1 System architecture . . . 43
4.2 System requirements . . . 45
4.3 Technologies . . . 48
4.3.1 Backend . . . 48
4.3.2 Frontend . . . 50
4.4 Integration of the IPFS resources . . . 51

5 Project Development 53
5.1 Broadcast plans . . . 53
5.2 Implementation of the IPFS in the players . . . 58
5.2.1 Integration of the IPFS in the browser player . . . 59
5.2.2 Integration of the IPFS in the Android player . . . 60
5.3 Configuration of a IPFS node . . . 62
5.4 IPFS Cluster implementation . . . 66
5.4.1 Cost efficiency of a IPFS cluster . . . 67
5.4.2 Integration of the IPFS Cluster in the players . . . 67

6 Tests and Results 71
6.1 Manual testing and simulation . . . 71
6.2 Unit Testing . . . 78

7 Conclusions 81

7.1 Future Work 82

References **83**

Appendix A **91**

Appendix B **97**

List of Figures

1.1 Media player closed in a advertising screen 3
1.2 Download error message in a advertising screen 4

2.1 Number of users using online ad blockers [1] 10
2.2 Typical structure of a CDN [2] 14
2.3 Visual representation of a centralized, descentralized and distributed network. 18

3.1 Overview of the IPFS stack 22
3.2 Base CID referencing to linked segments of a file 24
3.3 Data structure of linked files in a directory, based on the image [3] . . 26
3.4 The Kademlia binary tree as an example [4] 28
3.5 The process of locating a node by its ID [4]. 29
3.6 Example of how a file could be segmented into blocks 30
3.7 The process of transferring blocks of data (based on [5]) 31
3.8 Operation based example where the state propagates to all replicas . 35
3.9 State based example where the state changes are propagated to all replicas 36
3.10 Leader election algorithm 38

4.1 Architecture of the Vixtape system at the beginning of this project . . 44
4.2 The new Vixtape system architecture using the IPFS 45
4.3 Android boxes connected in a P2P network topology 46
4.4 UML use case of the interaction between the user, dashboard and API 47
4.5 Player logic to interact with the rest of the system's components . . . 48
4.6 Integration of the IPFS in the web browser player 51
4.7 Conversion of Go code to be used inside React applications 52

5.1 Response from the Vixtape API with a generated broadcast plan . . . 55
5.2 Generation of a broadcast plan with random creatives 55
5.3 Generation of a broadcast plan with the venue type algorithm 56

5.4 Relations between the database collections 57
5.5 Algorithm used by the web browser player to retrieve the needed content 60
5.6 Update procedure of the IPFS node configuration 66
5.7 Cluster peer configuration initialization workflow 69

6.1 Peers connected to each other after they startup 72
6.2 Accessing the content from the browser 77
6.3 New peer added on the **vixtape-cluster-network2** network 77

B.1 The player displaying a QR code for the user to bind it on his dashboard 97
B.2 The list of players owned by the end user 98
B.3 The first step of the binding process 98
B.4 The input token screen. The user should type in the generated token that can also be seen in Figure B.1 99
B.5 Setting up the player information, part I 99
B.6 Setting up the player information, part II 100
B.7 The user's campaign management page 100
B.8 The web browser player, playing a sample video 101
B.9 A Android box sample, connected to a display device 102

List of Tables

1.1 Project's calendarization 6

3.1 Comparison between CRDT and Raft 39
3.2 IPFS Cluster API endpoints list 40
3.3 The more relevant IPFS HTTP API endpoints 41

6.1 Testing the **PinPath** function 79
6.2 Testing the **HasFile** function 79
6.3 Testing the **UnpinPath** function 79

Glossary

Acronym	Description
API	*Application Programming Interface*
BP	*Broadcast Plan*
CDN	*Content Delivery Network*
CDN	*Content Distribution Networks*
CID	*Content Identifier*
CLI	*Command Line Interface*
CORS	*Cross-Origin Resource Sharing*
CPA	*Cost Per Action*
CPU	*Central Processing Unit*
CRDT	*Conflit-Free Replicated Data Types*
CTM	*Cost Per Thousand Impressions*
CTR	*Click-Through Rates*
CmRDT	*Commutative Replicated Data Type*
CvRDT	*Convergent Replicated Data Type*
DHT	*Distributed Hash Tables*
DNS	*Domain Name Server*
DOM	*Document Object Model*
DOOH	*Digital Out Of Home*
DSHT	*Coral Distributed Sloppy Hash Table*
FUSE	*Filesystem in Userspace*
HDFS	*Hadoop Distributed File System*
HTTP	*Hyper Text Transport Protocol*
IP	*Internet Protocol*
IPFS	*InterPlanetary File System*
IPLD	*InterPlanetary Linked Data*
IPNS	*InterPlanetary Name System*
mDNS	*Multicast Domain Name Server*
MFS	*Mutable Filesystem*
OOH	*Out of Home*
OS	*Operating System*
OTS	*Opportunity To See*

Acronym	Description
P2P	*Peer-to-Peer*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
RSA	*Rivest-Shamir-Adleman*
RTB	*Real Time Binding*
SEC	*Strong Eventual Consistency*
SQL	*Structured Query Language*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
UI	*User Interface*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
VCS	*Version Control System*
VM	*Virtual Machine*
VoD	*Video-on-Demand*
XOR	*Exclusive Or*

Chapter 1

Introduction

Advertising is part of our daily lives, in such a way that it can be looked at as being ubiquitous. Everyone has already came in contact with advertisements in many different forms and shapes, ones more creatives than others, digital or not, outdoor or indoor, but still, ads are now an inescapable reality. Besides the primary goal of promoting a company's product or service, the advertisements themselves can also be a source of revenue for 3rd parties. This fact was scaled up with the advent of online advertising, where almost every website uses ads displaying as a revenue source. But what if traditional brick and mortar businesses could also benefit from this revenue strategy? If it was possible to bring targeted and non invasive ads to local business like gyms, cafes and kiosks, this could create a new adverting service category with disruptive potential. This is the reasoning behind the Vixtape project and to which the works presented in this dissertation have made a significant contribution.

This chapter follows with the background information of how this project started in the first place, the description of the company behind it and where do the expected outputs fit in the advertising industry landscape. Moreover, it will be explained in detail the problem and the proposed methodology in order to achieve the desired goals.

1.1 Contextualization

The services provided by the advertising industry are very important for the business model of the majority of commercial brands. Without the use of advertisements we would never know that certain products or/and brands exist in the first place. There are also different ways to advertise a product or service, it may or may not make sense to advertise in the mass media, outdoor, indoor or on-

line. It greatly depends on several factors including the defined target audience. In this cases, profiles like the age group, genre, professional occupation, among many more, could be taken into consideration. It may make sense to advertise a promotion of diapers in TV late afternoon but not indoor of a college where the age group does not identify itself with the product. The same applies to almost everything companies and people need to advertise. In sum, advertisers need to take in consideration the group of people they are trying to reach, i.e., when and where they should advertise. This is the concept of target audience, show your ads where it really matters to reach only potential buyers for your product or service.

This is where the Vixtape [6] and RealtimeAds [7] projects come in. They started as two projects with the same goals that merged to be called Vixtape and the company behind it is called Mosano. This start-up company based in Matosinhos, Portugal, does technological consulting for various clients but also develops their own projects. As an example, Mosano developed the MPAC project which consists of a platform targeted for high-end multi-family complexes to control doors, gates and apartment locks with the use of Internet of Things technology and Machine Learning. Another one is a health care project called W-AI.DI. A virtual assistant designed to improve the treatment of diabetes, using Artificial Intelligence, that interacts in natural language to support doctors in evidence-based decision making.

The project where this thesis is integrated in is the Vixtape project that consists in a platform that can turn any screen into a digital signage. By connecting a Android box to any standard display screen and loading the Vixtape app, it starts to play ads based on the selected target audience, business model, and time-frame. This system can thus convert played advertisements into revenue for the screen owner. To improve the platform's adaptability and reliability it is envisioned that a distributed multimedia content management is a feasible approach. This way the ads would be shared across all Android devices without being sustained by a centralized content provider, that if unreachable, no more ads could be shown. Moreover, this approach addresses the most common problems found in currently installed advertising screens.

1.2 Problem definition and proposed solution

The Vixtape project aims to tackle several problems that are commonly seen in screens that play advertisements. For instance, to see blue or blank screens, closed media players like in Figure 1.1, error messages (Figure 1.2), stutters, buffering, the same content in a permanent loop, are some of the most common problems. Some of these systems use a pen drive to store the video ads playlist, but this method requires human intervention and lacks control over

what is being played at a given moment. Others use streaming, this allows to dynamically chose what content is being played and avoid repetition, however this method may face issues like stutters, buffering, bad image quality or blank screens. Downloading content in advance to the display schedule can solve a lot of the problems. But if this is done from a centralized source, in case the content provider stops working for some reason, there is no other option for the player to get new content. Typically in this situation the player will indefinitely loop the content it has locally stored. The Vixtape players development main goal is to always keep showing content even if there are some background problems happening at the moment, like the loss of internet connection or the unexpected malfunction of the content provider.

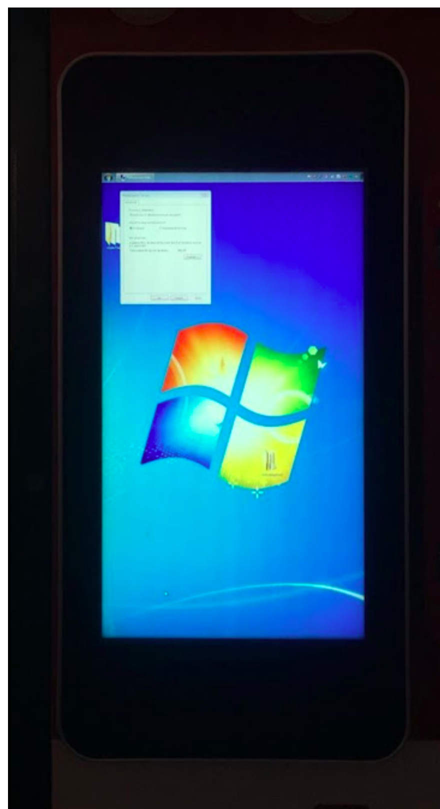


Figure 1.1: Media player closed in a advertising screen

The proposed solution to address these problems is to use, a much as possible, a private distributed Peer-to-Peer (P2P) network to disseminate the multimedia ads across all Android devices. To achieve this, the most promising framework is the InterPlanetary File System (IPFS) [8] as it will be discussed later. By using the IPFS technology it's assumed it will be possible, in case of Internet connection problems, to request new content to other devices in the private network cluster. Also in the case of low internet bandwidth, it will be highly

Figure 1.2: Download error message in a advertising screen

likely that the content that is going to be displayed is already in the device's memory. This also raises the need to develop tailor made novel complex advertisements scheduling algorithm. As the number of Vixtape devices grow into a worldwide private P2P network, the more content can be distributed among them, thus overall using less bandwidth to download content from the content provider server. This makes the Vixtape project highly scalable.

After analysis of the problems that this project aims to solve, and the proposed solution methodology, the workload can be divided in the following way:

1. Assess in detail the feasibility of the IPFS framework considering the project's goals
2. Implement a IPFS daemon in every displaying device
3. Develop the capability to remotely control the IPFS configuration of all devices
4. Understand the best possible cluster configuration to handle an undefined number of nodes inside the private network
5. Integrate the IPFS Cluster technology [9] in every IPFS node in order to control content storage and distribution
6. Configure a private network based on IPFS clusters

7. Program a scheduling algorithm capable of orchestrating the available content across the devices memory and in the cluster network

1.3 Thesis structure

The current chapter explains the context of this thesis, the problem that aims to solve and the proposed solution. The second chapter, the "State of the Art", will explain the different advertisement types and how they affect people's life. But the main focus is to survey the many network topologies and technologies that could support the intended content distribution method. The following chapter, entitled "InterPlanetary File System", takes a deeper look into how the IPFS works and its main advantages.

The fourth chapter explains the technological architecture of the project Vixtape. It also explains all the logic requirements in order for all the different components to work together. The project development chapter, which is the fifth one, exposes in more detail all the work done for this project and how the technologies introduced in the fourth chapter are used. The 6th chapter, "Tests and Results", will show the main project results and how they were validated. Finally, the last chapter addresses the conclusions taken from the development of this project and proposes some possible future developments.

Additionally, the Table 1.1 provides the calendar of the major workloads that went into this dissertation.

	February				**March**				**April**				**May**				**June**				**July**				**August**				**September**	
Week nb.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Project onboard	■	■																												
Study of the IPFS			■	■	■																									
Player and API developments					■	■	■	■																						
Integration of the IPFS with the API and the players									■	■	■	■	■	■	■															
Integration of the IPFS Cluster with the players																■	■	■	■	■	■	■								
Thesis document writing																							■	■	■	■	■	■	■	■

Table 1.1: Project's calendarization

Chapter 2

State of the Art

Nowadays advertising is seen by everyone in many different forms, some of them more efficient or pleasant than others. The way advertisements reach the potential consumer has changed over the years and it will not stop evolving any time soon.

2.1 Historical background

The evolution of advertising can be divided in four temporal periods. In the early civilizations of Egypt, Greece and Rome, outdoor advertising was used for slaves ads, household goods, tavern and many more. Then the industrial revolution between 1760 and 1830 resulted in the growth of the economy and mass transportation leading also to an increase of mass advertising. The period that goes from the middle of the 19th century to the beginning of the second world war has seen a surge in new clients, agencies and media. With the need for a vast amount of military equipment and uniforms, an increase in mass production and transportation emerged. Consequently, people were now able to buy goods not only from their local provision. Printed media was also more established and advertising was increasingly more cost effective. Nowadays, there is a fragmentation of agencies and media and sophisticated ways that marketeers use to expose advertisements to consumers.

The outdoor advertisement as a means of mass communication started in the Egyptian times, where they carved in huge basalt tables their laws, warnings and decrees. In the 1400s handbills and poster bills started to rise, and 200 years later, the outdoor sign appeared and spread throughout London streets. Between 1900 and 1912 the billboards were standardized in America, allowing nation wide advertisement campaigns of big companies like Coca-Cola, Kellogg

and Palmolive.

In the 18th and 19th centuries began to appear a synergy between newspapers and advertising, e.g., in 1704 the American journal "Boston Newsletter" published the first advertisement in a newspaper rewarding the capture of a criminal. Currently there are about 400 million persons worldwide buying newspapers daily. In the recent past some analysts predicted the demise of the newspaper due to the exponential growth of television and Internet, however, those predictions were not correct at all. The newspapers were able to adapt by creating new products, for example, free dailies, newspapers with a more compact form factor, new titles, better distribution and also by increasing their online presence.

With the introduction of television, the first advertisement was broadcasted in the United States on July 1st of 1941, where the Bulova Watch company paid 9 dollars for a 10 seconds ad before a baseball game between the Brooklyn Dodgers and the Philadelphia Phillies. But in those days it was the radio that lead the advertisement race due to the fact that most people owned a radio instead of a television. Additionally, the price to advertise on live television was far too expensive for some companies. Nowadays television is the biggest industry in terms of ad revenue, just in 2005 the total estimated income was 147 billion dollars.

The internet advertisement started to appear in 1994 with advertisers like AT&T, MCI, Sprint, Volvo and many more. Each year the online ads have grown, since there is no better place to use targeted advertising than on the Internet [10].

2.2 Targeted advertisement and brand building

In general the advertisements can be classified in two main categories: targeted and brand building. Brand building is normally seen has a means of mass communication, like TV, radio, magazines and newspapers, where the advertisement tends to be oriented by product/services/retail with the objective of establishing a good image and create demand for a product or service. The communication is typically one-to-many and designed to reach a mass audience, using an intrusion strategy intended to grab people's attention.

On the other hand, targeted advertisement is projected to help potential buyers to find captivating information based on their interests and needs. The communication is typically one-to-one and the main goal is for the potential buyers to identify themselves with the ads [11].

2.3 Digital Out Of Home

The expression Digital Out Of Home (DOOH) is simply an evolution of the (analog) Out of Home (OOH) expression, where the advertisements are static, offline and poorly directed. Normally this type of ad is placed in busy public places like shopping centers or transport stations, and it largely tends to follow the movement patterns of groups of people based on their daily routines [12]. While OOH has complex logistics, it's expensive and needs human effort to be visible, the DOOH through online digital advertisement can be a lot more creative and have a lot more visibility. This digital approach to advertising allows to monetize, modify, manage and display new content in a discreet way and in real-time.

But it's still unlikely that OOH will disappear in the foreseeable future. The first and most logical reason is due to the investment required to replace all billboards with digital screens, plus the screens get obsolete over time. Another reason are the required support infrastructures, since many locations don't have a power source or the network connectivity needed for digital screens, and most of the times the solution for these problems is way too expensive or difficult to solve. Moreover, some locations are regulated by legislation that don't allow digital outdoor screens. Nevertheless, OOH advertisement can also be creative just like most digital ads, the most common being the 3D posters that are still in demand among advertisers and agencies [13].

Traditionally the process of buying and selling advertising space was very time consuming due to the fact that everything was arranged manually. This process involved countless meetings, phone calls and e-mails, but nowadays this process can be much simpler and can take only a few milliseconds [14]. Someone who wants to place their ads in a DOOH network can do it using programmatic advertisement. This works through buying and selling ads in real-time, via a method called Real Time Binding (RTB). It consists in a real-time auction where advertising space buyers bid their offers to place their ads in a specific place and time frame, in order to reach a desired target audience [15]. A real-time bidding process can be done on open exchanges or private exchanges, which are invitation-only marketplaces.

An alternative to RTB is the so called "programmatic direct", which consists of directly selling advertising slots without a bidding process to buyers who have access to a programmatic network. Buyers can specify how many displays they want and in the process, get statistics like the number of impressions among other relevant metrics [16].

2.4 The problem with online advertisements

Nowadays agencies have more information about the consumers they want to reach, having access to huge stores of data that collect consumers habits. On the other hand many of those consumers, especially young people who are valued by advertisers, dislike online ads so much that they are paying to avoid them. The analyst Jay Pattisall said in an interview [17] "It's harder to reach audiences, the cost of marketing is going up, the number of channels has exponentially proliferated and the cost to cover all of those channels has proliferated. It's a continual pressure for marketeres - we're no longer just creating advertising campaigns three or four times a year and running them across a few networks and print". As advertisers bombard consumers across platforms like Twitch, Facebook, television, billboards and more, many consumers are trying to get away by singing up for ad blockers and subscription services. This trend can be observed from the chart in Figure 2.1.

Some start-ups have begun rewarding or compensating consumers to look at ads. However to effectively reach viewers, advertisers must also incorporate data-driven approaches like automation and machine learning technologies. It's expectable that this new approach will transform 80 percent of the advertising agencies jobs by 2030. For the first time Facebook, Google, Youtube and other online platforms are expected to be responsible for the majority of advertising according to WARC.

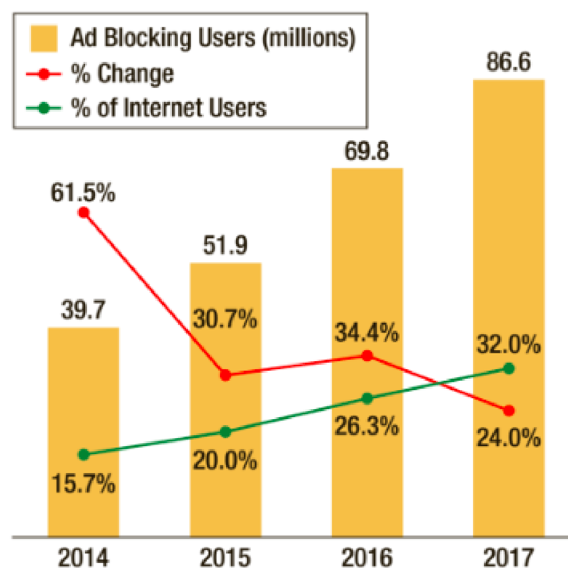


Figure 2.1: Number of users using online ad blockers [1]

2.4.1 Online advertisements effectiveness

Several companies offer measurement and analysis tools for online audiences, advertising, video, social media, and online behavior. Google Analytics and Ad Ratings are two of those companies. Most of these tools consist of a dashboard with the online viewers and unique audience visits, page views, time spent, loyalty, demographic information and consumer behavior. Online advertisers have the same measurement concerns as the conventional media: How many people clicked through a particular online ad? What actions were taken following click-troughs or site visits? How many visited a particular website? Is this form of online advertising yielding a suitable return on investment?

To actually measure the effectiveness of online ads there are specific units of measurement. A wide variety of metrics are used due to the diversity of advertisers specific goals and because there are many forms of online advertising. There are four general objectives to evaluate the effectiveness of online advertising and a variety of metrics that can be used to indicate whether the objective has been accomplished. Those objectives are:

1. The exposure value or popularity of a website or online ad, for example, the number of users exposed to an ad, number of unique visitors, and click-through rate.
2. The ability of a site to attract and hold users attention and the equality of costumer relationships, for example, the average time per visit, number of visits by unique visitors, and average interval between user visits.
3. The usefulness of websites, for example, the proportion of repeat visitors.
4. The ability to target users, for example, the profile of website visitors and visitors previous website search behavior.

The three most commonly used metrics are Click-Through Rates (CTR), Cost Per Thousand Impressions (CPM) and Cost Per Action (CPA).

The CTR represents the percentage of people, from the total that were exposed to an online ad, that actually clicked on it. This value has been decreasing over time especially for banner ads. Nevertheless, this type of ads still can have a positive effect on brand awareness even if the viewers do not click on them.

The CPM metric is an alternative to CTR, it estimates the cost to place an online ad on a per-thousand-impressions basis. The information given by CPM captures the online users Opportunity To See (OTS) an ad, but provides no information about the actual effect of an advertisement. Nowadays the CPM metric is starting to give way to CPA. The action that it's stated by this metric refers

to finding the number of users who actually visit a brand's website, register themselves or purchase anything on that advertised website. The majority of advertisers prefer to pay for online advertising on a CPA basis. The conditions for purchasing online advertising slots on CPA varies quite a lot, with higher prices being paid for actions involving actual purchases or actions closer to purchase. But since advertisers are interested in achieving specific results, such as increasing their sales, they are more willing to pay for metrics that show potential to achieve those results instead of paying for metrics like CTR, that gives less chances of achieving the desired results [1].

There is no such thing as the perfect metric to assess the effectiveness of online ads, however, some metrics may or may not be suitable for all advertisers. It is their duty to analyze which one better suits the company's business model.

2.5 Current state of the centralized Internet

The way Internet works is typically in a centralized way, this means that a web service is built around a server that handles all the requests and processing. When someone requests a web page using the HyperText Transfer Protocol (HTTP), usually that user types the domain in a browser that sends the GET request to that domain. Then the Domain Name System (DNS) will resolve the domain into an Internet Protocol (IP) address that correspond to the server's IP. After this operation the GET request is sent to the found IP, and then the server replies back with the content the user requested. In this case, if the server is offline or the content that is being requested is no longer in the same location, an HTTP error code is returned and the user does not receive what it he asked for.

Another big drawback is the lack of privacy and control over what is uploaded. For instance, when a user uploads a photograph to an online social platform, that photo is now owned by that platform and there is no guarantee if that photo will ever be deleted from their databases. Also the users of that platform need to trust on their security reliability, because it's not guaranteed that their photo will not going to be stolen by someone. The same applies to important files users upload to the cloud. In the event of a critical failure of the servers that store those files, the users probably will lose those important files forever.

The current state of Internet is inefficient. Consider two devices sharing the same network and one of them needs to download content from a server that is located 2000 km away from their geographic location. Consider also that later on the other device needs that same content. In this case, it will again request that content to the same server when the other device already has that content in memory. If the second device knew that the other had that content, it could request it directly to him saving time and internet bandwidth.

Given the present state of affairs, large scale content delivery is associated with a large distributed infrastructure which requires a big monetary investment. Content providers usually rely on third-party content distribution networks or build their own infrastructure to deal with the demand and maintain their quality of service. Global Internet traffic is increasing exponentially, being one of the reasons the Video-on-Demand (VoD) service which is expected to double by 2021 and constitutes around 70 percent of the internet traffic [18]. Although content providers are still able to manage the delivery of VoD traffic through their own servers, and do content caching during off-peak hours, they either rely on a third-party Content Distribution Network (CDN) or build their own content delivery infrastructure in order to continue providing quality of service to user applications during peak times. A CDN typically deploys a huge number of distributed cache servers so the content can be available at many locations, and they can also dynamically resolve content requests to the appropriate cache server. Moreover, Internet providers benefit from decreasing transit costs resulting from reduced peak time traffic, but they struggle to cope with sudden traffic shifts caused by the dynamic server selection algorithm of the CDN [19].

2.6 Decentralized Content Distribution Network overview

The Internet evolution brought challenges in managing and delivering content to users. One problem related to content distribution is when a web application receives too much traffic that clogs up the provided services. To deal with this issue Content Delivery Network (CDN) were designed to keep up with the increasing demand that the Internet still shows. A CDN is a collaborative collection of Web objects, where content is replicated across several web servers, that are scattered around the world in order to deliver that content to the end users [2].

Content delivery over the Web, especially video content, needs to be efficient. A CDN has to maximize bandwidth, increase accessibility and manage content replication. To do so, CDNs use the so called Surrogate servers. These machines are used to cache content and distribute it around the world to serve the closest clients, as it can be seen from Figure 2.2. The client-server communications are represented by two-ways communication flows. The client first communicates with the closest surrogate server and then the surrogate server with the origin server. To distribute copies of identical content over the servers, a good practice is to place the surrogate servers in strategic locations over a globally distributed infrastructure [20]. Doing so leads to:

- Reduction of the costs associated with a big data center infrastructure.
- Reduction in latency, since data is closer to the users.
- Improvement of the content quality, speed and reliability.
- Reduction of high loads on the origin servers.

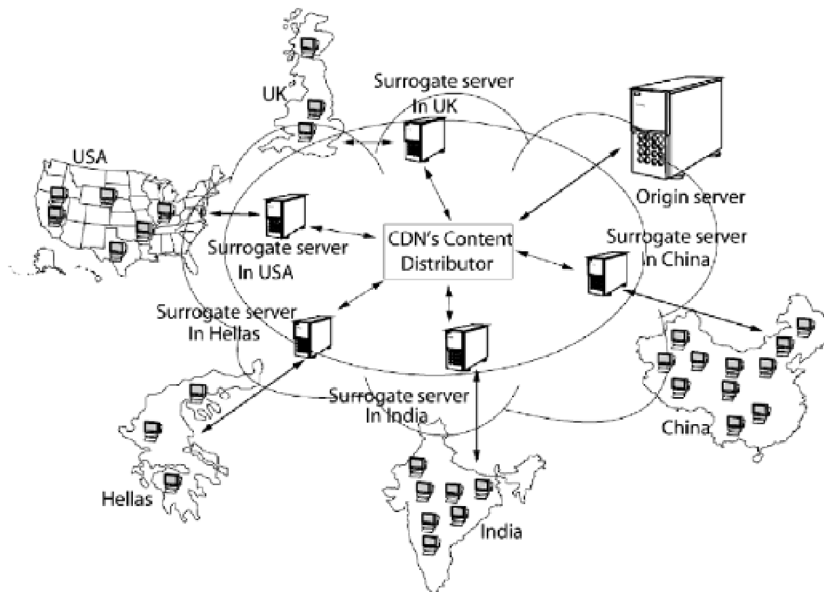


Figure 2.2: Typical structure of a CDN [2]

Content refers to any digital data resources and it consists of two main parts, the encoded media and metadata. In the encoded media we have, video, audio, documents, web pages and images. The metadata, on the other hand, encompasses all content description that allows identification, management and discovery of so. A CDN system can be composed by three parts: the content provider, the CDN provider and the end users.

Content provider: it can be seen as the origin server that holds all the Web objects to be distributed.

CDN provider: is the organization or company that provides the infrastructure facilities to content providers. These are the ones who own the surrogate servers that replicate the origin server content. The set of surrogate servers can be called Web Cluster. Surrogate servers also send accounting information about delivered content to the accounting system of the CDN for traffic reporting and billing purposes.

End users: are the ones who request content from the content provider website [2].

2.7 Distributed content distribution

There are some Peer-to-Peer (P2P) technologies that allow for decentralized content distribution, being the most famous one the BitTorrent network [21]. This system is used to download files shared by other computers in a P2P network, with 15 to 27 millions active peers at any given time during 2013. In this context, there is also the Blockchain technology that allowed the creation of cryptocurrencies and that can also be used to distribute content. This is achieved with what is called the smart contracts, a way of making secure and anonymous transactions without the need of any middleman. This distributed approach is also possible using the Hadoop framework, a technology developed to use on Big Data applications. It distributes data and computational operations across thousands of servers in a distributed way, providing high scalability and efficiency.

A promising alternative open source technology is the InterPlanetary File System (IPFS) [8]. It's used to create a decentralized distributed file system, that aims to be a complement or even replace the traditional centralized internet. It creates a big swarm of peers in a P2P network, where they all work alongside to store different parts of files and distribute them to peers that make a request. This technology is well suited to distribute video and audio files in the network, since files or parts of it can be stored across multiple peers in the network. So that when someone request that video, it fetches the several parts of it from all the available peers that have that video file.

2.7.1 BitTorrent

The BitTorrent is a very popular file distribution system and a well know alternative to the conventional centralized Internet. Instead of placing a file in a host server, the file is downloaded directly from multiple peers. When a peer needs a given file, it joins a swarm where all peers are sharing that same file. Additionally, when a peer is downloading a file it also uploads parts of it to other ones in the swarm. The protocol tries to keep the fairness between peers with the tit-for-tat strategy. This policy makes peers download and upload files in a evenly manner, preventing nodes from abusing or exploiting the system. Even after a peer as finished downloading a file, it can keep seeding it to other peers in the swarm, which BitTorrent rewards with BitTorrent Tokens [22].

Currently BitTorrent has Trackerless Torrents, that don't need a centralized server to manage the swarms. This way each peer is able to discover each other by using Distributed Hash Tables (DHT), where every peer stores a table with all the other peers that he knows. After that it can ask to a known peer to contact another peer and so forth, until the peer that has the wanted file is found and joins the respective swarm [23] [24].

2.7.2 Blockchain

A blockchain can be seen as a decentralized distributed immutable ledger. The blockchain is mostly known by its use in the cryptocurrency world where a block stores three types of information: the data it contains, its hash and the hash of the previous block. These blocks are then linked forming a chain, hence its name. To explain the function of blockchain in cryptocurrency let's assume the following situation. When a computer in the network (i.e., a node) named A performs a transaction with node B, that transaction is stored in a block. Afterwards, that transaction goes trough a hash function, generating a large string of letters and numbers, that represents the fingerprint of the data in the block. Since nodes are connected in a P2P topology forming a big cluster, that new block is sent to all nodes in the cluster without the need of a middleman. All nodes in the network holding the blockchain then validate the transaction and if the majority of them confirms its truthfulness, the block is then added to the chain.

In the context of content distribution, blockchain technology can be a game changer for content creators. For example, in platforms like Youtube creators only get 55 percent of the ad revenue from their videos [25]. Platforms that work as middlemans always profit from content creators success by taking a big share of their revenue. Also their content can be easily used by other people, e.g., having their music played on bars, stores and so on. This can happen without the content creator acknowledgement and without receiving any money from it.

In this regard blockchain has something called smart contracts, heavily used on the Ethereum cryptocurrency [26]. A smart contract is a protocol made to help exchange money, in a conflict-free way and without any middleman in the transaction. In case a certain condition that is specified in the smart contract gets validated, the transaction may happen. Since blockchain is used in P2P networks, all peers in the network will validate all transactions making it almost impossible for someone to temper with contracts. A good example would be a smart contract in the context of a fundraising campaign. Lets define that the fundraising will have a life span of a few days to raise a certain amount of money, otherwise the collected money should be refunded to everyone that donated. But if the goal is reached, then all the money is sent to whatever the cause was and without any fees in between.

Back again to content distribution, smart contracts can then be used for content creators to sell their content without any platform taking a big share of their revenue. Thus having full control of their content making it harder for others to use it without permission [27].

2.7.3 Hadoop

The Hadoop is a distributed open source framework, released in 2005 by the Apache Software Foundation [28], and is used to store and process very large data sets and distribute them to user applications. It was first mainly used for web search engines, like Yahoo or Google, to improve their performance by distributing data and performing calculations across multiple computers. It runs on a Hadoop Distributed File System (HDFS) cluster of thousands of servers that distribute data among them and execute user application tasks. The Hadoop provides a distributed file system and a framework for the analysis and transformation of data using what is called of MapReduce [29]. Its architecture consists on the following [30]:

- **NameNode**: is the master server in a HDFS cluster. It manages the HDFS namespace, which is a hierarchy of files and directories. The file content is split into blocks of 128 MB, being those blocks replicated across multiple DataNodes. The NameNode is in charge of maintaining the namespace tree and the mapping of file blocks to DataNodes. When a client wants to read from a file it contacts the NameNode for the location of the data blocks, then the client reads those blocks from the closest DataNode that contains those blocks. Additionally, if the client wants to write data it requests the NameNode to nominate three DataNodes to host the block replicas.

- **DataNode**: these nodes are responsible for storing the data. Each block of data on a DataNode is represented by two files in the local host native file system. The first contains the data itself and the second contains the block's metadata. The DataNodes send heartbeats to the NameNode to confirm that is operating and that the block replicas it hosts are available. Those heartbeats are sent every three seconds, and in the case the NameNode does not receive a heartbeat from a DataNode within 10 minutes, it assumes that the DataNode is unavailable and schedules the creation of new replicas of the blocks on other DataNodes. The heartbeats sent carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. Additionally, the NameNode replies to the hearbeats with instructions to the DataNodes. The instructions include the following commands: replicate blocks to other nodes, remove local block replicas, re-register or to shutdown the node or send an immediate block report.

2.7.4 InterPlanetary File System

Based on the previous sections one can point out that, a possible alternative to a CDN would be a system that uses the local storage space of users as an alternative to Surrogate servers storage. Even though CDNs are decentralized they still rely on a main server that manages the Surrogate machines, meaning that the typical CDN architecture uses both decentralized and centralized topologies, but it is not a distributed network model. A purely distributed topology means that the nodes in the network are all connected to each other without the need of dedicated servers, where in a CDN the nodes are connected only to the Surrogate servers. An illustration of these three different networking concepts can be seen in Figure 2.3.

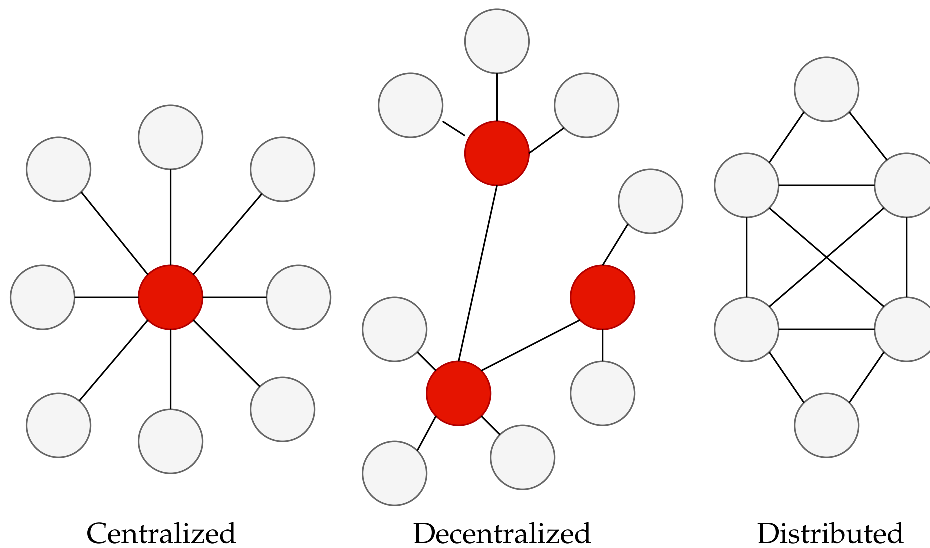


Figure 2.3: Visual representation of a centralized, descentralized and distributed network.

Here is where the InterPlanetary File System comes in. The IPFS is a distributed system for storing and accessing files, websites, applications and data. Built using a P2P topology, it allows for users to exchange data between them without the use of a server in the middle, with all nodes in the network working has a client and a server.

In the mainstream Internet users type in the location of the content they want to reach using a browser, however, in a IPFS network nothing is location based but content based. This means that if we are going to access a Wikipedia page, for example, instead of typing the Uniform Resource Locator (URL) of the page, and request the content to a server, we would search for something like:

/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco

This multihash represents the content of the Wikipedia version stored in the IPFS network. So instead of making the request to a server, IPFS asks the other devices that are connected to the IPFS network around the world if any has the content. Then the page can come from anyone that already requested that page or that has parts of it locally stored. Now that the end user already has the Wikipedia page, it can now also distribute it to other IPFS users. This works not only for web pages but any type of file, such as, documents, e-mails, picture or even database records.

Unlike a centralized network system, the P2P systems consider each computer as an interdependent node, meaning that all nodes can work has a file server and a client. However the owner of a node cannot delete data from it, only other nodes themselves. This posses a big issue since complete data deletion is impossible in P2P file sharing systems. After users upload their files into a P2P node, the network system will start to propagate the file through all neighborhood nodes. Therefore, the user cannot remove a file from the P2P system completely. This is due to the fact that P2P distribution systems treats each node equally and the original user will not have access to the file on the other network nodes [31].

The main advantages of the IPFS:

- It enables the download of files that aren't managed by a single organization. As an example it can be mentioned that if for some reason the Wikipedia servers were down, people would still be able to access their webpages requesting them to other IPFS users. This happened in 2017 in Turkey where the government blocked Wikipedia, so a group of people uploaded to the IPFS network a copy of the Turkish Wikipedia, making it impossible for the government to prevent the access to it [32].
- It can speed up the web. When a person that requests a file from a server that is physically thousands of kilometers away, it may take a lot of time waiting for the file to download. This will be even worst if that person has a poor Internet connection. However, if that file was already stored in a neighbor computer, it could be retrieved locally making it a lot quicker to access the data [33].

There are already some case study projects that use the IPFS as its backbone. Some of them are the following: PeerPad is a collaborative text editing web application [34], Berty is a mobile messaging app [35], a P2P video streaming platform named Blust [36] and many others applications. For more examples the reader can refer to [37].

2.8 Discussion

The mainstream Internet gets the job done, however it is inefficient in terms of resource usage, and can be expensive when delivering content to the entire world. Nowadays efficiency is key, the more efficient the methods chosen by companies are, the more profits they can make.

When designing the system architecture of the project Vixtape, the two most important things taken into consideration were the need to be investment efficient and to enhance quality over similar offers. That is why it was important to choose a distributed solution to manage the project's content distribution. The BitTorrent is a well known and stable option, but lacks the possibility to replicate the content to every player, and also still relies on a centralized server to guide peers to the swarms. The Blockchain technology could also be an option but it does not replicate content and its complexity could lead to an extended development time. The Hadoop could be a good option to distribute content within this context. However, it is mainly used to store huge amounts of data and perform complex operations with that data, which makes it more suited for the Big Data subject. Moreover, the Hadoop framework uses a centralized topology which leads to a single point of failure and does not offer the ability for this project to have a proper offline resilience. The IPFS seems to be the option because it is well supported by the open source community, easy to integrate with the Android and web players, and enables data replication through all Vixtape players. After the survey of the current state of content distribution it stands out that the new open source IPFS presents itself as the ideal support technology for the problems this project aims to solve.

This approach will leverage content distribution by using the storage of all the devices that will locally run the advertisements management app. This route is expected to be more efficient than using traditional CDN servers to cache all the data. By being efficient in terms of performance and associated costs, the final price of the entire service can be lower, thus more competitive across the market.

Overtaking all the common issues that are normally seen in advertising displays is the biggest priority for this project. The main one being that the displays should always be playing content under any circumstance. Connecting all Android Boxes in a P2P network will allow for them to share the content between them, even when experiencing Internet connection issues.

Chapter 3

InterPlanetary File System

The IPFS is a P2P version controlled file system inspired in other successful systems like BitTorrent. With P2P, users are able to share computer power, data, and bandwidth among one another.

3.1 Architecture Overview

The IPFS stack can be visualized in the diagram of Figure 3.1. The Application layer is on top of the stack and it can be any website or application that is built on top of IPFS, like project Vixtape is. After the Application layer, Naming data is achievable via the InterPlanetary Name System (IPNS), that allows for a more human readable URL instead of using multihashes directly. That data is then linked using MerkleDAG, where the data is divided into multiple blocks forming a graph. In this case, the multihash generated from the data represents a set of blocks all linked together that defines its data structure. In the Exchange layer, data is exchanged with an algorithm inspired from BitTorrent, called BitSwap. This algorithm is based on a probabilistic function that decides if it will exchange data or not, with a specific node based on the exchange history between them. The Routing part uses Distributed Hash Tables (DHT) to find other nodes in the network. All nodes store in a table the hashes of the nodes they know, with that info and using a binary tree search, nodes are capable of finding any node in the network. Finally, the Network layer has the the purpose of managing the connections between the nodes, and it can use various protocols depending on the configuration used, but it's mostly based on the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

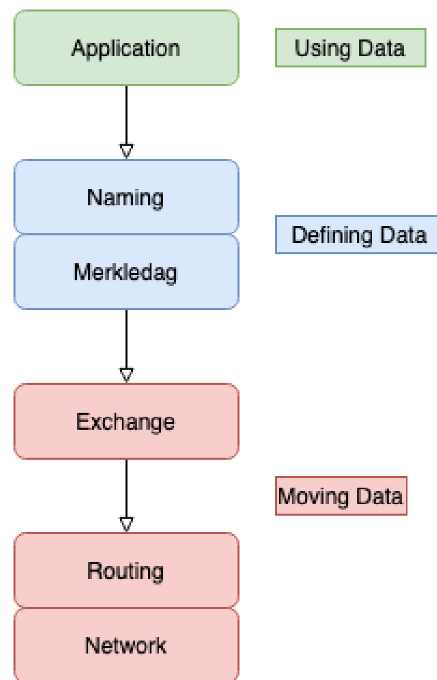


Figure 3.1: Overview of the IPFS stack

3.2 Content addressing

Consider the multihash example previously introduced in Section 2.7.4 that points to a copy of Wikipedia:

/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco

The string that comes after **/ipfs/** is called Content IDentifier (CID). This feature is how the IPFS can get content from multiple network nodes. The CID is a cryptographic hash of the content, consisting in what is called a multihash. A multihash follows a pattern of **<hash-func-type><digest-length><digest-value>** [38], where:

- **<hash-func-type>** is an unsigned integer variable identifying the hash function.
- **<digest-length>** is an unsigned integer variable counting the length of the digest in, bytes.
- **<digest-value>** is the hash function digest.

There are several hash functions available, but IPFS uses the sha2-256 [39] and then encodes the final multihash into a base58 string. The first byte of the

multihash will be **0x12** in hexadecimal, because that is the code that represents the hash function sha2-256 according to the multicodec table [40]. The digest length will always be 32 bytes which is **0x20** in hexadecimal, and the digest value is the data encoded from the sha2-256 hash function. Once the multihash is created it will then be encoded into a base58 string. Moreover, the reason why every CID starts with the prefix "Qm" is due to the fact that the sha2-256 string before being encoded into base58 looks like "1220...", and after encoding will look like "Qm..." [41]. As a result every hash is unique to the content, meaning that it will not exist equal files with different hashes.

3.3 InterPlanetaty Name System

At some point files are going to be updated and changed, therefore creating a different CID for it. This means that every time a file is modified there is the need to redistribute the content link. The IPNS solves this issue by creating and updating mutable links to IPFS content. A name in IPNS is the hash of a public key, which is associated with a record containing information about the hash it links to, and that is signed by the corresponding private key. The link that the IPNS generates points to the CID with the latest version of the content [42].

It is also possible to work with a more human friendly link, since links that consist of multihashes are hard to read and memorize. The IPFS also has DNSLink that work almost in the same way as IPNS, but instead of using another multihash that points to a content CID, it uses a readable name like in the following example:

```
/ipns/myexampledomain.org/media/
```

This allows for the creation of web apps within the IPFS network that can take advantage of a more friendly usage [42].

3.4 Merkle DAG

The name Merkle DAG comes from the inventor of hash trees Ralph Merkle and DAG meaning Directed Acyclic Graph. In practice IPFS does not hash the entire file into a single CID, what really happens is that the file is divided into smaller parts, with a maximum of 256 kB each. Those parts are then combined into a linked hierarchical data structure represented by the base CID. This process is illustrated in Figure 3.2. The hierarchical data structure is called a Merkle DAG, and what IPFS did was use this core data structure and named it InterPlanetary Linked Data (IPLD) [3] [43].

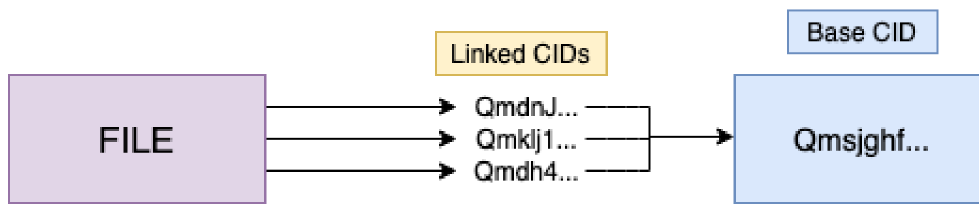


Figure 3.2: Base CID referencing to linked segments of a file

The Merkle DAG data structure allows [44]:

- Content Addressing: the content is identified by its multihash checksum, including links.
- Tamper resistance: all content is verified with its checksum. It can detect if data was tampered or corrupted.
- Deduplication: all objects that hold the exact same content are equal, and only stored once. A good example of this is illustrated in the Figure 3.3.

In IPLD objects have the Data and Links fields. Data is just an unstructured arbitrary binary data field, and Links is an array of Link structures. Each Link has a Name, a CID and a Size. The Name is the name of the Link, the CID is the Content Identifier of the linked IPFS object and the Size is always less than, or equal to, 256 kB. The IPFS Command Line Interface (CLI) allows to see the linked hashes of a CID using the command **ipfs ls -v <CID>**, as shown in the following example:

```
>> ipfs ls -v QmcNK2CFNjRSrnWJegyxxHrDdBL1haVKZxBeKueWEQNDFF

Hash                                           Size    Name
QmRdXs5EYKEdVEBn5nj5FHso14b5LwkSdLrtQFYHjvFWQ5 262144
QmcK31owJxzXwtmBWh165PDfkiAM6n4ER1PFdPzyGzSNro 262144
QmQbwspDnunsszNGpboTpbHu4vBzazRs3x2VzUmPzUv5wQ 262144
QmZjsApCyZxSfoRRpyjCpxsNjsnVzHWAdXAAT6DSgcGawq 262144
QmaqFHDGvmJENKickmwbgMmqr597k4Pnzk288rTXD4hRhj 262144
...
```

The hashes printed after the command are the hashes links of the content whose CID was used in the CLI command. The Name values are empty due to the fact that they are not specified in this case. Now with the next command it will be possible to see the data structures content. Keep in mind that the presented hashes list only refers to the first ones, from all the ones that are needed to address the entire content.

```
>> ipfs object get QmcNK2CFNjRSrnWJegyxxHrDdBL1haVKZxBeKue...

 {
   "Links":[
      {
         "Name":"",
         "Hash":"QmRdXs5EYKEdVEBn5nj5FHso14b5LwkSdLrtQFY...
         "Size":262158
      },
      {
         "Name":"",
         "Hash":"QmcK31owJxzXwtmBWh165PDfkiAM6n4ER1PFdPz...
         "Size":262158
      },
      {
         "Name":"",
         "Hash":"QmQbwspDnunsszNGpboTpbHu4vBzazRs3x2Vz...
         "Size":262158
      },
 ...
```

Merkle DAG allows the description of any type of data structure using IPLD. For instance, it's possible to add directory structures to the IPFS. The next example illustrates how data can be linked together and the benefits of deduplication. Consider the file structure:

```
dir/
|--oneFile.go
|-- *hello.txt
|--Another_dir
    |-- *another.txt
    |-- *and_hello.txt
```

Assuming that the files with the asterisk (*) contain the exact same text inside, they will have the same CID despite being different files. Adding the directory is done with the following command:

```
>> ipfs add -r dir/
```

The resulting DAG is illustrated in Figure 3.3. At the top level there can be seen the directory root that instead of a name it uses the CID. The root has a

direct link to the file **oneFile.go**, to the other sub-directory called **Another_dir** and to the file **hello.txt**. Both **another.txt** and **and_hello.txt** have the same CID, deduplicating because it only refers to content and not the file itself. Since **on-File.go** has a size bigger than 256 kB, it will be segmented into smaller parts that are linked back to the file. The CID at the top tree describes all the content below it.

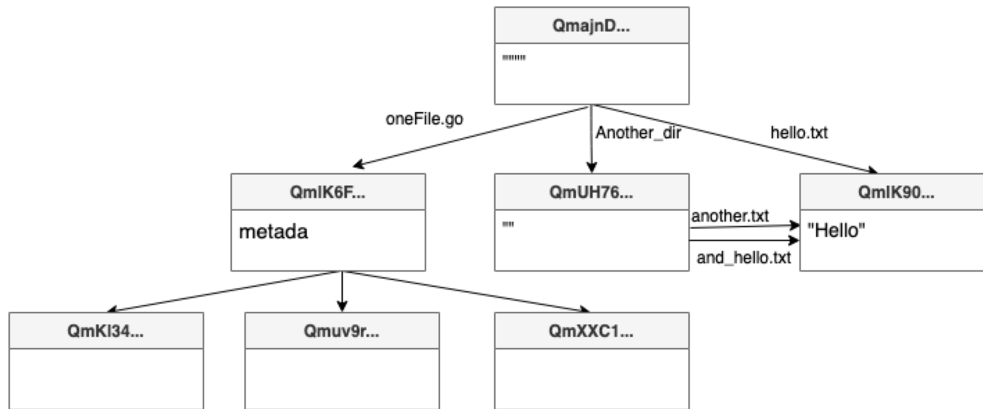


Figure 3.3: Data structure of linked files in a directory, based on the image [3]

3.5 Version Control System

One of the most popular and used Version Control System (VCS) is Git [45], and the IPFS version control system is inspired by it. Version control systems provide features to model files changing over time and distribute different versions efficiently. Git provides a vigorous Merkle DAG object model that captures a file system tree in a distributed-friendly way [46]:

1. Files, directories and changes are represented respectively by the objects blob, tree and commit. All are immutable objects.
2. Objects are content-addressed, i.e., files are never referenced by their name but by the cryptographic hash of their content. Git hashes the content with a SHA1 hash function and stores it in its database.
3. Links to other objects are embedded, forming a Merkle DAG data structure.
4. When a version changes it means that references were updated or/and objects were added.

The commit object represents a particular snapshot in the version history of an object. When comparing two objects of different commits, it is possible to see

the difference between two versions of the file system. To access all versions and the full history of the file system changes, all it's needed is a single commit and then all the children objects it references are accessible. The IPFS makes use of the Git version control tools. The objects model is not the same but compatible. It is possible to mount a IPFS file system as a Git repository, translating Git file system reads and writes to the IPFS formats [46].

3.6 Distributed Hash Tables

The IPFS relies heavily on a Distributed Hash Table (DHT), these are distributed key-value stores where keys are cryptographic hashes, and the values are blocks of data. Each node of a IPFS network is responsible for a subset of a DHT. This method allows the nodes to keep track of the data kept by other nodes in the network. The DHT supports decentralization and scalability, as opposed to using one table containing the information about what data exists in all the nodes, like it would be done using a centralized database. The table is distributed to all nodes in the network, so each one only owns a small part of the entire table. In order to keep this table up to date, every time a node adds a file to the network, this is announced to the other nodes so they can then update their tables.

A problem that might come up is when a node goes offline, since the node will not announce of so. I this situation, the nodes have in their tables that, e.g., node A has a certain block of data when the reality is that node A is offline at the moment. So when another node tries to connect to node A this one will not respond. To prevent this scenario, all nodes should re-announce to the network all their stored CIDs every 12 hours.

The IPFS uses a type of DHT called Kademlia, this allows for nodes to know where the data most likely is and how to reach other nodes that they do not know [47]. There are different types of DHT, e.g., Kademlia DHT, Coral DSHT and S/Kademlia DHT. The IPFS uses the Kademlia DHT with some additional features from Coral DSHT and S/Kademlia DHT.

3.6.1 Kademlia DHT

The Kademlia tables minimize the number of configuration messages that nodes send to discover other ones which spreads automatically through the lookups [4]. It also uses parallel and asynchronous queries to avoid delays from failed nodes. Kademlia follows the same approach as other DHTs where keys are opaque with 160-bit. Each node has an ID in the 160-bit key space, key values pairs are stored on nodes with IDs similar to the key for some relative accuracy.

This method is achieved with a XOR metric for distance between points in the key space.

Since each node has a 160-bit ID and keys in the DHT are also 160-bit, it's possible to achieve a notion of distance between two identifiers. Given two 160-bit identifiers being one x and the other y, Kademlia defines the distance between them as their bitwise Exclusive Or (XOR), interpreted as an integer, $d(x,y) = x \bigoplus y$. XOR is a valid, albeit non-Euclidean metric, so $d(x,y) = 0$, $d(x,y) > 0$ if $x \neq y$ and $\forall x,y : d(x,y) = d(y,x)$. XOR also offers the triangle property: $d(x,y) + d(y,z) \geq d(x,z)$. By the triangle property it follows $d(x,y) \bigoplus d(y,z) = d(x,z)$ and $\forall a \geq 0, b : a + b \geq a \bigoplus b$.

For a node to locate other nodes near a specific ID, Kademlia uses a single routing algorithm from start to finish. On the other hand, other systems use one algorithm to get close to the targeted node and a different algorithm for the last hops to finally get to it. Kademlia treats nodes as leaves of a binary tree, with each node's position determined by the shortest unique prefix of its ID. On Figure 3.4 it is shown the position of a node with an unique prefix 0011 representing the black dot in the right side. For every node the tree is divided into a series of successively lower subtrees that exclude the node. The highest subtree consists of the half of the binary tree not containing the node, while the next subtree consists of the other remaining half not containing the node, and so forth. In the given example with the node 0011, the subtrees are circled and consist of all nodes with prefixes 1, 01, 000, and 0010.

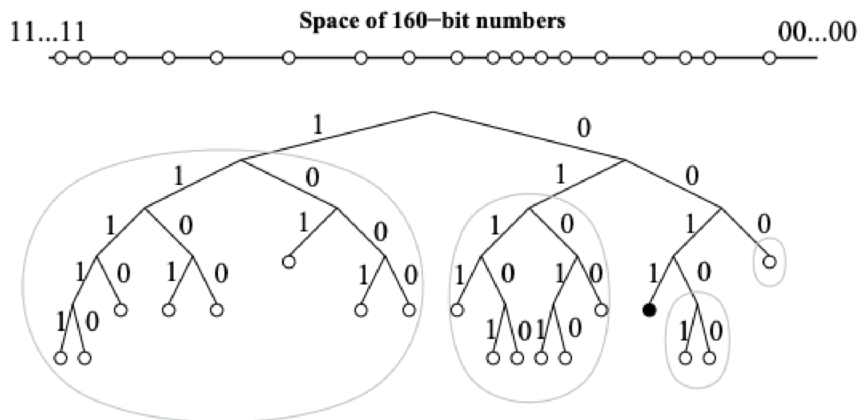


Figure 3.4: The Kademlia binary tree as an example [4]

The protocol makes sure that every node knows of at least one other node in each of its subtrees, in case that subtree contains a node. With this assurance, any node can locate any other node by its ID as it is shown in the example of Figure 3.5. This case shows the node 0011 locating node 1110, by successively

querying his known nodes by searching in lower subtrees until the lookup converges to the target node. The node 0011 finds the node 1110 by successively learning from, and querying closer nodes. Messages are first sent to node 101 which is already known by node 0011, then the next messages are sent to nodes returned from the previous messages, until it finds the targeted node [4].

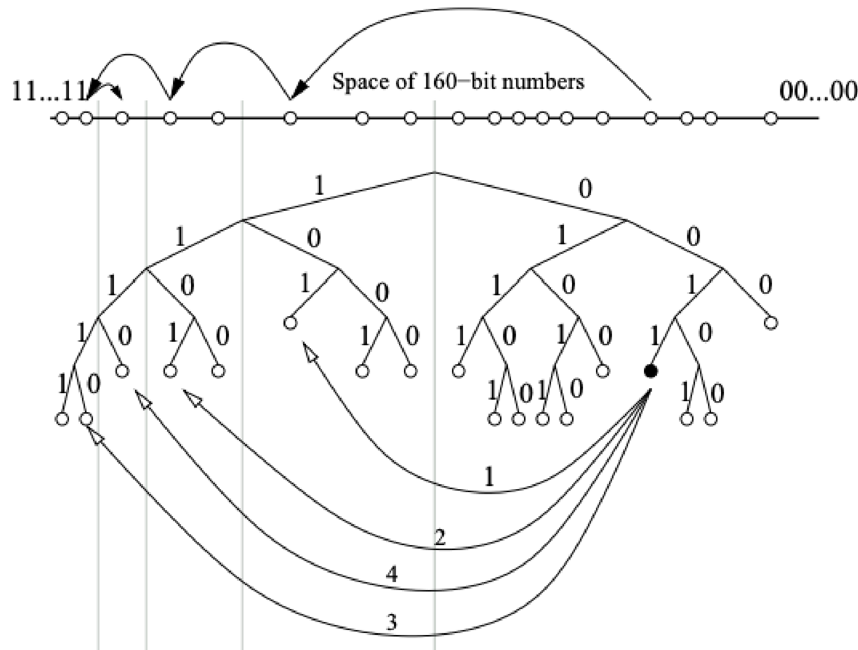


Figure 3.5: The process of locating a node by its ID [4].

3.6.2 Coral DSHT

The IPFS Kademlia implementation also makes use of some components relative to Coral Distributed Sloppy Hash Table (DSHT). Coral uses a sloppy storage technique that caches key/value pairs at peers whose IDs are close to the key being referenced. This technique avoids overloading all nearest nodes when a key becomes popular [48].

Coral uses several levels of DSHTs called clusters. It enables nodes to first query peers in their region in order to find nearby data without going far. With this method it reduces the number of lookups and the latency of each lookup [46].

3.6.3 S/Kademlia DHT

The S/Kademlia DHT offers an improvement in security, making it useful to protect against malicious attacks. S/Kademlia provides schemes for secure

node ID generation, and prevent Sybil attacks [49]. This is achieved by requiring nodes to create a Public Key Infrastructure key pair [50]. One of the schemes includes a proof-of-work crypto puzzle to make generating Sybils expensive.

For a deeper understanding of the Coral DSHT and the S/Kademlia DHT methods, the reader should refer to the cited references.

3.7 BitSwap protocol

The BitSwap protocol was inspired by the BitTorrent protocol. In BitTorrent the blocks of data are traded with peers that are also looking for those blocks, creating a torrent swarm around that file. With BitSwap there is only one big swarm, so when requesting content from the network, the blocks (Figure 3.6) can come from multiple peers or just one. It depends if more than one peer has the requested file. When peers are looking to acquire a set of blocks, those blocks are saved in a called *want_list*. To start the discovery process, to know what peers have the data, a *want-have* request that contains the root CID of the file is sent to all known peers in the network. In case no one answers, or the known peers do not have all the blocks, the peer that is requesting the data will search for unknown peers that might have them. The process of searching for unknown peers in the network was previously explained in the DHT section, 3.6. Peers that have the root block will send a *have* response and then a session will be created with all the peers that answered. The peer looking for the file will send each peer in that session a *want-have* requests, but this time with the CID of the blocks of data. This process is illustrated in the Figure 3.7.

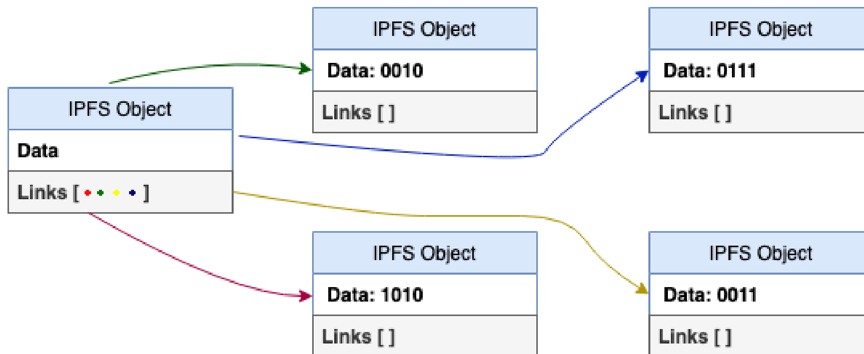


Figure 3.6: Example of how a file could be segmented into blocks

BitSwap operates as a persistent marketplace that encourages data replication. The protocol must encourage nodes to seed their blocks even when they don't need new blocks, because other peers might need blocks that they currently have. To motivate nodes to seed, BitSwap uses a credit system where the balance with other peers is tracked, being the number of bytes sent/received

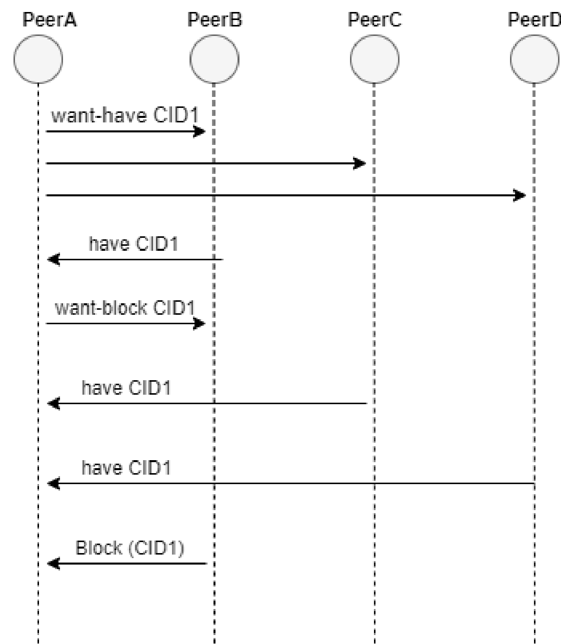


Figure 3.7: The process of transferring blocks of data (based on [5])

the "currency". When a peer requests blocks from another peer and that peer sends them, the peer that made the request now has a debt to the other peer. If the peer that requested the data already has a debt to the supplier node, then this peer uses a probabilistic function that determines if it will send the blocks or not. In the case it decides not to send the blocks then it'll ignore the peer for an *ignore_cooldown* timeout, which by default is 10 seconds. This prevents peers from trying to exploit the probability by causing more dice-rolls. The debt ratio between two peers can be described with the following equation:

$$r = \frac{bytesSent}{bytesReceived + 1}$$

Given r, the probability of sending to a debtor is:

$$P(send|r) = 1 - \frac{1}{1 + e^{(6-3r)}}$$

BitSwap peers might use different strategies when exchanging data, being the standard one the tit-for-tat strategy used by BitTorrent. Simply put, it consists in not receiving data if one doesn't share either. Additionally, it gets the rarest blocks first in order to improve the overall performance of the network

[51]. There are some other strategies like: BitTyrant (sharing the least possible), PropShare (sharing proportionally), BitThief (exploiting a vulnerability and never share). Despite the strategy used by the peers, all should aim to:

- Maximize the trade performance for the node, and the overall exchange.
- Prevent exploiting from peers.
- Be effective.
- Be compliant to trusted peers.

BitSwap nodes save accounting ledgers of the transfers with other nodes. When a data exchange starts both nodes exchange their ledger and compare if they're equal, otherwise the node that received the request to exchange data drops the connection. Also the ledge with that node restarts, losing all history of credit between them. If this happens the node can refuse to exchange future data with that node removing it from his trusted peers list. This prevents nodes from using tampered ledgers in order to receive data without seeding [46].

3.8 IPFS Cluster

The IPFS network can be public or private. When adding content to the public IPFS network, that content is accessible to everyone thus making it not the best option when dealing with sensitive data. Creating a private IPFS network can be a good solution for many use cases. Not only for privacy matters but also for performance reasons. By making peers know all the other peers in the same network, it can be a lot faster to fetch content from a known peer in the swarm, instead of asking to a lot of peers in the public network for a specific content.

To achieve this a complement to the IPFS was created, the IPFS Cluster. The IPFS Cluster is a distributed application that works separately with IPFS peers, providing data orchestration across the swarm. It maintains a global cluster pinset, replicates, tracks and allocates content to the IPFS peers in the same swarm [9].

The IPFS Cluster runs as an independent service from the IPFS and needs to run after the IPFS daemon starts up. It communicates with the IPFS daemon using his HTTP API port 5001, providing a fully featured API and CLI allowing to perform actions on the cluster of peers. Pinning supports custom data replication factors. By default it pins content across all peers in the swarm, and adds custom metadata to pins like giving it a name. Clusters allow what are called follower peers, they can store content that is pinned by others but are not allowed to change the pinset.

However, it is still possible to create a private IPFS network without the IPFS Cluster application. That can be accomplished by changing the IPFS configuration list of default bootstraping peers with the list of desired ones. But without the IPFS Cluster these peers will not replicate pinned content between each other. All the peers inside a cluster don't need to be connected to the public IPFS network since the IPFS daemon is a separate service from the IPFS Cluster. For example, one IPFS peer can request content from the public network and pin that content inside the cluster it belongs to, even if some of the peers are not connected to the public IPFS network. When starting the IPFS Cluster service it needs to be specified at least one peer to bootstrap with, in order to join the cluster. Additionally, the peers can also rely on Multicast Domain Name Server (mDNS) to discover other peers inside the same network [52].

When a cluster peer starts up it is very important to take into consideration what type of "consensus" is the best one to use, considering that this decision depends on the context the IPFS Cluster is going to run. The startup consensus component is in charge of:

- Managing the global cluster pinset by announcing changes to other peers and on its own.
- Managing the persistence storage of pinset-related data on disk.
- Managing the cluster peerset by adding or removing peers from the cluster.
- Setting peers trust.

There are two available consensus and when the IPFS Cluster starts up one of them must be chosen: the Conflict-free Replicated Data Types (CRDT) or the Raft Consensus.

3.8.1 CRDT

Achieving replication and consistency is a very important feature of any distributed system. One of the first motivations that drove the development of CRDT was collaborative text editing. The CRDT is a type of data structure used to replicate data across multiple nodes in a network, that allows to work offline, accessing the same data from different devices, and allow people to modify, add or remove data.

The CRDT satisfies the principle of Strong Eventual Consistency (SEC) which means that, if two replicas see the same event, they immediately share the same state. Another characteristic of CRDT is monotonicity, meaning the absence of rollbacks. A good example is comparing the behavior of Git with

CRDT, in the situation where two programmers modify the same line of code and then make a pull request. In this case, Git would throw a merge conflict that needed to be resolved manually. While with CRDT replicas are guaranteed to converge in a self-stabilised manner, regardless the number of failures that might occur. Additionally there are two types of CRDT: state based and operation based.

3.8.1.1 Operation based

An operation based CRDT, also called Commutative Replicaded Data Type (CmRDT), is a immutable sequence of objects, or by other words a tuple. It is represented by (S, s^0, q, t, u, P), where a replica has state $s_i \in S$, being that the initial state is s^0 and the state domain S. The pair (t, u) are the methods that update the objects, where t is a side-effect-free prepare-update method and u is an effect-update method. The prepare-update executes at the single replica where the operation is invoked. At the replica source, prepare-update method t is followed immediately by effect-update method u. The update operation broadcasts to all other replicas in the cluster and expects them to replay that update. P is a delivery precondition, the effect-update method is enabled only if the precondition is satisfied [53]. Operation based CRTD needs to follow the rules:

1. All concurrent operations must commute.
2. Updates must be applied exactly once.
3. Updates must be applied in order.

In Figure 3.8 it can be seen an example of how a simple counter value change can be propagated to all replicas. Assuming 0 (zero) being the initial state s^0, the addition of the constant 3 (three) to replica x1 is represented by x1.$t(0)$; x1.$u(3)$, where t is the prepare-update method and u the effect-update. Then replica x1 broadcasts the update to replicas x2 and x3 in the cluster.

A common case is when more than one replicas in the cluster update its state at the same time. What happens is that since it's a commutative operation the changes will be applied, i.e., $(x + 3) + 2 = (x + 2) + 3$. If instead of adding 3 to the initial state of x1, the operation was a multiplication, what would happen is that the addition of 2 or the multiplication would not take place. Note that the operation is not commutative $(x * 3) + 2 \neq (x * 2) + 3$, thus the final state would not be the same.

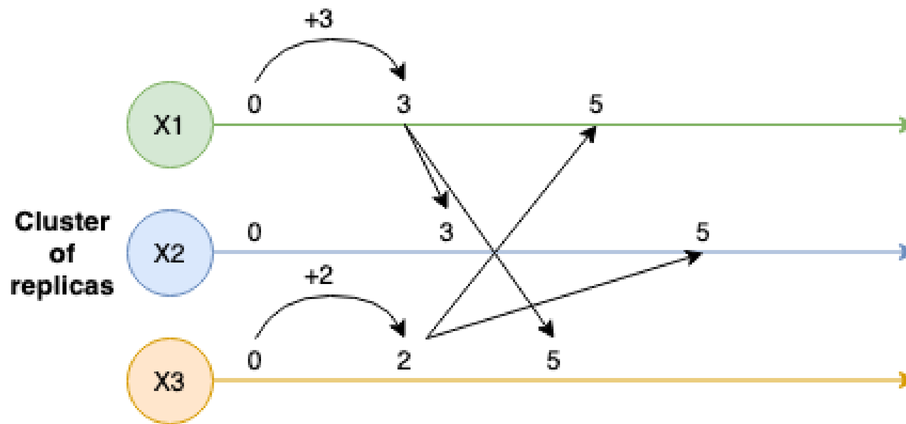


Figure 3.8: Operation based example where the state propagates to all replicas

3.8.1.2 State based

The state based CRDT also called Convergent Replicated Data Type (CvRDT), is considerably different from the operation based. Instead of sending the update to all replicas in a cluster, it first updates its own state and only then it will send the update to another replica, that other replica will merge the state and send it to the next replica. Eventually, every state update will reach all replicas in the cluster in a way that all have the same state. State based CRDT also uses a tuple type (S, s^0, q, u, m), where S and s^0 does not differ from the operation based object. A client may read the object's state via query method q and modify it using the update method u. The method m merges the state from the remote replica. Note that each method only executes at a single replica [53]. In sum the state based CRDT needs to follow the rules:

1. Allow re-transmissions,

   - The merge function should be idempotent.

2. The output must be independent of the order of merges,

   - The merge function is commutative and associative.

3. Needs a concept of "going forward", like indexes or timestamps.

4. Updates and merges always go forward,

   - Co-current states may not be comparable.

As an illustration of what was previously presented, Figure 3.9 shows how a object's state propagates to all replicas in a state based CRDT. Assume that an empty object is the initial state s^0 and that the first update is made by the

replica x1. This replica then broadcasts its update to only one of the replicas, which turned out to be x2, that in turn broadcasts the x1 update do x3. However, consider that at the time x2 received the update from x1 and merged it, it also updated. What happens is that x2 now broadcasts its new state back to x1, following the same cycle until all replicas ended up with the same state. In the rare event of, for example, the last state update {a:3, b:5} arriving at x3 before the first update {a:3}, the final output would be the same. This would be the case because the merge function knows that the object {a:3} is older than the the object {a:3, b:5}. This behaviour is commonly achieved by using timestamps or indexes.

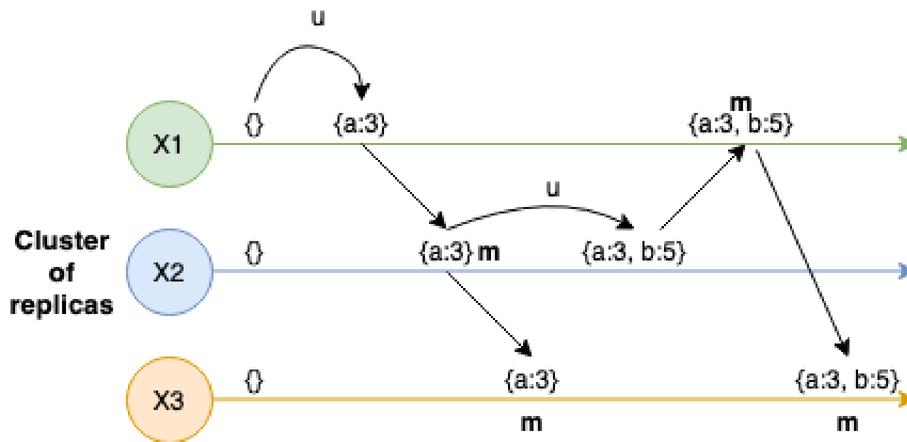


Figure 3.9: State based example where the state changes are propagated to all replicas

3.8.2 Raft consensus

The Raft consensus achieves the same final result as the CRDT but uses a completely different algorithm. Inspired by the Paxos consensus algorithm [54], Raft aimed to be easy to understand among developers and students.

A node in the cluster can have only one of three possible states: leader, follower or candidate. A node will always start up as a follower and then it may turn into a candidate, possibly into a leader in a later stage. In normal operation there is only one leader node while the remaining are followers. The follower nodes are passive, since they only respond to requests from the leader or candidates nodes. A leader node handles all client requests, in case a client sends a request to a follower node, this one will redirect the client's request to the leader. When a node is a candidate it means he can be elected to be the leader of the cluster.

Raft divides time into terms, they are numbered with consecutive integers

and every time there is a new election the term increases by one. Every follower node has a timeout, which is a random period between 150 ms and 300 ms. That time gets reset whenever the leader node sends what is a called a "append entry" message that is sent in intervals specified by a heartbeat timeout parameter. In case there is no leader or the leader became offline, the first follower node to reach the timeout will turn into a candidate node and starts an election process. For the node to win the election it needs to have the majority of the votes from the other nodes. So the candidate node votes for himself and sends a request vote message to the other nodes. If a node has not yet voted during the current term, its vote is then given to the candidate, after which the node timeout is reset. With the majority of the votes the candidate becomes the leader. The diagram presented in Figure 3.10 illustrates this leader election algorithm.

After the leader has been elected, then it is ready so receive requests from clients. Each client request contains a command to be executed by the node, being also added as an entry to the node's log. At this stage the entry is still uncommitted so it won't update the node's state yet. First it will send append entries messages to all nodes in the cluster with the command sent by the client. When the majority of the nodes reply, the leader then commits the log entry to its state and notifies all nodes of so. Only then will the other nodes also commit the log entry. After that the entire cluster will share the same state and the leader informs the client. In case followers nodes shutdown, run slowly, or network issues are detected, the leader retries append entry messages until all log entries have been stored by all followers [55] [56].

3.8.3 Consensus comparison

After studying both consensus it becomes clear the fact that both achieve the same end result but with different strategies. It is also clear that they can't always be used in all distributed systems applications. Depending on the context a cluster is going to be used, it is important to choose the proper consensus.

From the two consensus, CRDT and Raft, implemented by the IPFS Cluster only one can be selected. According to the IPFS Cluster documentation [57] a comparative sum up is given in Table 3.1.

As a result, the CRDT should be used when:

- It's expected the peers in the cluster to easily come and go.
- If the cluster is expected to contain follower peers that don't have permissions to modify the pinset.

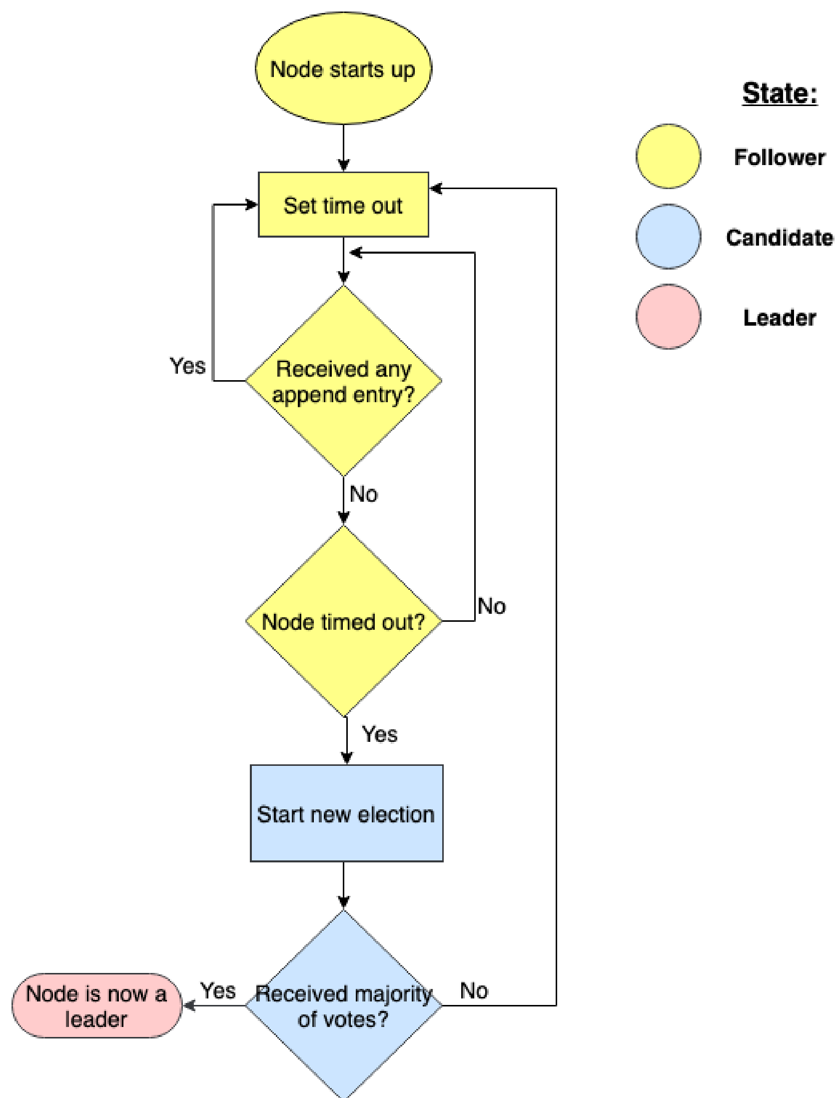


Figure 3.10: Leader election algorithm

- In case there are no fixed peer(s) for bootstrapping at start up, then peers will need to take advantage of the mDNS autodiscovery.

And choose Raft when:

- The number of peers in the cluster are always the same and always running.
- Reliability is important, because Raft is older than CRDT and also more tested.
- The cluster may have partitions with divergent states.

CRDT	Raft
GossipSub broadcast	Explicit connection to everyone + acknowledgements
Trusted-peer support	All peers trusted
"Followers peers" and "Publisher peers" support	All peers are publishers
Peers can come and go freely	>50% must be online at all times or nothing works.
State size always grow	State size reduced after deletions
Cluster state compaction only possible by taken a full cluster offline	Automatic compaction of the state
Potentially slow first-sync	Faster first-sync by sending full snapshot
Works with very large number of peers	Works with a small number of peers
Based on IPFS-tech (bitswap, dht, pubsub)	Based on hashicorp-tech (Raft)
Strong Eventual Consistency: Pinsets can diverge until they are consolidated	Consensus: Pinsets can be outdated but never diverge
Fast Pin ingestion	Slow pin ingestion
Pin commited != Pin arrived to most peers	Pin commited == Pin arrived to most peers

Table 3.1: Comparison between CRDT and Raft

3.8.4 IPFS Cluster Application Program Interface

The IPFS Cluster API runs over port 9094, hence a request must be sent to **<peer IP>:9094/<endpoint>** to trigger the desired event. The endpoints and their functionality are listed in Table 3.2 [58].

3.9 IPFS Application Program Interface

A running IPFS node offers a HTTP API [59] accessible through port 5001. This API has a large number of endpoints that go from the more important functionalities, e.g., adding or getting files from the network, to debug ones, like, getting ledger info from another peer or even the peer's DHT info. It is not mandatory to submit a request using the API to perform any action, because the IPFS library **go-ipfs** allows direct calls to the desired methods within the code through its core API. The IPFS also has an embedded CLI that performs requests to the HTTP API, this is much more easier than performing Client URL (CURL)

Method	Endpoint	Description
GET	/id	Returns the cluster peer information, such the number of peers it sees and its own addresses.
GET	/version	Returns the Cluster version.
GET	/peers	Shows the other peers in the cluster information.
DELETE	/peers/<peerID>	Removes a peer from the cluster.
POST	/add	Add content to the cluster.
GET	/allocations	List of pins and their locations.
GET	/allocations/<CID>	Shows the allocations of a given CID.
GET	/pins	Local status of all tracked CIDs.
POST	/pins/sync	Sync local status from IPFS.
GET	/pins/<CID>	Local status of a given CID.
POST	/pins/<CID>	Pins a CID.
POST	/pins/<ipfs | ipns | ipld>/<path>	Pins using an IPFS path.
DELETE	/pins/<CID>	Unpins a CID.
DELETE	/pins/<ipfs | ipns | ipld>/<path>	UNpins an IPFS path.
POST	/pins/<CID>/sync	Sync a CID.
POST	/pins/<CID>/recover	Recovers a CID.
POST	/pins/recover	Recovers all pins in the receiving Cluster peer.
GET	/health/graph	Get connection graph.

Table 3.2: IPFS Cluster API endpoints list

operations into the endpoints. Also the CLI already prettifies the output logs showing only the necessary information in a cleaner manner.

On the provided API, the operations can be called doing **<Node IP>:5001/api/v0/<Operation>**. Using the **cat** operation has an example, the call would be **127.0.0.1:5001/api/v0/cat?arg=<ipfs-path>**, which in turn returns the IPFS object data. The same operation using the IPFS CLI would be done using the command **ipfs cat <ipfs-path>**.

The most relevant endpoints on the context of this project are listed in the Table 3.3, while the full list of explained IPFS HTTP API endpoints can be found in Appendix A of this dissertation.

Method	Endpoint	Description
POST	/api/v0/add	Adds a file or directory to ipfs.
POST	/api/v0/bootstrap/add	Add peers to the bootstrap peers array in the config.
POST	/api/v0/bootstrap/add/default	Add default peers to the bootstrap peers array in the config.
POST	/api/v0/bootstrap/rm?arg=<peer>	Removes a peer from the bootstrap peers array.
POST	/api/v0/config?arg=<key>&arg=<value>	Gets and set the peers config values.
POST	/api/v0/config/replace	Passes a file to replace the current config file.
POST	/api/v0/ls?arg=<ipfs-path>	List directory contents for Unix filesystem objects.
POST	/api/v0/get?arg=<ipfs-path>	Downloads IPFS objects given its path as argument.
POST	/api/v0/pin/add?arg=<path>	Pins objects to the local storage.
POST	/api/v0/pin/ls	List the objects pinned in the local storage.
POST	/api/v0/pin/rm?arg=<ipfs-path>	Removes pinned objects from the local storage.

Table 3.3: The more relevant IPFS HTTP API endpoints

Chapter 4

Project Architecture and Technologies

In this chapter it will be detailed the technological stack used for the development of this dissertation project. The Vixtape project, as a all, makes use of both centralized and decentralized network models, therefore it will be explained in detail the project requirements, modules and how these systems communicate and work together.

4.1 System architecture

The Vixtape project aims to solve the multiple problems that are commonly seen in advertising screens that mostly are the result of a centralized approach to content distribution. The proposed solution to overtake these problems is to use a private distributed P2P network in order to have accessible content across all Android boxes that will run the ads when connected to a display device. As previously discussed, the most promising route is to use the IPFS technology as a way for the players to request new content.

It must be stated, however, that at the start of the works presented in this dissertation the Vixtape was already an ongoing project. At this point the Mosano's Vixtape project only has a centralized approach that is depicted in Figure 4.1. Considering the new goals for this project, the current centralized system will have to be redone in order to cope with the new distributed and decentralized approach, which is the main focus of the work exposed in this document. Furthermore, in case the centralized part fails or vice versa, the displays must be able to keep running content as long as one of the components is working.

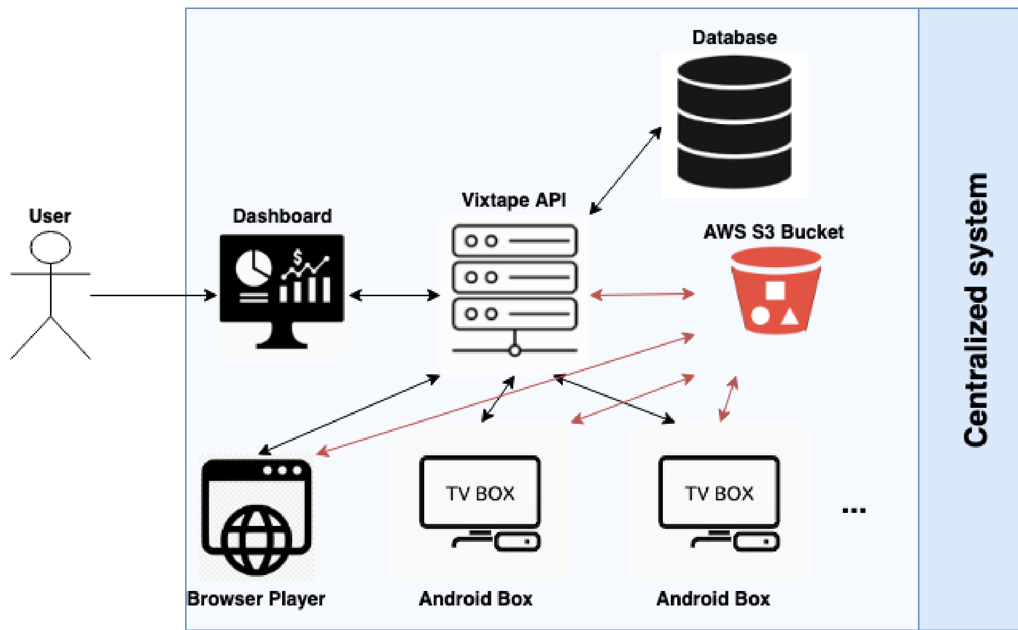


Figure 4.1: Architecture of the Vixtape system at the beginning of this project

The new system architecture for the Vixtape project can be divided in two main parts (Figure 4.2). The first one includes a dashboard app that grants the end user control over his displays and the content they show, see the generated revenue, among other features. All this is possible through the interaction of the Dashboard with the Vixtape backend API that works alongside a Database. Additionally, each Android Box that runs the ads player is also connected to the Vixtape API, thus allowing interactions like receiving scheduled campaigns to display. For a better understanding of how the dashboard settings influence the player operation, the section Appendix B provides an insight. So far this is a usual architecture that most web development projects follow. In this particular case, there needs to be a centralized approach for the user to be able to manage his displays, and for the Android boxes to be able to receive scheduling information.

The decentralized components lead to the second main part of the Vixtape project and this report describes its project, development and tests. One of the main elements of this part of the project are the Android boxes that will run the ads/content player. In order for a box to know the content is should play it must retrieve a broadcast plan using the Vixtape API. With that information they will need not only to fetch new content from the centralized AWS S3 bucket, but then they also must distribute it to the other Vixtape Android boxes using the IPFS public network and/or the IPFS cluster network to which they will belong to. The project also includes the development of a web browser player that will

Figure 4.2: The new Vixtape system architecture using the IPFS

use the same underlying logic of the Android player, but in this case it will not include the IPFS Cluster. Nevertheless, it still should connect to the IPFS public network from which will be able to fetch content from other players.

After a Android box boots, it will join a P2P private network formed by the other Mosano's Vixtape service Android boxes. The main intent being that the distributed network topology of the Android boxes (Figure 4.3) is achievable by means of the IPFS technology.

4.2 System requirements

The top requirements of the entire system can be divided into two different parts. The first one regards the interaction of the end user with the app control dashboard, where the user inputs will affect what type of content is going to be displayed in his televisions connected to the Android boxes. This part, not being the main focus of this work it is still an important factor to be taken into consideration throughout the design and development process. The second part of

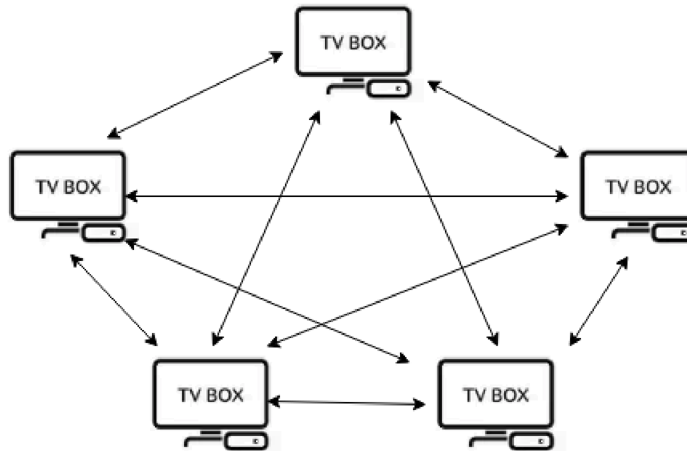


Figure 4.3: Android boxes connected in a P2P network topology

the system comprises the workflow between the centralized and the distributed components, and all the decision making procedures. This second part will be responsible for the content storage management and content viewing.

Abstracting from certain somewhat standard workflows like user authentication, session creation, fields update, password change, among others, this section will just detail the main functionalities that fall into the scope of this project. Ultimately, through the dashboard the user must be able to:

- Register in the backend system the players he owns.
- Define the type of audience, business model and geospacial location in order to get relevant ads.
- Schedule the viewing of ads and other multimedia content. This feature should allow the selection of, not only when to display, but also what to display.
- Upload his own content to the system.

This set of requirements is also presented in the form of a Unified Modeling Language (UML) use case diagram shown in Figure 4.4, where a client of the Vixtape service interacts with the backend API via the app control dashboard.

Once a player is registered the workflow should be fully automated. This means that as soon as the player is ready to display content, it should immediately fetch broadcast plans. It is expected that this will be the only time the user will ever has to see a loading screen, due to the lack of locally stored content

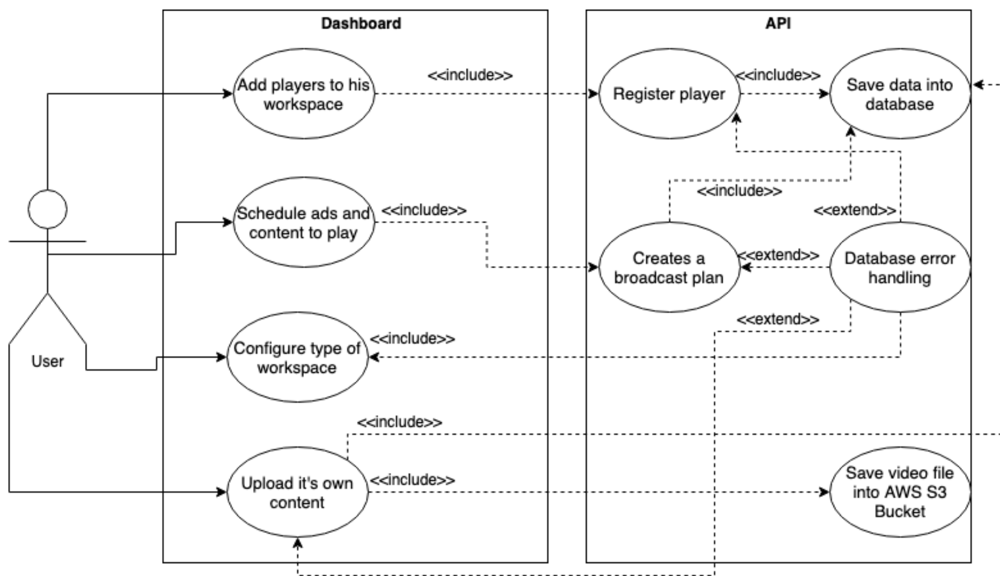


Figure 4.4: UML use case of the interaction between the user, dashboard and API

the first time a player runs. The diagram of Figure 4.5 summarizes the player's logic.

The aim is for the players to be able to:

- Always have at any given time at least 20 broadcast plans, which is a list of content, stored in memory.
- Fetch content from the IPFS network or from the AWS S3 bucket.
- Upload content to the IPFS network and to the IPFS Cluster in case the video file does not exist in it.

The player features will heavily rely on the Vixtape API which needs to be able to:

- Save new content that arrives to the AWS S3 bucket and to the IPFS.
- Generate broadcast plans (BP) based on the user's workspace settings.
- Send to the respective players the broadcast plans specified by the user.

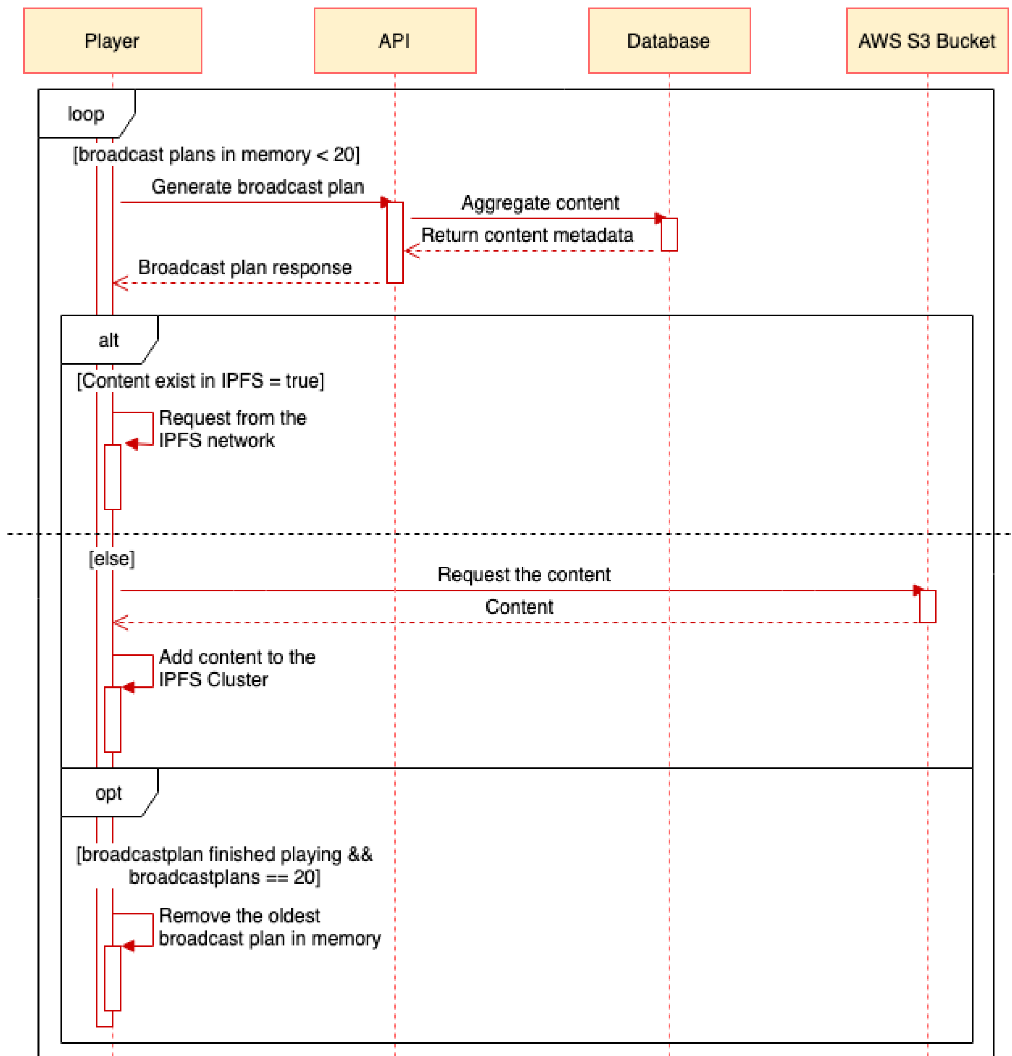


Figure 4.5: Player logic to interact with the rest of the system's components

4.3 Technologies

Since the goals of this dissertation work seat within the broader ongoing Vixtape project, the overall development framework was already established. Therefor, the stack of technologies and tools needed for the development was already defined by Mosano from the start.

4.3.1 Backend

The backend consists of a API on the server side developed using the NodeJS framework and GraphQL as the communication framework. Then IPFS will be embedded both in the backend API as in the Android Boxes in order for these

devices to be able to communicate with the public IPFS network. Additionally, the IPFS Cluster will also be used in all Android boxes to form a private P2P network. To store all data it will be used a NoSQL database named MongoDB. Finally, the AWS S3 Bucket, also known as Amazon Simple Storage Service, will store all content to be distributed to the players.

Vixtape's backend technological stack breakdown:

- **TypeScript** is a typed version of the programming language JavaScript. It extends JavaScript by adding types and type-checking, which detects bugs in the code before it gets build, speeding up development time [60].
- **NodeJS** is an open-source runtime environment that allows execution of JavaScript for server-side applications. NodeJS is event-driven and non-blocking I/O, due to the fact that NodeJS is asynchronous. It was created on top of the V8 engine and it was initially developed by Google and integrated on the Google Chrome browser. It compiles JavaScript source code to native machine code [61].
- **GraphQL** is a query language for APIs, and a server-side runtime for executing queries in order to fetch only the data that is really intended. It was created by Facebook when they started to have scalability problems when rebuilding the Facebook mobile app using the traditional RESTful protocol. A RESTful API is simpler to understand but it's not efficient, when a GET request is sent to the API the client receives also lot of unnecessary data. If the client side only wants, e.g., the phone number of a user, with REST it will receive all of the user info, making it necessary to iterate through the response object to get the information. With GraphQL the client specifies exactly what data it needs, by making use of typed data in order to describe it [62].
- **MongoDB** is a non-relational database, also known as NoSQL database. It doesn't work like the traditional Structured Query Language (SQL) databases like mySQL or Postgres where data is store in a two dimensional row-column structure. MongoDB is document based and supports all types of data, which in turn is stored in a key-value pair topology [63]. All data is stored in a collection that holds multiple documents, that have a unique ID inside the database. The data itself is stored in a Binary JSON (BSON) format, which is a variant of the JavaScript Object Notation (JSON). The BSON structure encodes types and length information allowing it to be parsed more quickly [64].
- **Go** is a open source programming language created by Google that is statically typed and compiled. While JavaScript is an interpreted language, Go

is similar to C where is compiled to native machine code, making it faster and more adequate for heavy tasks, such has machine learning, video editing, big data, and many more [65]. The motivation for the creation of Go was the need to make a less complex language, while at the same time exploit the rise of multicore Central Processing Unit (CPU) devices. The Go language offers support for concurrency, garbage collection and automatic memory management [66].

- **Kotlin** was created by JetBrains and designed for Java Virtual Machine and Android, combining both object-oriented and functional programming features. It is fully compatible with Java and can even be compiled to JavaScript source code. The motivation for the creation of Kotlin was due to the need of having a much simpler language than Java. Instead of creating a direct competitor to Java, Kotlin was created to work alongside it on the development of Android apps [67] [68].

- **Docker** is used to isolate applications inside containers, enabling them to run on any Operating System (OS). When developing any type of application the dependencies change from machine to machine, due to the different OS versions and environments. This fact can lead to a common struggle that developers face when applications only run in their machines. Docker makes it easier to create, deploy and run applications since they are inside containers that provide the needed resources and isolates them from the host OS. Docker can then be seen as a Virtual Machine (VM), but instead of creating an entire virtual OS like VMs do, containers share the host machine OS kernel, thus not needing an OS per container [69].

- **Kubernetes** works alongside Docker and it is used for container orchestration. When deploying an application in a production environment the number of containers running at any given time needs to be managed. The Kubernetes tool ensures that there are no down times and it can scale the services up or down depending on the traffic load. It also has a self-healing mechanism that in an automated way restarts failed containers or even kills containers that don't comply with user-defined health checks [70].

4.3.2 Frontend

The frontend is what the end user interacts with, it will be developed using ReactJS for the browser and with React-Native for the Android box. React and React-Native are JavaScript libraries for building user interfaces that were created by Facebook. It enhances the following features:

- **Component Based**: components are encapsulated and return JSX code, which is a syntax extension to JavaScript. Each component has it own state and it can be stateless too. Moreover, it can be easily reused by other components allowing for a easier development.
- **Virtual DOM**: React has its own virtual Document Object Model (DOM) and compares it with the browser DOM updating only what is really necessary. This allows good interface performances.

4.4 Integration of the IPFS resources

For this project will not be used the official CLI versions of IPFS and IPFS Cluster, since the adopted technologies framework only allows the use of their libraries and APIs embedded in the players. For the web browser player it will be used the **js-ipfs** library running in the browser's service workers, like illustrated in the diagram of Figure 4.6. The IPFS Node should be initialized in the service worker which will then be able to request content from the IPFS network using the IPFS API, previously detailed in Section 3.9. At runtime it will communicate with the Application via Remote Procedure Call (RPC). The node can then make use of the endpoint **file ls** to verify if a file is stored in any of the network peers, and in case it is not, it will fetch it from the AWS S3 Bucket and afterwards add it to the network with the **add** endpoint.

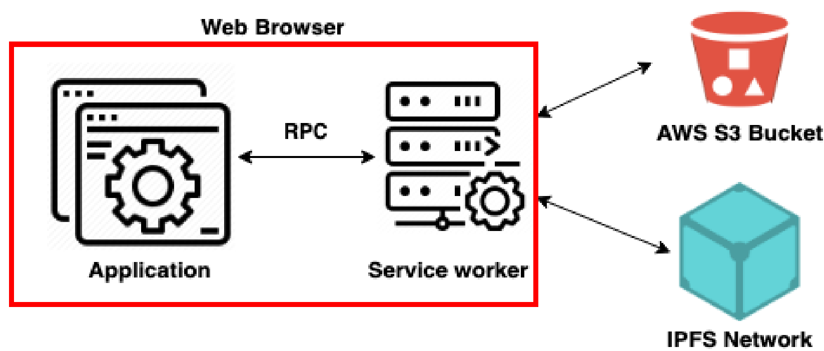


Figure 4.6: Integration of the IPFS in the web browser player

On the other hand, the Android players will make use of the **go-ipfs** and **ipfs-cluster** libraries to run the IPFS Node and the IPFS Cluster. This implies the need to convert the library methods written in Go into Native modules so they can be imported into React. This can be achieved by using a official Go library named **go-mobile** that will allow the code to be used in Kotlin and then be converted into Native modules to be used in React. The workflow depicted in Figure 4.7 shows this conversion process. The Android player will use the same

API endpoint as the browser player to search for content in the network. But to add a file to the network, in this case, it will use the Cluster API by making a request to its **add** endpoint. Instead of just adding the file to the IPFS network, it will also replicate it across all peers in the cluster.

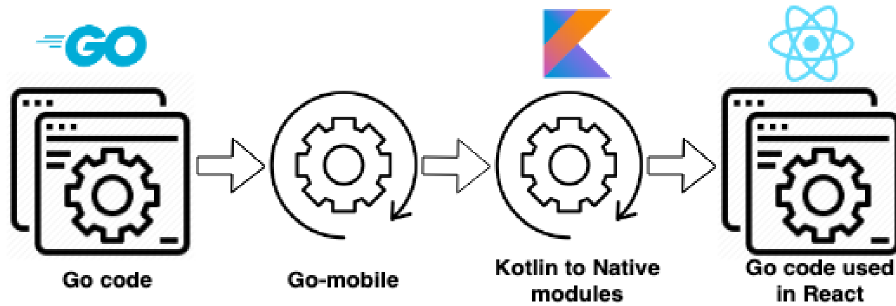


Figure 4.7: Conversion of Go code to be used inside React applications

Chapter 5

Project Development

Knowing the system requirements and restrictions, this chapter will then be devoted to explain all the implementations made in order to achieve the proposed goals. The development included not only the Android boxes players, but also a web browser version of it. The reason for its use will be highlighted in this chapter. It will also be described how it was possible to embed the IPFS libraries writen in Go, into a React-Native application. Finally the most important topic will be addressed: how will the Android boxes share content with each other.

5.1 Broadcast plans

One of the first challenges was the design and development of the broadcast plans related features. A broadcast plan was conceived as simply being an array of objects. Each object contains a list of CIDs that represents the CID of a, from now on called, creative. A creative is an entity that consists of a advertisement or any other content. A plan also includes the duration of the broadcast in milliseconds and the timestamp of the time it's suppose to start playing, and when it should end. The full definition of a broadcast plan is as follows:

- **player_id: ObjectID** - the object ID of the player's document in MongoDB. This identifies the player that requested the broadcast plan.
- **duration: number** - the duration of all ads/content combined, in milliseconds.
- **generated_at: Date** - the timestamp of when the broadcast plan was generated.

- **start_playing_at: Date** - the timestamp of when the broadcast plan is suppose to start playing.
- **finish_playing_at: Date** - the timestamp of when the broadcast plan will end. This value results from the sum of the **duration** and **start_playing_at** keys.
- **campaign_ad_group_id: ObjectID** - the object ID of a campaign document in mongoDB. A broadcast plan needs to be related to a campaign. This entity holds crucial information, like the venue type, geospacial location, timestamp of when it needs to start playing, list of ads/content to be played and so on. Without this information it wouldn't be possible to know which ads/content to select in order to target a given audience.
- **creatives_ids: ObjectID[]** - an array of object IDs of the individual ads/-content documents in MongoDB. Those documents hold metadata of the content: the type of the file (which could be a .jpg or .mp4), the duration of it, the name and its IPFS CID.

The broadcast plans are fully generated by the Vixtape API, based on the user inputs via the dashboard app. After the player gets registered in the back-end system and his status gets defined as ACTIVATED, it then can start sending requests to the API for broadcast plans. The request is made with an argument that sets the number of broadcast plans it needs, considering that a broadcast plan can contain as many creatives as the user wants. However, if the user doesn't select any of the available creatives the Vixtape API, by default, automatically picks a maximum of 5 creatives for each broadcast plan.

Figure 5.1 shows an example of the Vixtape API response to the GraphQL mutation "**playerGenerateBroadcastPlans**". This request was made using the GraphQL playground, which is used for testing and debugging. It allows making any request to any endpoint of the API and see its output. Additionally, it also provides documentation about the API endpoints.

At this early stage in the development it was decided to implement two straightforward algorithms for automatically choosing the content to be included in a broadcast plan. The first one is non targeted, meaning it will randomly select 5 creatives independently of the campaign's audience type. On the other hand the second algorithm is based on the venue type, therefor, it will only choose creatives that make sense for the targeted audience. When a broadcast plan request is made to the API, the decision of which algorithm to use is made randomly. The flowchart of Figure 5.2 illustrates how the first automatic selection algorithm works.

```
{
  "data": {
    "playerGenerateBroadcastPlans": [
      {
        "_id": "5ecfd4814881e46927a601be",
        "creatives_ids": [
          "5ebe81e9a382a18fe403a6d5",
          "5ebe81dba382a18fe403a6cc",
          "5ebe81d1a382a18fe403a6c5",
          "5ebe81d1a382a18fe403a6c6",
          "5ebe81b4a382a18fe403a6bc"
        ],
        "duration": 46590,
        "campaign_ad_group_id":
"5ebe9df0c4b7a49cf05a8721",
        "start_playing_at": "2020-05-
27T18:43:50.000Z",
        "finish_playing_at": "2020-05-
27T18:44:36.590Z"
      },
```


Figure 5.1: Response from the Vixtape API with a generated broadcast plan

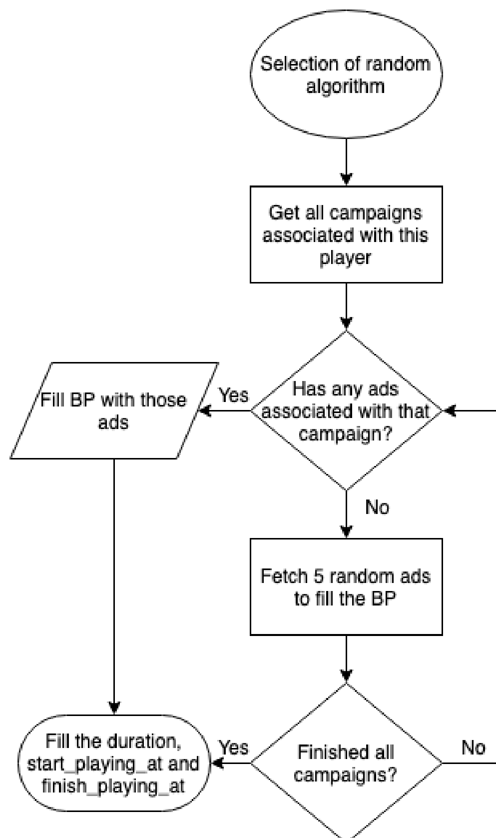


Figure 5.2: Generation of a broadcast plan with random creatives

Regarding the second algorithm, the schema of the collection that holds all creatives documents has a key called **categories_ids**. This key is an array that holds multiple IDs that represent the documents in the categories collection. Additionally, the documents in the venue types collection also have the same **categories_ids** key. So by matching at least one category in a certain creative with a category in the venue type, it's possible to achieve a certain level of audience targeting. For example, consider a campaign stored in the database where the venue type is a restaurant, and that one of its many categories is "alcoholic drinks". If there is also a creative about gin tonic in the database that has the same "alcoholic drinks" category (among others), then this creative can be selected to be part of the broadcast plan. The flowchart of this second algorithm is presented in Figure 5.3.

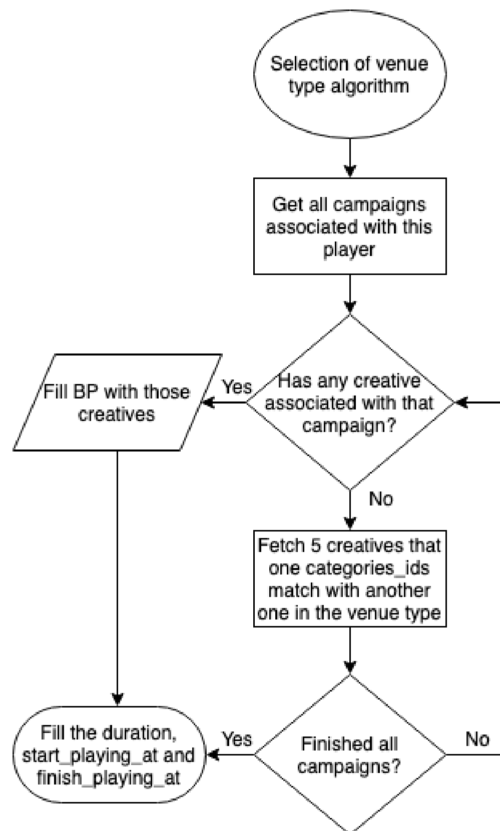


Figure 5.3: Generation of a broadcast plan with the venue type algorithm

The database structure that supports this use case can be seen in Figure 5.4. This model shows how the defined MongoDB collections are related and how all the documents will be divided into small groups. This approach simplifies data aggregation in order to generate a broadcast plan that makes sense for a target audience. Looking to the **BroadcastPlans** collection, one can observe that it is re-

lated to the **CampaignAdGroup** collection that aggregates information related to a campaign which includes, e.g., the set of players for that campaign. A campaign is created by the end user on the dashboard and it consists on selecting the players to be used and further the content to be displayed on them. Then the **Audiences** collection stores the geographic location and the venue type for the targeted audience. The **VenueTypes** collection stores the set of categories that represent the type of categories for a campaign. The list of creatives to be displayed in a campaign, resides in the **Creatives** collection that stores the creative's metadata. Additionally to the automated broadcast plans creation, the end user via the dashboard interface can specify a group of creatives to be added to a campaign.

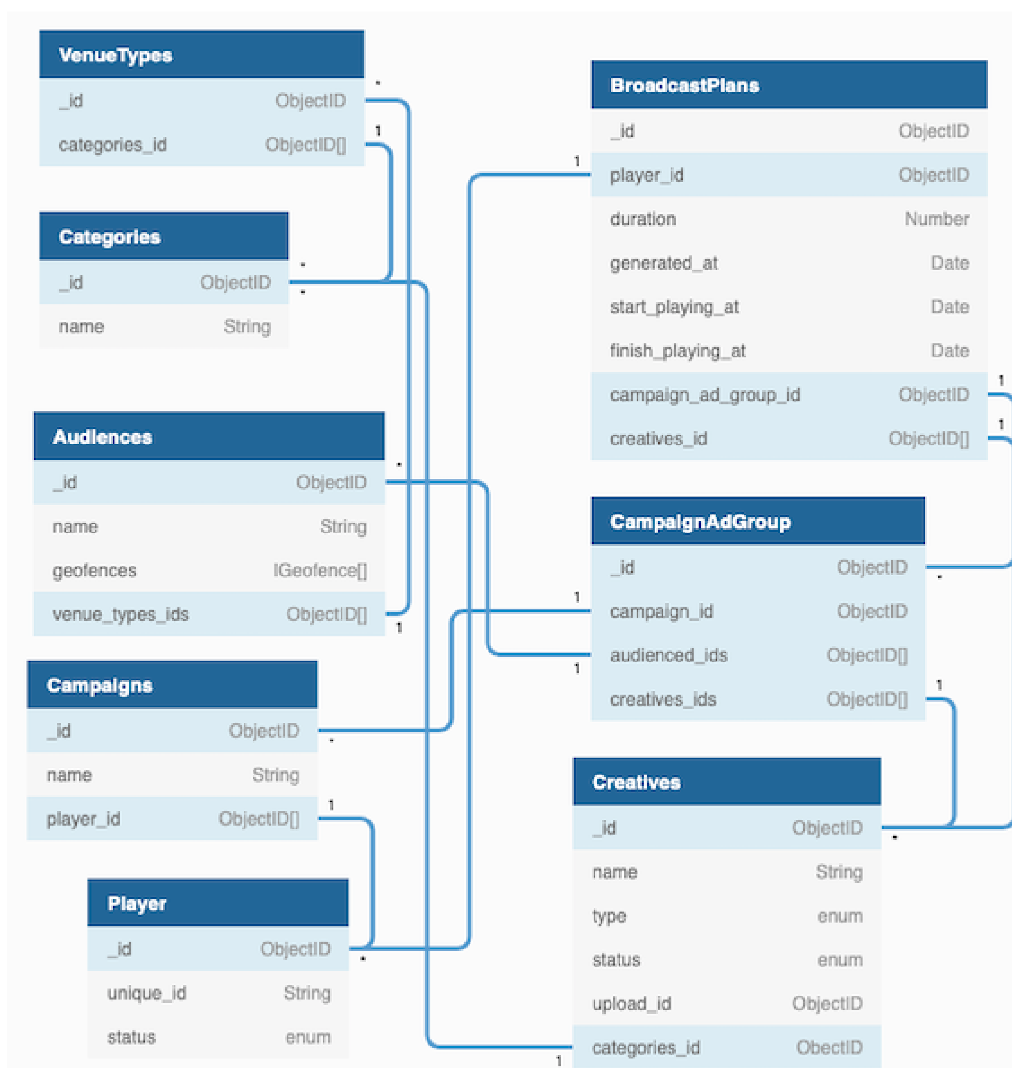


Figure 5.4: Relations between the database collections

5.2 Implementation of the IPFS in the players

There are two types of players, a web version and an Android version. The web player was developed using React and the Android version using React Native, thus making it possible for both versions to share the same core code.

The adopted approach was to make a React component that executes all the player logic. Only when developing the User Interface (UI) components there was the need to write separated components for React and React Native. This happens because React Native does not use HTML to render the app, but instead it uses the native components from iOS or Android UI. To have an agile implementation of these differences, one only needs to deploy two files with the same name but with an additional .native extension on the React Native one, e.g., **empty-screen.tsx** for React and **empty-screen.native.tsx** for React Native. The following two code listings exemplify how it could look like.

File for React named **empty-screens.tsx**:

```
1  export default class EmptyScreen extends PureComponent {
2    render() {
3      return (
4        <div className="App">
5          <body>
6            <div className="wrapper">
7              <div className="logo-container">
8                <Logo className="App-logo" />
9              </div>
10             <div className="text-lorum">
11               There are no ads to show you {':('}
12             </div>
13             <div className="text-bottom">
14               Downloading advertisement content...
15             </div>
16           </div>
17         </body>
18       </div>
19     )
20   }
21 }
```

File for React Native named **empty-screens.native.tsx**:

```
export default class EmptyScreen extends Component {
  render() {
    return (
      <SafeAreaView style={styles.container}>
        <Logo style={styles.logo} />
        <Text style={styles.text}>There are no ads to show
           you {':('} </Text>
        <Text style={styles.textBottom}>
          Downloading advertisement content...
        </Text>
      </SafeAreaView>
    )
  }
}
```

Regarding the integration of the IPFS features into the players, the task posed a new set of challenges. In sum, it is not possible to use the IPFS library in the same way with the browser and Android versions. As a result, two completely different approaches were used with the adoption of two different libraries and two different methods.

5.2.1 Integration of the IPFS in the browser player

The browser version of the content player was, from the start, defined as not being one of the main goals of the project. However, it was developed alongside the Android version due to the fact that both versions can share code in order to achieve the same behavior. Moreover, with the web version it's faster to run the code and to perform tests.

To run a IPFS node in the web browser, a service worker was used. A service worker is a browser component that allows to run scripts on the background, it can be used for push notifications or even make HTTP requests. This service worker is then used to run the IPFS Node and to execute all the functionalities needed for communicating with the player's core logic via Remote Procedure Call (RPC). Instead of the traditional HTTP request/response, the RCP makes use of Event Listeners that receive instructions through messages.

That being said, the library **js-ipfs** [71] was used in order to interact with the IPFS network from within the web player. Additionally, if the requested content is not found in the IPFS network, the player will do a normal HTTP request to fetch it from the AWS S3 Bucket. The logic used to get content from the IPFS network is shown in the flowchart of Figure 5.5.

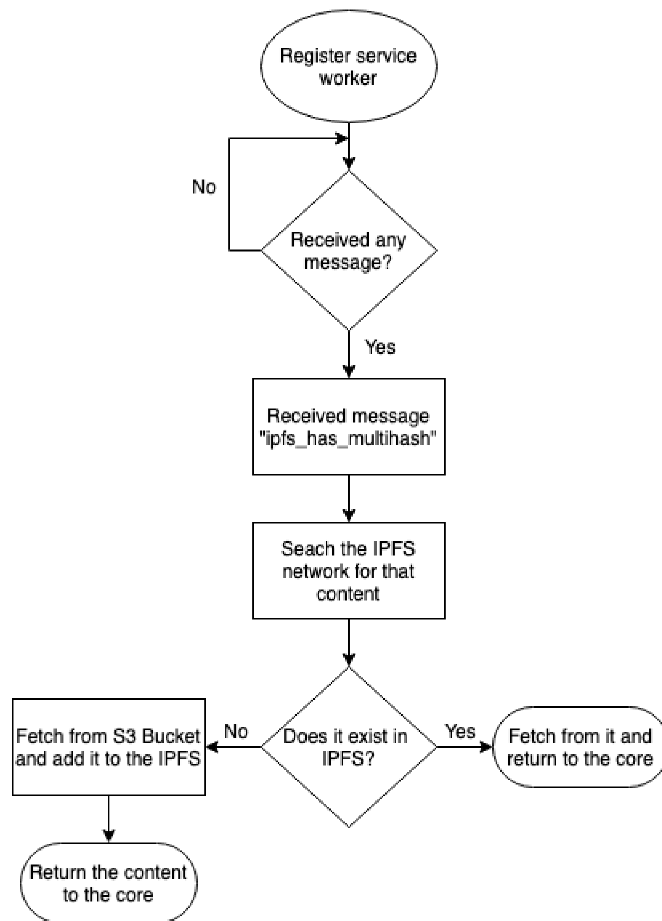


Figure 5.5: Algorithm used by the web browser player to retrieve the needed content

5.2.2 Integration of the IPFS in the Android player

For the React Native version of the player running on a Android box the approach needs to be completely different. Mainly because the player app will not run on a browser, thus not having the possibility to use a service worker. In order to integrate the IPFS Go library into React Native, it first needs a few modifications. Although there is a IPFS JavaScript library readily available, this project uses the **go-ipfs** [72] library. The reasons being the following:

1. Go is better than JavaScript for intensive computation tasks [73].
2. Go is faster than JavaScript due to the fact that JavaScript is an interpreted language, while Go is a compiled light-weight language.
3. The **go-ipfs** library is more tested and stable than the **js-ipfs** library. The official IPFS version is written in Go, while the JavaScript version is just an

adaptation to be used in browsers and on NodeJS APIs.

4. There is no JavaScript version for the IPFS Cluster, there is only a Go version.

In order to run Go code in React Native a bridge between Go and JavaScript was created. The entire procedure was achieved by making use of a package developed by the official Golang team, named Go Mobile [74]. This software provides a way to write Go packages that can be used with Android and iOS. It generates language bindings for Java, if the target OS is Android, or Objective-C, in the case of being iOS. With the Go package ready, the following command is used to create the Go Mobile package:

```
>> gomobile bind -v \
     -target android \
     -o ./build/android.aar \
     -classpath net.realtimeads.ipfs \
     ${GO_MODULE};
```

After executing the previous command the package named **net.realtimeads.ipfs** is yet not ready to be imported into the React Native project. For that to be possible the methods created need first to be converted into Native Modules so they can to be used with React Native. This step is done using Kotlin as a middle layer. By creating a class that extends **ReactContextBaseJavaModule** [75] with the methods imported (as public) from the Go package, it is then possible to import the class into React Native and use its public methods. This two-step process is exemplified in the following code snippets that show how the Native Modules are created with Kotlin and afterwards, how they can be imported into React Native.

The first code listing refers to the Kotlin file where the package **net.realtimeads.ipfs** is included and the IPFS module is imported. After that it exemplifies how a method gets exposed. In this case, the code **Ipfs.hasFile()** calls a method imported from the Go package that verifies if a certain content exists in the IPFS network.

Finally the code can then be imported into React Native and the functions written in the Go package can be used. This is exemplified in the second code listing, with the call to the function **hasFile()**.

Example of the first step:

```
    package net.realtimeads.ipfs
    ...
    import ipfs.Ipfs

    class RNIpfsModule(var reactContext:
       ReactApplicationContext):

    ReactContextBaseJavaModule(reactContext) {
    ...
        @ReactMethod
        public fun hasFile(multihash: String, promise:
           Promise) {
            Thread(Runnable {
                try {
                    var multihash = Ipfs.hasFile(multihash)
                    promise.resolve(multihash)
                } catch (err: Exception) {
                    promise.resolve(false)
                }
            }).start()
        }
    }
```

Example of the second step:

```
import { NativeModules } from "react-native";
const { RNIpfs } = NativeModules;

    export async function preload(multihash: string,
       storageBucket: string) {
      if (await RNIpfs.hasFile(multihash)) {
        return;
      }
      ...
    }
```

5.3 Configuration of a IPFS node

Independently of where the IPFS datastore path is located in the file system, it holds multiple files and folders. One of them is the **config** file, that contains the IPFS configuration in JSON format. The configuration file consists of a big nested set of objects whose primary keys, and a small explanation, are presented in the following list:

- **Addresses:** holds the addresses of various listeners such as:
  - **API:** default value is /ip4/127.0.0.1/tcp/5001.
  - **Gateway:** default value is /ip4/127.0.0.1/tcp/8080.
  - **Swarm:** default values is /ip4/127.0.0.1/tcp/4001, /ip6/::/tcp/4001, /ip6/0.0.0.0/udp/4001/quic and /ip6/::/udp/4001/quic. The Swarm port is used to listen for P2P connections from other nodes.
- **API:** set of HTTP headers for the responses.
- **Identity:** has the following sub keys:
  - **PeerID:** is a unique ID attributed to this node. It is used by outside nodes to find him.
  - **PrivKey:** is the node private key used for encryption, it's a base64 encoded protobuf.
- **AutoNAT:** contains the configuration options for the AutoNAT service. It helps other nodes on the network determine if they are publicly reachable from the rest of the Internet.
- **Bootstrap:** is an array of multiaddrs of trusted nodes to connect on initialization. If not specified the default value is an array of various ipfs.io bootstrap nodes.
- **Datastore:** has information related to the construction and operation of the on-disk storage. Like the maximum storage size, the percentage of the max storage size which will trigger a garbage collection and many more.
- **Discover:** has options for configuring the node discovery mechanisms, it focus around mDNS. If it is enabled or not, and the interval between discovery checks.
- **Gateway:** multiple options for the HTTP gateway, like the HTTP headers of the gateway responses, among others.
- **IPNS:** holds configurations for the IPNS, like the republish period which is the frequency it republish IPNS records to stay up to date in the network and other ones.
- **Mounts:** is the Filesystem in USEspace (FUSE) mount point configuration options:
  - **IPFS:** mountpoint for /ipfs.
  - **IPNS:** mountpoint for /ipns.

  - **FuseAllowOther:** sets the FUSE to allow other option on the mount-point.

- **Pubsub:** is used for the configuration of the IPFS pubsub subsystem:
  - **Router:** sets the default router used by pubsub to route messages to peers. There is "floodsub", which floods messages to all connected peers, which is very reliable but inefficient. The default option "gossipsub", is a more robust routing algorithm that will build an overlay mesh from a subset of the links in the network.
  - **DisableSigning:** disables message signing and signature verification.
  - **Peering:** the peering subsystem is used to remain connected to, and reconnect to, a set of nodes. This is used to improve reliability by creating links between frequently useful peers. It holds one key named **Peers** and is an array of objects that consists of the "ID" of a node and the "Addrs" it addresses.

- **Reprovider:** holds various configurations about the reprovider. It is used to announce the available data to the other nodes:
  - **Interval:** the time interval between announcements.
  - **Strategy:** there are three strategies available to choose. The "all" strategy will announce all stored data, "pinned" will only announce pinned data and "root" will only announce directly pinned keys and root keys of recursive pins.

- **Routing:** holds one key names "Type". It can be defined as "dht" or "none". If set to "none" there is no way for the node to be connected to the IPFS network, thus when trying to fetch some specific data it is needed to know before hand who has that data. If defined as "dht", which is the default one, the node uses the IPFS DHT in order to discover and be discovered by other nodes.

- **Swarm:** the swarm configuration is the biggest one with 24 keys. It is used for network configuration such as transport configuration, addresses filtering, bandwidth metrics, connection manager configuration and some more.

After this summed up overview of the IPFS node configuration parameters, it will now be detailed how the features added to the Vixtape API can change the configuration of any of the players (i.e., the IPFS nodes) during runtime and without any content displaying interruption.

In order to explain how this is achieved, firstly the used technologies stack must be detailed. One of the elements used is what it's called a State Manager, named MobX [76], this is used in React apps in order to store the state of a running app and make it accessible to any other component. The state in a React app is what drives the application, where the current state is what defines what the current view should be, and holds any type of information that can be used for any matters. The ads player saves in it's state the player's model that contains all its information, some of those attributes are the following:

- **unique_id:** is a unique ID generated by the API and attributed to the player.
- **STATUS:** is a enum type, and it can be one of the following: "REGISTERED", "BINDED", "ACTIVATED", "DEACTIVATED", "ARCHIVED". A player only displays content after being set as "ACTIVATED".
- **venue_types_id:** this item was already addressed in Section 5.1 and it refers to one of the broadcast plans parameters. It's an array containing which venue types IDs the player is inserted into.
- **secret:** is a generated token used for authentication. Without this token the player will not have access to the backend API.
- **created:** the date when the current player was created.
- **binded:** the date when the player was binded by the end user.
- **ipfs:** the ipfs attribute is the most important one in this context. This key is a copy of the entire IPFS **config** file.

These are only some of the player's attributes. The full set of attributes is also sent by the player to the backend database and stored on its document, in the **Player** collection. When changing any value of the **ipfs** object of a specific player in the database, one must ensure that those changes also happen in player's local state data. To achieve that, MongoDB offers the Change Streams [77] feature that monitors changes performed in a collection. After a value in the **Player** collection gets updated, the API receives the changed document and triggers a GraphQL subscription. A GraphQL subscription is the third possible operation in addition to queries and mutations. Subscriptions are made to only fetch data, but unlike queries where the client needs to formally request it, subscriptions maintain an active connection between client and server via WebSockets. This enables to push updates to the clients when a certain event happens.

After defining a subscription between the player and the API, every time the database player document is changed the subscription will be triggered, thanks

to the MongoDB Change Streams feature. This way, the player is now able to receive via WebSocket the updated player model and replace its local configuration. This method is also employed to change the **STATUS** attribute of the player.

The player after receiving a new set of attributes compares the local **ipfs** object with the new one, in order to assess if any changes to the IPFS configuration were made. In case the two objects are different, then it updates the configuration by calling the functions from the service worker or from the Go packages, depending on the player version. There are some attributes in the IPFS **config** that require a node restart to be applied. However, since the player only compares if there is any difference between the objects as a all, and not the individual changed attributes, the IPFS node will always performs a restart. While the node restarts, the content that is being displayed keeps running with no interruptions. Furthermore, a IPFS node restart cycle only takes about 10 seconds. The diagram presented in Figure 5.6 illustrates the previously described sequence.

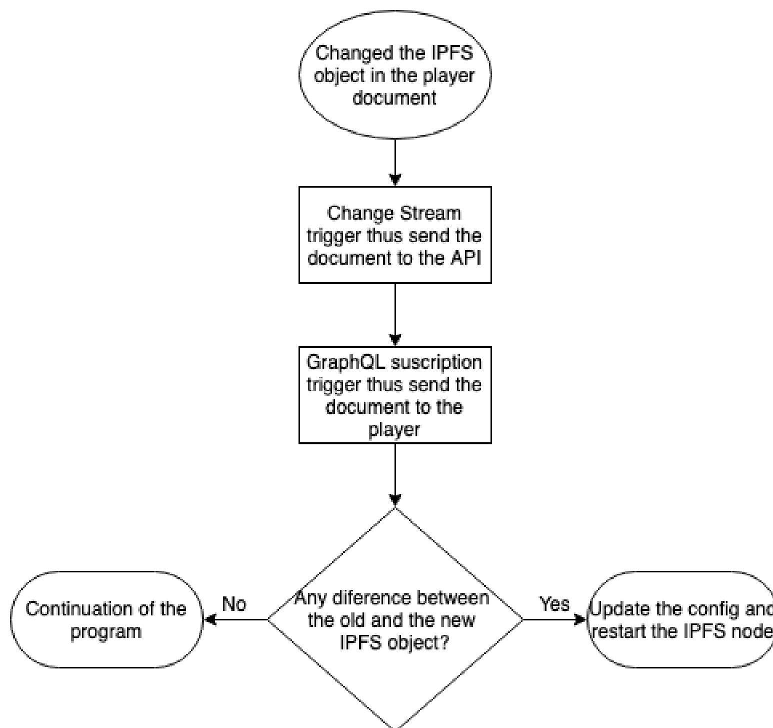


Figure 5.6: Update procedure of the IPFS node configuration

5.4 IPFS Cluster implementation

The creation of a IPFS cluster can only be done after having the IPFS nodes in place. The IPFS Cluster tool enables the players to share content between

them without the need of a formal request, because every time a player in the cluster has a new file, that file will be propagated to every other player inside the cluster. This will not only improve service reliability, by not repeating requests of content that other players already have in memory, but also will reduce the overall costs of running the Vixtape service.

5.4.1 Cost efficiency of a IPFS cluster

Discarding the infrastructure, backend API and dashboard costs, it's not a straightforward task to predict the exact costs of running the players. First the costs will depend a lot on the end user usage, because some may never turn the Android box off while others may only have them online for few hours a day. Secondly, the content data size varies, which makes the cost depend on the length and quality of the media files. Lets assume a scenario where this project does not make use of IPFS nor IPFS Cluster, thus every time the player needs content it needs to get it from the AWS S3 Bucket. A study made by Mosano concluded that in this scenario the costs would be something around 8 €/month, per box. Considering that this project has worldwide deployment ambitions, in the case of having a million boxes alive at any given time during the day, that would translate into a considerable amount of money. Most likely turning the Vixtape project too expensive to be commercially competitive. The integration of the IPFS Cluster technology will greatly reduce the need of the players to get content from the AWS S3 Bucket, which saves bandwidth and consequently the associated the costs. Moreover, it also reduces the need to fetch content from the public IPFS Network. While this hasn't any direct monetary costs, the used bandwidth has.

The end user can create a network of boxes, which consists of a set of Vixtape boxes that will display equivalent content. Thus instead of having, e.g., 10 boxes independently fetching the same content at the same time, one of them will fetch the new content and add that file to the cluster. After that, the other 9 boxes will have that content available in memory and don't need to waste any bandwidth to get it. In the end, the more boxes work in sync, the more efficient the cluster will be on the long run.

5.4.2 Integration of the IPFS Cluster in the players

In Section 5.2.2 it was explained how the IPFS Node was integrated with the Android player. As mentioned also in that section, the IPFS Cluster is only available in Go, thus making this technology only feasible to be used with the Android Player and not with the browser version. As expected, the adopted method in this case was exactly the same as it was with the IPFS node. The

official **ipfs-cluster** library [78] was used and integrated with the previous developed IPFS node code.

In order for the cluster peers to be able to connect to each other they must share the same *secret*, which is a 32-byte hex-encoded string. This prevents outside peers from joining the private cluster and change the pinset. The other key aspect is that they must use the same configuration file. This file is JSON formated and named **service.json**, it holds the entire configuration of the cluster including the secret itself. To share this information across the cluster we have three options. The first is by uploading the file into the cloud to make it available for the peers to download it. The **ipfs-cluster** library has a method called **LoadJSONFromHTTPSource** that takes the configuration file URL as argument. The second method, **LoadJSONFromFile** accepts a local path for the file. Finally, the third one is by using the method **LoadJSON** which takes a JSON object as an argument. This last one is used because it avoids the dependence of a centralized cloud service to store the cluster configuration file. For this method to work, a configuration file is provided and parsed in order to change certain values. Then the parsed JSON is loaded using the third method. The cluster also has a file named **identity.json**, which holds the values for its cluster peer ID and its private key.

Using the CLI version of the IPFS Cluster, the first step to start a cluster peer is to use the command **ipfs-cluster-servive init** to initialize the **service.json** and **identity.json** files. Then the command **ipfs-cluster-service daemon** launches the cluster peer itself. This command accepts an important flag **–bootstap <peer-multiaddress1,peer-multiaddress2>** that specifies to which peers it should connect to at boot. But since the CLI tool of the IPFS Cluster is not a viable integration for the Android player, the **ipfs-cluster** library was used to code the initialization and launch of the peers.

To do the configuration initialization a data store path for the **service.json** and **identity.json** files needs to by defined. This path was set as **"./cluster/config"** in this case. Then, if the **service.json** or **identity.json** files don't exist, they must be generated. For this, an **ID** and **private_key** keys for the **identity.json** file are generated by the library. On the other hand, for the **service.json** file a **cluster.peername** key needs to be unique. So before it can be generated, this key is defined by a random peer name like **"vixtape-cluster-<randomID>"**, where **<randomID>** is a 9-digit random string. In the unlikely scenario of equal **<randomID>** strings, there will be no problem because the **cluster.peername** is used only for the cluster to be easily identified by humans. The cluster peers use their unique IDs to communicate between them and not the **cluster.peername**. To finish the initialization it is then needed to generate an empty file on the same path as the **service.json** and **identity.json** named **peerstore**. This file will be updated by the cluster itself with known peer addresses. The file can also be pre-filled

with other peer multiaddress to help the cluster connect to those peers, however for this case, those multiaddress will be specified in the key **peer_addresses** in the **service.json** file. Figure 5.7 shows the diagram that sums up the initialization workflow.

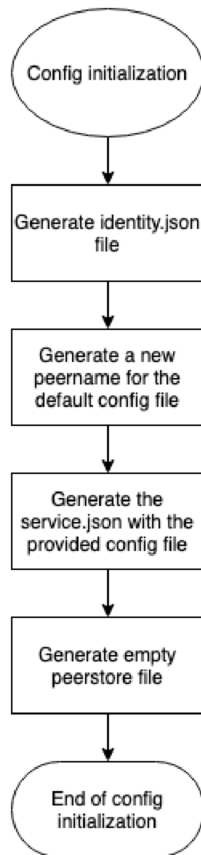


Figure 5.7: Cluster peer configuration initialization workflow

Only after the data store path and the configuration files are initialized, the cluster peer can be started. The first thing is to create a P2P host, using the library **libp2p** [79]. This is a popular library modularized out of IPFS that can be used in different projects that require a P2P solution. When creating a cluster host it returns: pubsub, DHT and host instances that persist in the datastore for shared use by all cluster components. The host will make use of the DHT for routing. Then it's instantiated a disk informer and the tracing tools based on the configuration in **service.json**.

Defining the consensus is one of the most important parts of the cluster, the consensus needs to be the same across all peers. For this project the CRDT consensus was chosen. To setup this consensus it is needed to have the host, pubsub and DHT instances returned by the **libp2p**, and the configurations of the CRDT.

The CRDT configuration has a key named **trusted_peers** that holds the multiaddresses of the peers allowed to share data. However, in this case it was passed the value **" * "** that makes a peer trust all the other peers in the cluster.

The cluster peer and the IPFS node need a way to communicate, that is done by both using the other's API. The IPFS Cluster API runs on port 9094 and needs to communicate with the IPFS Node API on port 5001, thus a connector between them was created. The API configuration of the cluster holds information about the headers and Cross-Origin Resource Sharing (CORS) policy configuration. Once that is initialized, it is then created a connector component between the cluster API and the IPFS node API for them to communicate.

The last step is to initialize a stateless tracker. The configuration of so has two keys: **max_pin_queue_size** which is the maximum pin or unpin requests that can be queued before throwing an error, and the **concurrent_pins** that is how many parallel pins and unpins requests can be made to the IPFS. Once all is instantiated, it's all then passed as arguments to the method called **NewCluster** that will initialize the cluster peer.

Chapter 6

Tests and Results

In this chapter it will be demonstrated the outcome of the previously developed IPFS features in an environment created to share content within a cluster. The app that starts each IPFS node and cluster peer was not tested while running on the Android boxes due to hardware availability limitations at the time of the tests schedule.

6.1 Manual testing and simulation

The code follows the simple logic of first initializing the IPFS Node and, if it succeeds, then starting the IPFS Cluster functionalities. Initially, to create a manageable testing setup a network was created using Docker, named **vixtape-cluster-network**. For this, the following command was used:

```
>> docker network create --driver bridge --subnet 172.20.0.1/24
--gateway 172.20.0.1 vixtape-cluster-network
```

Then, the following step is to run the docker container:

```
>> docker run -d -p 9194:9094 -p 9196:9096
--network vixtape-cluster-network
--name bootcluster poc-ipfs-cluster
```

This command starts a docker container and exposes its port 9094 (i.e., the Cluster API) through port 9194 of the host machine. This way every request that arrives at <host_machine_IP>:9194 will be redirected to port 9094 of the running container. The same is also done for the Gateway using the ports 9096 and 9196.

The flag **--network** defines the network to be use, while the flag **--name** gives a name to the container. The last argument is the image to run in the container.

The **poc-ipfs-cluster** accepts an environmental variable named **ADDR** that sets the address of another cluster peer to bootstrap with when starting up. In our case it's not being used because the test cluster peers will be running inside the same network and can auto discover themselves with mDNS. Nevertheless, all it would be necessary was to add the flag **-e ADDR=<peer-addr>** to the command line. For this manual test kubernetes was also not used, hence this process was repeated manually two more times in order to setup three testing peers. Finally, a fourth peer outside the defined network is also used for testing purposes. Later on, another peer will be added belonging to a new network, because with Vixtape it's expected for peers to join and leave frequently.

An illustration of how the peers become arranged with this testing setup is shown in Figure 6.1. The names inside the circles are the last 4 characters of the used **peername**, they are respectively: "Mosano", "vixtape-cluster-T69eKGIMR", "vixtape-cluster-czkvcMIMR" and "vixtape-cluster-oVtNcMSGg". The peer in the Host network is not running the developed Vixtape cluster code, instead it's running the official and stable version of the IPFS Cluster. The reason of this is testing and debugging convenience, since this peer supports the use of the IPFS CLI commands. With this version of the IPFS Cluster the **ipfs-cluster-ctl** module is separated from the core, making it possible to use it with the developed cluster version. It basically performs requests to the Cluster API, to the default source at **/ip4/127.0.0.1/tcp/9094**, and prettifies the output logs. As previously stated, by making a request to port 9194, it gets redirect to the running container. Therefore, to setup this mechanism the flag **-host** is used in order to define to whom it must send the request, e.g., **ipfs-cluster-ctl –host /ip4/127.0.0.1/tcp/9194 <COMMAND>**.

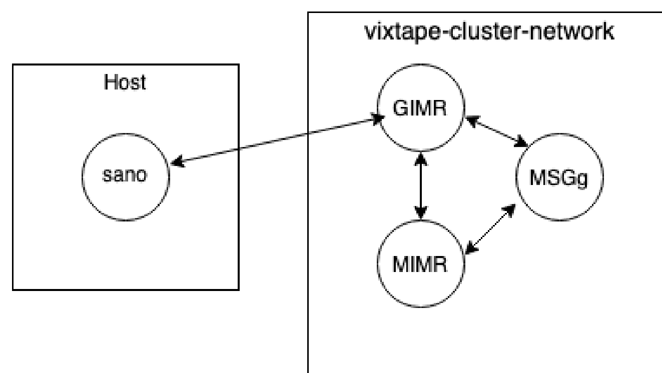


Figure 6.1: Peers connected to each other after they startup

For the peer in the Host network to bootstrap to the "GIMR" peer and join the cluster, the following command is used:

```
>> ipfs-cluster-service daemon --bootstrap /ip4/127.0.0.1/tcp
/9196/p2p/12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek
```

The "sano" peer now belongs to the cluster. To verify this, the command **peers ls** is used as follows:

```
>> ipfs-cluster-ctl peers ls

-12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE |
ERROR: failed to find any peer in table
-12D3KooWQGabfmCNorimaQNSzEmhRDtcEy2MoPtZW34aJtd1cTSc |
ERROR: failed to find any peer in table
-12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek |
vixtape-cluster-T69eKGIMR | Sees 3 other peers
> Addresses:
- /ip4/127.0.0.1/tcp/9096/p2p
/12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek
- /ip4/172.20.0.2/tcp/9096/p2p
/12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek
> IPFS: QmZwMtrdRiZsBG6diULCrLHePcFBk81xTxtUJDf9ST6rxc
- /ip4/127.0.0.1/tcp/4001/p2p
/QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
- /ip4/188.82.83.87/tcp/41001
/p2p/QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
- /ip4/192.168.1.78/tcp/4001
/p2p/QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
```

The output of the **peers ls** command shows that the first two peers returned an error. This was due to the fact that the "GIMR" peer reported how many peers it sees and who they are. In this case the other peers are running inside docker and the exposed ports to the host are 9296 and 9396, instead of 9096. So the peer asking does not know of that and tries to communicate using the port 9096, which of course will return an error. This can be proved by making a **peers ls** from a peer located inside the docker network.

```
>> ipfs-cluster-ctl --host /ip4/127.0.0.1/tcp/9194 peers ls

-12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE |
vixtape-cluster-czkvcMIMR | Sees 3 other peers
> Addresses:
- /ip4/127.0.0.1/tcp/9096/p2p
```

```
/12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE
- /ip4/172.20.0.4/tcp/9096/p2p
/12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE
> IPFS: QmPQmGTxbUFBXUQAfegLpRbTZKGz5f6sFQ7H3hZXUnssyP
- /ip4/127.0.0.1/tcp/4001/p2p
/QmPQmGTxbUFBXUQAfegLpRbTZKGz5f6sFQ7H3hZXUnssyP
- /ip4/172.20.0.4/tcp/4001/p2p
/QmPQmGTxbUFBXUQAfegLpRbTZKGz5f6sFQ7H3hZXUnssyP

12D3KooWQGabfmCNorimaQNSzEmhRDtcEy2MoPtZW34aJtd1cTSc |
vixtape-cluster-oVtNcMSGg | Sees 3 other peers
> Addresses:
- /ip4/127.0.0.1/tcp/9096/p2p
/12D3KooWQGabfmCNorimaQNSzEmhRDtcEy2MoPtZW34aJtd1cTSc
- /ip4/172.20.0.3/tcp/9096/p2p
/12D3KooWQGabfmCNorimaQNSzEmhRDtcEy2MoPtZW34aJtd1cTSc
> IPFS: QmbdaKUJ4KEHo6AFz18bx3DdyVDy2enq9NxPUxghRPD5ax
- /ip4/127.0.0.1/tcp/4001/p2p
/QmbdaKUJ4KEHo6AFz18bx3DdyVDy2enq9NxPUxghRPD5ax
- /ip4/172.20.0.3/tcp/4001/p2p
/QmbdaKUJ4KEHo6AFz18bx3DdyVDy2enq9NxPUxghRPD5ax

12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek |
vixtape-cluster-T69eKGIMR | Sees 3 other peers
> Addresses:
- /ip4/127.0.0.1/tcp/9096/p2p
/12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek
- /ip4/172.20.0.2/tcp/9096/p2p
/12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek
> IPFS: QmZwMtrdRiZsBG6diULCrLHePcFBk81xTxtUJDf9ST6rxc
- /ip4/127.0.0.1/tcp/4001/p2p
/QmZwMtrdRiZsBG6diULCrLHePcFBk81xTxtUJDf9ST6rxc
- /ip4/172.20.0.2/tcp/4001/p2p
/QmZwMtrdRiZsBG6diULCrLHePcFBk81xTxtUJDf9ST6rxc

12D3KooWRVXgd25qgdBVzzeFx9SksVMReRSrUJQQ91W6q91mL7QY |
Mosano | Sees 3 other peers
> Addresses:
- /ip4/127.0.0.1/tcp/9096/p2p
/12D3KooWRVXgd25qgdBVzzeFx9SksVMReRSrUJQQ91W6q91mL7QY
- /ip4/127.0.0.1/udp/9096/quic/p2p
/12D3KooWRVXgd25qgdBVzzeFx9SksVMReRSrUJQQ91W6q91mL7QY
- /ip4/192.168.1.78/tcp/9096/p2p
/12D3KooWRVXgd25qgdBVzzeFx9SksVMReRSrUJQQ91W6q91mL7QY
- /ip4/192.168.1.78/udp/9096/quic/p2p
/12D3KooWRVXgd25qgdBVzzeFx9SksVMReRSrUJQQ91W6q91mL7QY
> IPFS: QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
```

```
- /ip4/127.0.0.1/tcp/4001/p2p
/QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
- /ip4/188.82.83.87/tcp/41001/p2p
/QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
- /ip4/192.168.1.78/tcp/4001/p2p
/QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
```

Now that the cluster is set up, the "sano" peer will add an .mp4 file with 11,8 MB that will be propagated to all peers in the cluster:

```
>> ipfs-cluster-ctl add ../PexelsVideo.mp4

added QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t PexelsVideo.mp4
```

Running the command **status** returns if the file is pinned or not in each peer:

```
>> ipfs-cluster-ctl
status QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t

QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t :
> 12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE :
CLUSTER_ERROR: failed to find any peer in table
> 12D3KooWQGabfmCNorimaQNSzEmhRDtcEy2MoPtZW34aJtd1cTSc :
CLUSTER_ERROR: failed to find any peer in table
> vixtape-cluster-T69eKGIMR : PINNED
> Mosano : PINNED
```

Once again the "sano" peer does not have access to the two peers that previously returned an error. This is an expectable situation that might occur when some peers may not be able to see other ones due to network restrictions. But that doesn't mean that the file did not propagate to those peers. As mentioned in Section 3.8.1.2, with the CRDT state based consensus the peers will send the file to one known peer, than this peer does the same so that in the end all peers will share the same state. So the "sano" peer will send to a peer to whom it can communicate, and then that same peer will send to another. Running the same command but in a different peer can prove such behavior:

```
>> ipfs-cluster-ctl --host /ip4/127.0.0.1/tcp/9194
status QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t

QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t :
> vixtape-cluster-czkvcMIMR : PINNED
> vixtape-cluster-oVtNcMSGg : PINNED
> vixtape-cluster-T69eKGIMR : PINNED
> Mosano : PINNED
```

The next test shows the result of adding another file, from one of the peers that the "sano" peer is not able to see. The file used is simple .png image:

```
>> ipfs-cluster-ctl --host /ip4/127.0.0.1/tcp/9294 add ../icon.png
added QmbtHgxAWz1ghG7uSrYjqYCNaNz4SF91i1BVMnRTaXAWVa icon.png

>> ipfs-cluster-ctl status QmbtHgxAWz1ghG7uSrYjqYCNaNz4SF91i1BVMnRTaXAWVa

QmbtHgxAWz1ghG7uSrYjqYCNaNz4SF91i1BVMnRTaXAWVa :
> 12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE :
CLUSTER_ERROR: failed to find any peer in table
> vixtape-cluster-oVtNcMSGg : PINNED
> vixtape-cluster-T69eKGIMR : PINNED
> Mosano : PINNED
```

Once again, using the **status** command from the "sano" peer it shows that now he knows the **peername** and that the file is pinned from the peer who added it. This happens because the peers know who added the files, thus storing this information in their DHT even though the peer that added it is unreachable from the "sano" peer.

Since the cluster works as a pin orchestrator to the IPFS Node and a data replicator, the files are not accessed from the IPFS Cluster itself but from the IPFS. Running now the IPFS CLI it's possible to see from the DHT of the peer who can retrieve a specific object, this time the .mp4 file:

```
>> ipfs dht findprovs QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t

QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
QmPQmGTxbUFBXUQAfegLpRbTZKGz5f6sFQ7H3hZXUnssyP
QmZwMtrdRiZsBG6diULCrLHePcFBk81xTxtUJDf9ST6rxc
QmbdaKUJ4KEHo6AFz18bx3DdyVDy2enq9NxPUxghRPD5ax
```

The output shows all the IPFS IDs that belong to all peers of the cluster. This means that all of them have in their memory the file and are ready to distribute it. By using the IPFS official website [80] it's possible to access data from the IPFS public network by typing in a web browser **ipfs.io/ipfs/<CID>** where **<CID>** is the CID of the .mp4 file added to the cluster. This shows that one does not need to have a running IPFS instance in order to get content from the network. Figure 6.2 shows the .mp4 file being played on a browser.

The last test serves to demonstrate how to add to the current cluster a peer belonging to a different network. The representation of so can be seen in the Figure 6.3, where the new peer is called "iSGR". To keep the figure simple not

Figure 6.2: Accessing the content from the browser

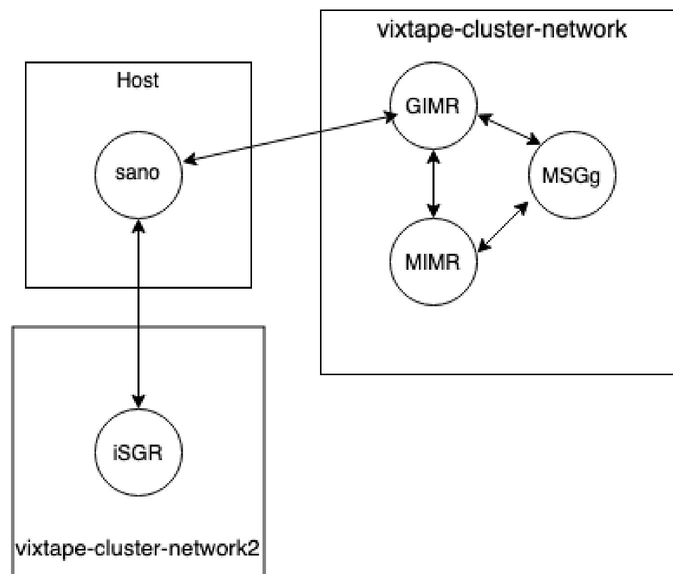


Figure 6.3: New peer added on the **vixtape-cluster-network2** network

all connections are represented since peers are all connected to each other (recall Figure 4.3).

Bootstrapping the new peer to the "sano" peer will make him join the cluster. This means that for any new peer to join a cluster, all it's needed is to bootstrap it to any of peers that currently belong to the cluster. Since the "iSGR" peer joined after the .mp4 file was propagated, running the **status** command from it shows the following:

```
>> ipfs-cluster-ctl --host /ip4/127.0.0.1/tcp/9494
status QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t

QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1t :
> 12D3KooWLKPD8ghygHwMC3kHyJMGhRYGanDJVx19XKTCCAk4RyBF : REMOTE
> 12D3KooWN6Af2D5hJ3kLoC9TohR34U4aTZKtrrno26SoijKbVuhE :
CLUSTER_ERROR: failed to find any peer in table
> 12D3KooWQGabfmCNorimaQNSzEmhRDtcEy2MoPtZW34aJtd1cTSc :
CLUSTER_ERROR: failed to find any peer in table
> 12D3KooWQcWwCsBinYBckyE3iD1NuBCPZQkcsacYkSgh6p3Abkek :
CLUSTER_ERROR: failed to find any peer in table
> Mosano : PINNED
```

The output prints that the file is not **PINNED** but is **REMOTE** instead, meaning that the pinned file is allocated in another cluster peer. In other words, it does not have the file pinned in memory but it knows from whom to get it from. To finalize, it can be seen once again by running the **dht findprovs** command that this new peer is included in the tables, even without having the content in memory. But since he can inform any peer that makes a content request where to find it, he gets included in the DHT of all peers.

```
>> ipfs dht findprovs QmZXyabqkV67TZBRnjM5BYUZb5JcsKzUW5oQ3T4SW8Cr1

QmZUPJZk7d9FmPdih38tZ1XLB5PL2rB8uV5tsJ3zwuvmuQ
QmPQmGTxbUFBXUQAfegLpRbTZKGz5f6sFQ7H3hZXUnssyP
QmZwMtrdRiZsBG6diULCrLHePcFBk81xTxtUJDf9ST6rxc
QmbdaKUJ4KEHo6AFz18bx3DdyVDy2enq9NxPUxghRPD5ax
QmaTjP21k6Yeid2VrZw4P3BXtLg22fmBMyg3p7A2Mcnq8m
```

6.2 Unit Testing

Unit testing is where specific parts of the code are tested so ensure that the functions work as expected in all possible scenarios. The goal is to make sure that the a running IPFS node can search for the needed content and that the a cluster peer can add and remove it from the cluster.

In the language Go to make a unit test the file name must contain **"_test.go"** in the file's name. Then the package "testing" needs to be imported to validate the functions. In this case the logic behind the functions are not too elaborate since they all look as follows:

```
1  package main
2
3  import (
4    "testing"
5  )
6
7  func Testfunction(t *testing.T){
8      if _,err := FunctionToTest(arguments); err != nil {
9          t.Error("Expected Function to do ...")
10     }
11 }
```

To perform the tests the command **go test** is used, that tells if the tests passed or failed. Tables 6.1, 6.2 and 6.3 show some examples of the unit tests results.

Function	PinPath()
Description	Add to the cluster an IPFS path
Result	PASS

Table 6.1: Testing the **PinPath** function

Function	HasFile()
Description	Searches for the file in IPFS given the IPFS path
Result	PASS

Table 6.2: Testing the **HasFile** function

Function	UnpinPath()
Description	Removes the CID of the IPFS path from the cluster peers
Result	PASS

Table 6.3: Testing the **UnpinPath** function

Chapter 7

Conclusions

The main goal for this project was to achieve the integration of distributed network components into the Vixtape media player that, in the beginning, had only some basic features. It was only able to play random content retrieved from a centralized source and now has the capability of playing scheduled ads obtained from a distributed network setup. The implementation of this distributed approach, albeit with some centralized dependencies, into the Android and web players had the clear purpose of improving reliability, reducing the operational costs and improve offline resilience.

From the start, it was expected that towards the end stages of the project it would be possible to test and deploy the developed solution in a real-life like environment. As it became clear during this dissertation, that was not possible due to the lack of hardware availability (i.e., the Android boxes) and also some additional constrains imposed by the pandemic state we experienced. Despite this, based on what was developed and tested it was possible to conclude that the IPFS technology is a great solution for all the problems this project aimed to solve. In more detail, the IPFS Node and Cluster implementations showed that the players were now able to fetch content from other peers without jeopardise to other aspects, such has, video quality and download speed. The possibility of using the IPFS Cluster for data replication allowed the players to have locally stored content without even doing a formal request of it, thus always being ready to keep displaying ads even if the Internet fails.

Given the state of the art in this field, it becomes clear that the IPFS route might be the optimal solution for many projects that aim to achieve distributed data distribution. Also the flexibility and reliability that comes with it exposes some of weaker aspects of today's mainstream Internet. The IPFS may not, or does not intent to, be able to completely replace it, but indeed it can work along

side it in order to provide a more robust and efficient service. A healthy blend of both may change the way people see and interact with the Internet. Maybe the today situations where services are unavailable or just slow will stop being a reality, sooner rather then later. The IPFS is indeed a new technology that came to stay and change the way data is exchanged and stored.

7.1 Future Work

One of the objectives was not only to create the overall content distribution infrastructure, but also to try to make the best possible use of it. Since that in the current state of development the players still depend on the (centralized) Vixtape API to decide which content to display, meaning that if the Vixtape API goes offline, the players even with all the stored content are not capable of fully using it. The ability for the players to generate their own broadcast plans, based on the content they locally have, is an important feature in order to achieve full offline operation. The creation of such scheduling features could also include the creation of an in-memory timeline of broadcast plans. This way the end users would also be able to schedule the ads that are shown, and when this happens. All these features are all planned to be implemented in the near future, now that the entire distributed system is ready to be integrated into the production player's logic.

References

[1] T. A. S. J. Craig Andrews, "Advertising, promotion, and other aspects of integrated marketing communications." https://books.google.pt/books?id=clwvDwAAQBAJ&pg=PA287&dq=ad-blocker+in+marketing&hl=en&sa=X&ved=0ahUKEwj175_Qj9noAhWaD2MBHZFADQoQ6AEIMTAB#v=onepage&q=ad-blocker%20in%20marketing&f=false. (Accessed on 04/13/2020). [Quoted on p. v, 10, 12]

[2] "Content delivery networks - google livros." https://books.google.pt/books?id=p2C2cZkTrmsC&pg=PA140&lpg=PA140&dq=cdn+content+stored+procedure&source=bl&ots=gdCj7vlm-u&sig=ACfU3U3R2oHRHjLCGf73e4qodDFxE9XgaA&hl=pt-PT&sa=X&ved=2ahUKEwiWtZGxs53pAhUT4OAKHRi-DU0Q6AEwAHoECAcQAQ#v=onepage&q=cdn%20content%20stored%20procedure&f=false. (Accessed on 05/05/2020). [Quoted on p. v, 13, 14]

[3] "What's really happening when you add a file to ipfs?." https://medium.com/textileio/whats-really-happening-when-you-add-a-file-to-ipfs-ae3b8b5e4b0f. (Accessed on 05/18/2020). [Quoted on p. v, 23, 26]

[4] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric. in peer-to-peer systems, pages 53–65. springer." https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf. (Accessed on 04/21/2020). [Quoted on p. v, 27, 28, 29]

[5] "Bitswap | ipfs docs." https://docs.ipfs.io/concepts/bitswap/#bitswap. (Accessed on 09/06/2020). [Quoted on p. v, 31]

[6] "Vixtape: Art, culture & commerce." https://www.vixtape.tv/. (Accessed on 04/20/2020). [Quoted on p. 2]

[7] "Realtimeads - a new way of advertising." https://realtimeads.net/. (Accessed on 04/20/2020). [Quoted on p. 2]

[8] "What is ipfs? – ipfs documentation." https://docs.ipfs.io/introduction/overview/#ipfs-is-a-distributed-system-for-storing-and-accessing-files-websites-applications-and-data. (Accessed on 05/11/2020). [Quoted on p. 3, 15]

[9] "Ipfs cluster - pinset orchestration for ipfs." https://cluster.ipfs.io/. (Accessed on 06/02/2020). [Quoted on p. 4, 32]

[10] "The sage handbook of advertising - google livros." https://books.google.pt/books?hl=pt-PT&lr=&id=IcEWBZvpFHoC&oi=fnd&pg=PA17&dq=history+of+advertising&ots=0PAX3apqTd&sig=ctL877EhxEy97LRo0FYtfoK_rkc&redir_esc=y#v=onepage&q&f=false. (Accessed on 11/15/2019). [Quoted on p. 8]

[11] "Understanding consumers attitude toward advertising." https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1517&context=amcis2002. (Accessed on 12/05/2019). [Quoted on p. 8]

[12] "4 things to know as out-of-home goes programmatic | adexchanger." https://www.adexchanger.com/digital-out-of-home/4-things-know-home-goes-programmatic/?fbclid=IwAR0LZmAZesTztvAlJeKzCHCqkYWQPsMal0rIDVANG3QQWxDLlWt4MwhdojI. (Accessed on 03/30/2020). [Quoted on p. 9]

[13] "Programmatic digital ooh does not spell the death of traditional, static outdoor | the drum." https://www.thedrum.com/news/2019/05/14/programmatic-digital-ooh-does-not-spell-the-death-traditional-static-outdoor-0. (Accessed on 03/30/2020). [Quoted on p. 9]

[14] "The power of big data and algorithms for advertising and customer communication - ieee conference publication." https://ieeexplore.ieee.org/document/7872882. (Accessed on 12/17/2019). [Quoted on p. 9]

[15] "Programmatic digital out of home advertising 2019: What's in it?." https://medium.com/@thewritedecision/programmatic-digital-out-of-home-advertising-will-be-big-d6d19ed0c187. (Accessed on 12/16/2019). [Quoted on p. 9]

[16] "What is programmatic digital out-of-home? | broadsign." https://broadsign.com/blog/what-is-programmatic-digital-out-of-home/. (Accessed on 03/30/2020). [Quoted on p. 9]

[17] "The advertising industry has a problem: People hate ads - the new york times." https://www.nytimes.com/2019/10/28/business/media/advertising-industry-research.html. (Accessed on 04/13/2020). [Quoted on p. 10]

[18] "Cisco annual internet report - cisco annual internet report (2018–2023) white paper - cisco." https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. (Accessed on 03/31/2020). [Quoted on p. 13]

[19] "Towards peer-to-peer content retrieval markets enhancing ipfs with icn." (Accessed on 03/31/2020). [Quoted on p. 13]

[20] "Acm: Digital library: Communications of the acm." https://dl.acm.org/doi/fullHtml/10.1145/1107458.1107462#R11. (Accessed on 05/05/2020). [Quoted on p. 13]

[21] "Bittorrent mainline dht measurement." https://www.cl.cam.ac.uk/~lw525/MLDHT/. (Accessed on 06/22/2020). [Quoted on p. 15]

[22] "Bittorrent (btt) white paper." https://www.bittorrent.com/btt/btt-docs/BitTorrent_(BTT)_White_Paper_v0.8.7_Feb_2019.pdf. (Accessed on 08/20/2020). [Quoted on p. 15]

[23] "bittorrentecon.pdf." https://www.cs.swarthmore.edu/~newhall/readings/bittorrentecon.pdf. (Accessed on 06/22/2020). [Quoted on p. 15]

[24] "How does bittorrent work?." https://www.howtogeek.com/141257/htg-explains-how-does-bittorrent-work/. (Accessed on 06/22/2020). [Quoted on p. 15]

[25] "How youtube ad revenue works." https://www.investopedia.com/articles/personal-finance/032615/how-youtube-ad-revenue-works.asp. (Accessed on 08/20/2020). [Quoted on p. 16]

[26] "Ethereum 101 - coindesk." https://www.coindesk.com/learn/ethereum-101/ethereum-smart-contracts-work. (Accessed on 07/08/2020). [Quoted on p. 16]

[27] "The blockchain-based digital content distribution system." https://www.researchgate.net/publication/308861913_The_Blockchain-Based_Digital_Content_Distribution_System. (Accessed on 07/08/2020). [Quoted on p. 16]

[28] "What is hadoop?." https://www.bernardmarr.com/default.asp?contentID=1080. (Accessed on 09/08/2020). [Quoted on p. 17]

[29] "Mapreduce: Simplified data processing on large clusters." https://www.usenix.org/legacy/events/osdi04/tech/full_papers/dean/dean.pdf. (Accessed on 09/08/2020). [Quoted on p. 17]

[30] S. R. R. C. Konstantin Shvachko, Hairong Kuang, "The hadoop distributed file system." (Accessed on 09/08/2020). [Quoted on p. 17]

[31] "A monitorable peer-to-peer file sharing mechanism." (Accessed on 03/31/2020). [Quoted on p. 19]

[32] "Turkey can't block this copy of wikipedia | | observer." https://observer.com/2017/05/turkey-wikipedia-ipfs/. (Accessed on 05/11/2020). [Quoted on p. 19]

[33] "What is ipfs? – ipfs documentation." https://docs.ipfs.io/introduction/overview/#so-why-does-that-matter. (Accessed on 05/11/2020). [Quoted on p. 19]

[34] "Peerpad." https://peerpad.net. (Accessed on 08/31/2020). [Quoted on p. 19]

[35] "Berty · berty technologies." https://berty.tech/. (Accessed on 08/31/2020). [Quoted on p. 19]

[36] "Home - blust - cinema definition 4k movies, 4k tv, and 4k games." http://www.blust.tv/. (Accessed on 08/31/2020). [Quoted on p. 19]

[37] "Usage ideas & examples | ipfs docs." https://docs.ipfs.io/concepts/usage-ideas-examples/. (Accessed on 08/31/2020). [Quoted on p. 19]

[38] "Multihash." https://multiformats.io/multihash/. (Accessed on 05/11/2020). [Quoted on p. 22]

[39] "Sha-2 – wikipédia, a enciclopédia livre." https://pt.wikipedia.org/wiki/SHA-2. (Accessed on 08/22/2020). [Quoted on p. 22]

[40] "multicodec/table.csv at master · multiformats/multicodec." https://github.com/multiformats/multicodec/blob/master/table.csv. (Accessed on 05/11/2020). [Quoted on p. 23]

[41] "multihash/readme.md at master · multiformats/multihash." https://github.com/multiformats/multihash/blob/master/README.md. (Accessed on 08/31/2020). [Quoted on p. 23]

[42] "Ipns – ipfs documentation." https://docs.ipfs.io/guides/concepts/ipns/. (Accessed on 05/12/2020). [Quoted on p. 23]

[43] "merkle-crdts.pdf." https://hector.link/presentations/merkle-crdts/merkle-crdts.pdf. (Accessed on 05/18/2020). [Quoted on p. 23]

[44] "Merkle-dags – ipfs documentation." https://docs.ipfs.io/guides/concepts/merkle-dag/. (Accessed on 05/18/2020). [Quoted on p. 24]

[45] "Git - about version control." https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. (Accessed on 05/26/2020). [Quoted on p. 26]

[46] J. Benet, "Ipfs - content addressed, versioned, p2p file system." https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf. (Accessed on 05/05/2020). [Quoted on p. 26, 27, 29, 32]

[47] "Distributed hash tables (dht) – ipfs documentation." https://docs.ipfs.io/guides/concepts/dht/. (Accessed on 04/07/2020). [Quoted on p. 27]

[48] "coral-nsdi04.dvi." https://pdfs.semanticscholar.org/4143/006482848287f667067654d0a2f4a87a8bfb.pdf. (Accessed on 05/25/2020). [Quoted on p. 29]

[49] "Sybil attack - an overview | sciencedirect topics." https://www.sciencedirect.com/topics/computer-science/sybil-attack. (Accessed on 05/25/2020). [Quoted on p. 30]

[50] "What is pki (public key infrastructure)? - definition from whatis.com." https://searchsecurity.techtarget.com/definition/PKI. (Accessed on 05/25/2020). [Quoted on p. 30]

[51] "Background." http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf. (Accessed on 05/19/2020). [Quoted on p. 32]

[52] "Architecture overview - pinset orchestration for ipfs." https://cluster.ipfs.io/documentation/deployment/architecture/. (Accessed on 06/02/2020). [Quoted on p. 33]

[53] "Conflict-free replicated data types." https://hal.inria.fr/inria-00609399v1/document. (Accessed on 06/10/2020). [Quoted on p. 34, 35]

[54] "acm.dvi." https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf. (Accessed on 06/15/2020). [Quoted on p. 36]

[55] "In search of an understandable consensus algorithm." https://raft.github.io/raft.pdf. (Accessed on 06/16/2020). [Quoted on p. 37]

[56] "Raft consensus algorithm." https://raft.github.io/. (Accessed on 06/16/2020). [Quoted on p. 37]

[57] "Consensus components - pinset orchestration for ipfs." https://cluster.ipfs.io/documentation/guides/consensus/. (Accessed on 06/16/2020). [Quoted on p. 37]

[58] "Rest api - pinset orchestration for ipfs." https://cluster.ipfs.io/documentation/reference/api/. (Accessed on 08/08/2020). [Quoted on p. 39]

[59] "Http api | ipfs docs." https://docs.ipfs.io/reference/http/api/#api-v0-version-deps. (Accessed on 08/07/2020). [Quoted on p. 39]

[60] "Typescript: Typed javascript at any scale.." https://www.typescriptlang.org/. (Accessed on 08/07/2020). [Quoted on p. 49]

[61] "Node.js." https://nodejs.org/en/. (Accessed on 08/07/2020). [Quoted on p. 49]

[62] "Graphql: A data query language - facebook engineering." https://engineering.fb.com/core-data/graphql-a-data-query-language/. (Accessed on 08/07/2020). [Quoted on p. 49]

[63] "The most popular database for modern apps | mongodb." https://www.mongodb.com/. (Accessed on 08/07/2020). [Quoted on p. 49]

[64] "Json and bson | mongodb." https://www.mongodb.com/json-and-bson. (Accessed on 08/07/2020). [Quoted on p. 49]

[65] "What is go language and why use it for your project." https://yalantis.com/blog/why-use-go/. (Accessed on 08/07/2020). [Quoted on p. 50]

[66] "Frequently asked questions (faq) - the go programming language." https://golang.org/doc/faq#What_is_the_purpose_of_the_project. (Accessed on 08/07/2020). [Quoted on p. 50]

[67] "Why developers prefer kotlin. the key kotlin features & updates - dev." https://dev.to/blak_it/why-developers-prefer-kotlin-the-key-kotlin-features--updates-10hp. (Accessed on 08/07/2020). [Quoted on p. 50]

[68] "Kotlin programming language." https://kotlinlang.org/. (Accessed on 08/07/2020). [Quoted on p. 50]

[69] "What is a container? | app containerization | docker." https://www.docker.com/resources/what-container. (Accessed on 08/07/2020). [Quoted on p. 50]

[70] "What is a container? | app containerization | docker." https://www.docker.com/resources/what-container. (Accessed on 08/07/2020). [Quoted on p. 50]

[71] "ipfs/js-ipfs: Ipfs implementation in javascript." https://github.com/ipfs/js-ipfs. (Accessed on 07/20/2020). [Quoted on p. 59]

[72] "ipfs/go-ipfs: Ipfs implementation in go." https://github.com/ipfs/go-ipfs. (Accessed on 07/21/2020). [Quoted on p. 60]

[73] "Golang and node.js comparison: Scalability, performance, and tools." https://yalantis.com/blog/golang-vs-nodejs-comparison/. (Accessed on 08/28/2020). [Quoted on p. 60]

[74] "golang/mobile: [mirror] go on mobile." https://github.com/golang/mobile. (Accessed on 07/21/2020). [Quoted on p. 61]

[75] "Native modules · react native." https://reactnative.dev/docs/native-modules-android.html. (Accessed on 07/21/2020). [Quoted on p. 61]

[76] "Introduction · mobx." https://mobx.js.org/README.html. (Accessed on 07/28/2020). [Quoted on p. 65]

[77] "Change streams — mongodb manual." https://docs.mongodb.com/manual/changeStreams/. (Accessed on 07/23/2020). [Quoted on p. 65]

[78] "ipfs/ipfs-cluster: Pinset orchestration for ipfs." https://github.com/ipfs/ipfs-cluster. (Accessed on 07/24/2020). [Quoted on p. 68]

[79] "libp2p/go-libp2p: libp2p implementation in go." https://github.com/libp2p/go-libp2p. (Accessed on 07/27/2020). [Quoted on p. 69]

[80] "Ipfs powers the distributed web." https://ipfs.io/. (Accessed on 05/12/2020). [Quoted on p. 76]

Appendix A

The following list explains the full IPFS HTTP API available endpoints.

- **/api/v0/add** - Adds a file or directory to ipfs.
- **/api/v0/bitswap...** - All bitswap options available:
  - **/ledger?arg=<peer>** - Shows the current ledger for a peer.
  - **/reprovide** - Trigger reprovider to announde the data it has to the network.
  - **/stat** - Shows diagnostic information on the bitswap agent.
  - **/wantlist** - Shows the wantlist of the peer.
- **/api/v0/block...** - All the options for the blocks option. A block refers to a single unit of data. When adding a file to IPFS it creates a Merkle DAG out of the data of that file. The file is then broken into blocks to arrange it in a tree-like structure and linking them all together.
  - **/get?arg=<key>** - Takes the multihash of a block of data and gets it from the network.
  - **/put** - Stores a block or a set of blocks.
  - **/rm** - Removes a block or a set of blocks.
  - **/stat** - Prints information about a block.
- **/api/v0/bootstrap...** - Prints the list of Peers to bootstrap stored in its config file. This command also offers more options:
  - **/add** - Add peers to the bootstrap peers array in the config.
  - **/add/default** - Add the default peers to the Peers array.
  - **/rm?arg=<peer>** - Removes a peer from the bootstrap peers array.

- **/rm/all** - Removes all peers from the Peers array.

- **/api/v0/cat** - Shows the IPFS object data.
- **/api/v0/cid...** - The cid options are:
  - **/base32arg=<cid>** - Converts a CID into a Base32 CID version 1.
  - **/bases** - List the available multibase encodings.
  - **/codecs** - List available CID codecs.
  - **/format?arg=<cid>&v=<version>&codec=<codec>&b=<multibase>** - Formats and converts a CID to the desired version, codec to convert and the multibase to display the CID.
  - **/hashes** - List available multihashes.
- **/api/v0/commands** - List all available commands.
- **/api/v0/config?arg=<key>&arg=<value>** - Gets and set the peers config values by passing the key-value has an argument.
  - **/edit** - Opens the config file for editing.
  - **/profile/apply?arg=<profile>** - Apllies a profile to the config.
  - **/replace** - Passes a file to replace the current config file.
  - **/show** - Outputs the config file content.
- **/api/v0/dag** - All dag options:
  - **/export?arg=<cid>** - Stream the selected DAG as a .car stream and takes the CID of a DAG root.
  - **/get?arg=<ref>** - Gets a DAG node from the network.
  - **/import** - Import the contents of .car files.
  - **/put** - Addd a DAG node to the network.
  - **/resolve?arg=<ref>** - Resolves a IPLD block taking the path of it as argument.
- **/api/v0/dht** - All options for the dht:
  - **/findpeer?arg=<peerID>** - Finds the multiaddresses associated with a Peer ID.
  - **/findprovs?arg=<key>** - Finds the peers that can provide a certain value given its key.
  - **/get?arg=<key>** - Giving a key as argument, queries the routing system for the best peer to provide it.

  - **/provide?arg=<key>&verbose=<value>** - Announce to the network that it can provide a given key-value pair.
  - **/put?arg=<key>&verbose=<value>** - Write a key-value pair to the routing system.
  - **/query?arg=<peerID>** - Finds the closest Peer IDs to a given peer by querying the DHT, it takes the target Peer ID as argument.

- **/diag...** - All diag options:
  - **/cmds** - List commands run and can take other options to:
  - **/cmds/clear** - Clears inactive requests from the log
  - **/cmds/set-time** - Set how long to keep inactive requests in the log.
  - **/sys** - Prints system diagnostic information.

- **/api/v0/dns?arg=<domain-name>** - Resolves DNS links by taking a domain-name as argument.

- **/api/v0/file/ls** - List directory contents for Unix filesystem objects.

- **/api/v0/files...** - All files options:
  - **/ls?arg=<ipfs-path>** - List directory contents for Unix filesystem objects.
  - **/chcid** - It changes the CID version or hash function of the root nodee of a given path.
  - **/cp?arg=<source>&arg=<dest>** - Copies any IPFS files and directories into the Mutable Filesystem (MFS).
  - **/flush?arg=<path>** - Flush a given path's data to disk. item **/ls?arg=<path>** - List directories in the local mutable namespace.
  - **/mkdir?arg=<path>** - Makes a directory on the given path.
  - **/mv?arg=<source>&arg=<dest>** - Move files.
  - **/read?arg=<path>** - Reads a file in a given MFS.
  - **/rm?arg=<path>** - Removes a file.
  - **/stat?arg=<path>** - Displays file status.
  - **/write?arg=<path>** - Write to a mutable file in a given filesystem.

- **/filestore** - All filestore options:
  - **/dups** - List blocks that are both in the filestore and standard block storage
  - **/ls** - List objects in filestore.

  - **/verify** - Verifies objects in filestore.

- **/api/v0/get?arg=<ipfs-path>** - Downloads IPFS objects given its path as argument.
- **/api/v0/id** - Shows the IPFS node ID info or can take other peerID as argument.
- **/api/v0/key...** - All key options:
  - **/gen?arg=<name>** - Generates 2048 bit Rivest-Shamir-Adleman (RSA) key, takes the name of the key as argument.
  - **/list** - List all local keypairs.
  - **/rename?arg=<name>&arg=<newName>** - Renames a keypair taking the old and new name as argument.
  - **/rm?arg=<name>** - Removes a keypair by the given name.
- **/api/v0/ls?arg=<ipfs-path>** - List directory contents for Unix filesystem objects, takes the path to a IPFS object as argument.
- **/api/v0/monut** - Mounts the IPFS to the filesystem.
- **/api/v0/name...** - This section is related to IPNS:
  - **/publish?arg=<ipfs-path>** - Publish IPNS names and takes the ipfs path of the object to be published as argument.
  - **/pubsub/cancel?arg=<name>** - Cancel a name subscription.
  - **/pubsub/state** - Queries the state of IPNS pubsub.
  - **/pubsub/subs** - Show current name subscriptions. item **/resolve** - Resolve IPNS names.
  - **/api/v0/object...** - All object options:

  - **/data?arg=<cid>** - Outputs the raw bytes of an IPFS object.
  - **/diff?arg=<obj_a>&arg=<obj_b>** - Displays the difference between two IPFS objects.
  - **/get/arg=<cid>** - Gets and serializes the DAG node of the IPFS object.
  - **/links?arg=<cid>** - Outputs the all the linked data in a IPFS object.
  - **/new** - Creates a new object from a IPFS template.
  - **/patch/add-link?arg=<root>&arg=<name>&arg=<ref>** - Adds a link to a given IPFS object.
  - **/patch/append-data?arg=<cid>** - Appends data to a data segment of a DAG node.

- **/patch/rm-link?arg=<root>&arg=<name>** - Removes a link from a given IPFS object.
- **/patch/set-data?arg=<cid>** - Sets the data field of an IPFS object.
- **/put** - Stores the input as a DAG object.
- **/stat?arg=<cid>** - Gets stats for a given DAG Node.

- **/api/v0/p2p...** - All P2P options:
  - **/close** - Stop listening for new connections to forward.
  - **/forward?arg=<protocol>&arg=<listen-address>&arg=<target-address>** - Forward connections to libp2p service, taking the protocol name, listening endpoint and the target endpoint.
  - **/listen?arg=<protocol>&arg=<target-address>** - Creates a libp2p service, takes the protocol name and target endpoint as arguments.
  - **/ls** - List active p2p listeners.
  - **/stream/close** - Close active p2p stream.
  - **/stream/ls** - List active p2p streams.

- **/api/v0/pin...** - All pin options:
  - **/add?arg=<path>** - Pins objects to the local storage.
  - **/ls** - List the objects pinned in the local storage.
  - **/rm?arg=<ipfs-path>** - Removes pinned objects from the local storage.
  - **/update?arg=<from-path>&arg=<to-path>** - Updates a recursive pin.
  - **/verify** - Verify that recursive pins are complete.

- **api/v0/ping** - Pings other IPFS peers to see if they are alive.

- **/api/v0/pubsub...** - All pubsub options:
  - **/ls** - List subscribed topics by name.
  - **/peers** - List peers that are currently pubsubbing with.
  - **/pub?arg=<topic>&arg=<data>** - Publish a message to a given pubsub topic, takes the topic and the data to send as arguments.
  - **/sub?arg=<topic>** - Subscribes to a given topic.

- **/api/v0/refs?arg=<ipfs-path>** - Lists links from an object.

- **/api/v0/refs/local** - Lists all local references.

- **/api/v0/repo...** - All repo options:
  - **/fsck** - Removes repo lockfiles.
  - **/gc** - Performs a garbage collection sweep on the repo.
  - **/stat** - Gets stats for the currently used repo.
  - **/verify** - Verifies that all blocks in the repo are not corrupted.
  - **/version** - Shows the repo version.
- **/api/v0/resolve?arg=<name>** - Resolves the value of names to IPFS. Takes the name to be resolves as argument.
- **/api/v0/shutdown** - Shuts down the running IPFS daemon.
- **/api/v0/stats...** - All stats options:
  - **/bitswap** - Shows diagnostic information on the bitswap agent.
  - **/bw** - Prints the IPFS bandwidth information.
  - **/repo** - shows stats for the currently used repo.
- **/api/v0/swarm...** - All swarm options:
  - **/addrs** - Lists known addresses.
  - **/addrs/listen** - Lists interface listening addresses.
  - **/addrs/local** - Lists local addresses.
  - **/connect?arg=<address>** - Opens a connection to a given address.
  - **/disconnect?arg=<address>** - Closes the connection to a given address.
  - **/filters** - Manipulates address filters.
  - **/filters/add?arg=<address>** - Adds an address filter.
  - **/filters/rm?arg=<address>** - Removes an address filter.
  - **/peers** - Lists all peers with open connections.
- **/api/v0/tar/add** - Imports a tar file into IPFS.
- **/api/v0/tar/cat?arg=<ipfs-path>** - Exports a tar file from IPFS.
- **/api/v0/version** - Shows the IPFS version currently being used.
- **/api/v0/version/deps** - Shows information about dependencies used for build

Appendix B

This section presents a set of images of the ads players and the Vixtape dashboard user interface. They aim to clarify the workflow between the players and the end user inputs.

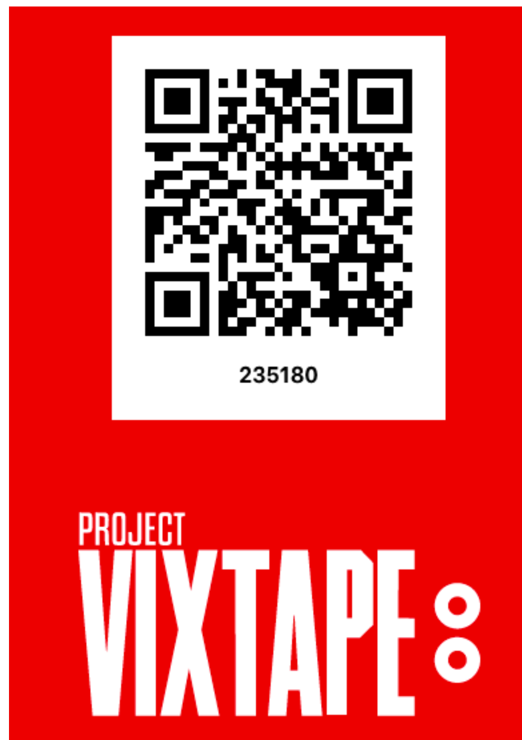


Figure B.1: The player displaying a QR code for the user to bind it on his dashboard

Players 1 TOTAL

List View Map View + Add Player

NAME	STATUS	ORIENTATION	AREA TYPE	VENUE TYPE	CREATED AT
5F33E3	ACTIVATED	LANDSCAPE			14:09, 08 SEP 2020

1-1 of 1

Figure B.2: The list of players owned by the end user

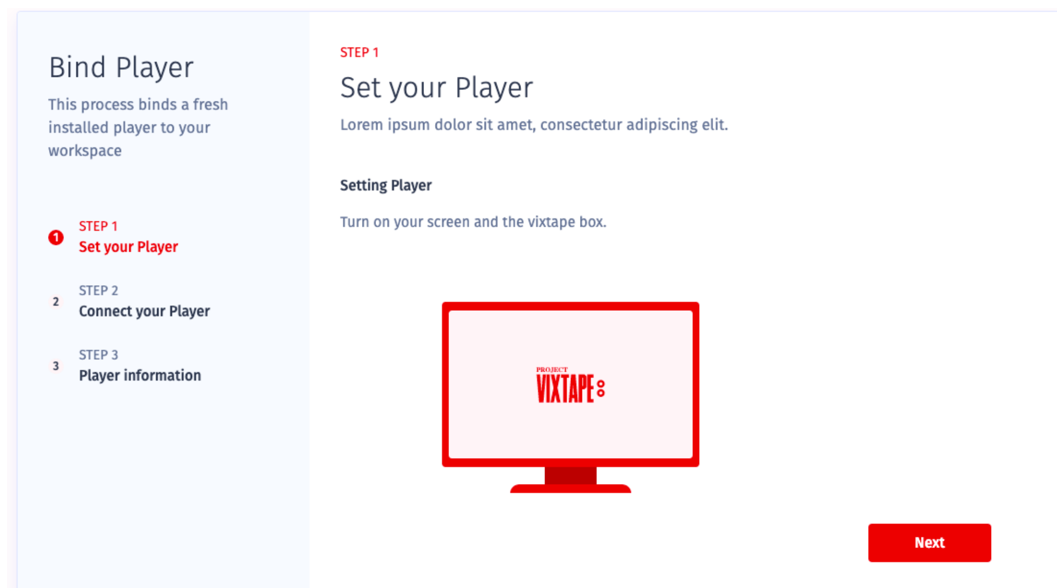


Figure B.3: The first step of the binding process

Bind Player
This process binds a fresh installed player to your workspace

~~STEP 1~~
Set your Player

STEP 2
Connect your Player

STEP 3
Player information

STEP 2
Connect your Player

Connection screen

In this process we will identify which Player belongs to your workspace. Your Player should now be showing a set of 6 characters and changing the set from time to time. Please insert those characters on the following boxes and press next.

BINDING CODE

2 3 5 1 8 0

Back Next

Figure B.4: The input token screen. The user should type in the generated token that can also be seen in Figure B.1

Bind Player
This process binds a fresh installed player to your workspace

~~STEP 1~~
Set your Player

~~STEP 2~~
Connect your Player

STEP 3
Player information

Name
Address
Geolocation
Settings

STEP 3
Player information

Name

5f5767

Address

STREET NAME
SUITE, APT.
CITY
STATE
ZIP CODE
COUNTRY

Geolocation

Figure B.5: Setting up the player information, part I

Figure B.6: Setting up the player information, part II

Figure B.7: The user's campaign management page

Figure B.8: The web browser player, playing a sample video

Figure B.9: A Android box sample, connected to a display device