The Design and Implementation of a Security and Containment Platform for Peer-to-Peer Media Distribution

Quiran Storey

Thesis presented in partial fulfilment of the requirements for the degree
Master of Science in Engineering
in the Faculty of Engineering at Stellenbosch University

Supervisor: Prof. G-J. van Rooyen
Department of Electrical and Electronic Engineering

December 2013

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2013

Abstract

The Design and Implementation of a Security and Containment Platform for Peer-to-Peer Media Distribution

Q. Storey

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng

December 2013

The way in which people consume video is changing with the adoption of new technologies such as tablet computers and smart televisions. These new technologies, along with the Internet, are moving video distribution away from satellite and terrestrial broadcast to distribution over the Internet. Services online now offer the same content that originally was only available on satellite broadcast television. However, these services are only viable in countries with high speed, inexpensive Internet bandwidth. The need therefore exists for alternative services to deliver content in countries where bandwidth is still expensive and slow. These include many of the developing nations of Africa.

In this thesis we design and develop a video distribution platform that relies on peer-to-peer networking to deliver high quality video content. We use an existing video streaming peer-to-peer protocol as the primary distribution mechanism, but allow users to share video over other protocols and services. These can include BitTorrent, DC++ and users sharing hard drives with one another. In order to protect the video content, we design and implement a security scheme that prevents users from pirating video content, while allowing easy distribution of video data. The core of the security scheme requires a low bandwidth Internet connection to a server that streams keys to unlock the video content. The project also includes the development of a custom video player application to integrate with the security scheme.

The platform is not limited to, but is aimed at high speed local area networks where bandwidth is free. In order for the platform to support feasible business models, we provision additional services, such as video cataloging and search, video usage monitoring and platform administration. The thesis includes a literature study on techniques and solutions to secure video entertainment, specifically in a peer-to-peer environment.

Uittreksel

Die ontwerp en implimentasie van 'n sekure en begeslote platvorm vir portuurnetwerk mediaverspreiding

Q. Storey

Departement Elektriese en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MScIng

Desember 2013

Die wyse waarvolgens mense video verbruik is aan die verander met die ingebruikneming van nuwe tegnologie soos tabletrekenaars en slim televisiestelle. Hierdie nuwe tegnologie tesame met die Internet maak dat die verspreiding van video al hoe minder plaasvind deur middel van satellietuitsendings en al hoe meer versprei word deur die Internet. Aanlyn-Internetdienste bied deesdae dieselfde inhoud aan as wat voorheen slegs deur beeldsending versprei is. Hierdie dienste is egter slegs lewensvatbaar in lande met hoëspoed- en goedkoop Internetbandwydte. Daar is dus 'n behoefte aan alternatiewe tot hierdie dienste in lande waar bandwydte steeds duur en stadig is. Baie lande in Afrika kan in hierdie kategorie ingesluit word.

In hierdie tesis word 'n videoverspreidingsplatform ontwerp en ontwikkel, wat van portuurnetwerke gebruik maak om hoëkwaliteit-beeldmateriaal te versprei. Die stelsel gebruik 'n bestaande portuurnetwerk-datavloeiprotokol as die premêre verspreidingsmeganisme, maar laat gebruikers ook toe om videoinhoud direk met ander gebruikers en dienste te deel. BitTorrent, DC++ en gebruikers wat hardeskywe met mekaar deel word hierby ingesluit. Ten einde die videoinhoud te beskerm ontwerp en implimenteer ons ’n sekuriteitstelsel wat verhoed dat gebruikers die videoinhoud onregmatig kan toe-eien, maar wat terselfdertyd die verspreiding van die data vergemaklik. Hierdie sluit die ontwikkeling van ’n pasgemaakte videospeler in. Die kern van die sekuriteit-stelsel benodig ’n lae-bandwydte-Internetverbinding na ’n bediener wat sleutels uitsaai om die videoinhoud te ontsluit.

Alhoewel nie daartoe beperk nie, is die platform gemik op hoëspoed-plaas-likegebiedsnetwerke met gratis bandwydte. Om die platvorm aan ’n haalbare sakemodel te laat voldoen het ons vir addisionele dienste soos videokatalogi-sering met soekfunksies, videoverbruikersmonitering en platvormadministrasie voorsiening gemaak. Die tesis sluit ’n literatuurstudie oor tegnieke en op-lossings vir die beskerming van video data, spesifiek in die portuurnetwerke omgeving, in.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

1. My family and friends for their support throughout this project;
2. Everyone at the MIH Media Lab who created a creative space to try new ideas;
3. MIH for the funding of my research;
4. Jacques Bruwer for his work on the underlying Jax.TV protocol; and
5. My supervisor Professor Gert-Jan van Rooyen.

Terms of Reference

This project was commissioned by the MIH Media Lab and MIH, and the following specific objectives were placed on the project:

- Evaluate current video encryption techniques and existing DRM solutions.
- Design and implement a security mechanism for peer-to-peer media distribution.
- Create a working test environment which includes an operating version of Jax.TV (a protocol technology demonstrator).
- Design and implement a complete peer-to-peer video-on-demand platform by extending Jax.TV, as the primary distribution mechanism.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

AAC	Advanced Audio Codec
AMQP	Advanced Message Queueing Protocol
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
AVC	Advanced Video Codec
CLI	Command-Line Interface
DECE	Digital Entertainment Content Ecosystem
DRM	Digital Rights Management
DSM	Digital Storage Media
DVB	Digital Video Broadcasting
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HTML	Hypertext Markup Language
HD	High Definition
IP	Intellectual Property
IPTV	Internet Protocol Television
JSON	JavaScript Object Notation
LAN	Local Area Network
MAN	Metropolitan Area Network
MPEG	Moving Pictures Expert Group
MVC	Model-View-Controller

PRNG	Pseudorandom Number Generator
PVR	Personal Video Recorder
RAM	Random Access Memory
REST	Representational State Transfer
RDBMS	Relational Database Management System
TCP	Transmission Control Protocol
SD	Standard Definition
SDK	Software Development Kit
SSL	Secure Sockets Layer
URL	Uniform Resource Locator

Abbreviations

Kbps	kilobits per second
MB	megabyte
Mbps	megabits per second
MBps	megabytes per second
TV	television
XOR	exclusive OR

Chapter 1

Introduction

Content providers are adopting new technologies for the distribution of their content. Recently, Netflix [1], an online streaming video service, acquired exclusive American television (TV) subscription rights to feature films released by Walt Disney Studios [2]. This is the first time a major Hollywood studio is opting for digital distribution over traditional TV [3]. It is clear that there is a shift to new digital distribution technologies, but most of these require high capacity Internet bandwidth to operate. A solution is required for regions where Internet bandwidth is either expensive or low in capacity. A peer-to-peer media distribution system was developed at Stellenbosch University as prior work to address this requirement, but needs to be extended to a functional platform with video security and the necessary platform services to support feasible business models.

In this thesis we develop a video platform with a security scheme to prevent users from pirating media from the platform. This security scheme is designed specifically for peer-to-peer media distribution, to allow users to freely distribute media through a variety of ways. The video platform includes the following services: video search, download, playback, catalogue services, usage tracking and user-based authentication. These services are developed along with a client application and servers to support the different services offered by the platform. The client application allows users to watch video content from the platform and requires a low bandwidth real-time connection with servers on the platform.

The remainder of this chapter provides the background and objectives of this thesis.

1.1 Background

During the 20th century television as a content delivery platform, was the only source of video entertainment available to consumers in their homes. TV did this by making programming available through technologies such as TV receivers, decoders (set-top boxes), video cassette and DVD players. While the TV receiver, decoder and DVD are still widely used, desktops, laptops, smartphones and tablets are now able to deliver the same video content to consumers that was previously only available via TV receivers, decoders and DVDs.

Along with these devices and terrestrial and satellite broadcast TV, the Internet now provides another means of accessing video entertainment. This happens both legally and illegally. Some of these legal services include Netflix [1], YouTube [4], Apple's iTunes Store [5] and DSTV on Demand [6].

Netflix is a subscription service that allows users to watch an unlimited number of movies and TV shows each month and is charged for at a flat rate of $7.99 per month. Netflix streams the video data directly to the device used to play the video. These include the computer, Nintendo Wii, Sony Playstation 3 and Microsoft XBOX 360. Netflix is currently available in 40 countries around the world [1].

YouTube is a platform that was launched in 2005 that allows users to discover, watch and share videos online [7]. YouTube is a free service to users watching and sharing videos, while allowing businesses to advertise on the platform. Advertisements can be text that is overlaid on the video or in-stream advertisements similar to traditional TV advertisements [8]. As of 2010, YouTube includes a rental service (although it is still in beta) and has started video streaming live events [9]. In 2012 the London Olympic games were broadcast live, for free, on YouTube using the International Olympic Committee's YouTube channel [10]. YouTube, like Netflix, streams all the video content to the user and does not offer video downloads.

Apple has the iTunes Store where users can purchase TV shows (that are free of advertisements) and purchase or rent movies [5]. Apple currently supports playback on computers and their devices. Apple provides the user with the means by which to discover new content through browsing, search and recommendations. Apple only streams video data to the Apple TV, while on the other devices the service requires that the user download the file before viewing it (unlike YouTube and Netflix).

DStv, a South African satellite pay TV operator, launched DStv on Demand in 2010 [6]. This service provides existing DStv customers with the ability to watch series, movies or sport highlights that have already been broadcast on DStv [11]. DStv on Demand allows the user to either stream the video data or download the video before watching it. DStv BoxOffice is another service from DStv that allows users to rent newly released and classic movies online [12].

1.2 Previous Work

The live streaming and on-demand video services listed above are ideal in countries served by cheap, high speed Internet, but for example, are not feasible in the developing nations of Africa. Here Internet bandwidth is scarce and costs are usually thousands of times higher than that of the developed world [13]. According to statistics from the International Telecommunications Union (ITU), users in Europe have 25 times more international Internet capacity than users in Africa [14]. We can speculate that the poor fixed broadband penetration of 0.2% [14] attributes to this lack of Internet capacity. A solution is required that is specifically suited for these regions.

In 2009, the MIH Media Lab commissioned a project to design and implement a peer-to-peer video-on-demand system that would allow users to view real-time and delay-tolerant media, as well as provide a system for a third party to inject content into the system and monitor viewership. The project chose to implement the GoalBit platform [15] for the under lying transportation mechanism and the project was named Jax.TV [16].

Jax.TV uses a peer-to-peer architecture but focuses on high speed local area networks (LANs), instead of the Internet. The details of Jax.TV are covered in Chapter 2 as part of the literature study. Jax.TV is primarily targeted at campus-wide networks, gated communities and wireless user groups (WUGs).

Campus-wide networks are typically high speed LANs in buildings or residences that are interconnected using a combination of ethernet and fiber optic networks. Some campuses have buildings that span the greater part of a city and this is an example of a metropolitan area network (MAN).

Gated communities (also known as security villages) are residential areas that are walled or fenced off with access control. Gated communities are becoming more and more popular [17] in South Africa and these communities allow for companies to offer multi-play services, such as TV, Internet, telephony and security. Although configuration is unique for each community, typical configuration comprises of a fiber optic network installed within the gated community with a terminal device installed in each household. A data center is installed on the network and acts as the gateway to the community. The data center receives Internet, telephony and TV and distributes these services to the homes within the community.

Wireless user groups (WUGs) use standard WiFi equipment to create infrastructure to exchange information and data [18], but do not necessarily have to offer Internet access. WUGs typically span entire cities, such as Cape Town [19] and Pretoria [20]. WUGs can be operated as darknets, whereby the users are connected by high-bandwidth connections and are able to distribute content over this network [21].

1.3 Problem Statement

Jax.TV is ideal for the deployment on campus-wide networks, gated communities networks and darknets, and provides bandwidth intensive video streaming and downloads to operate in regions where Internet bandwidth is either not cheap or not available in a high capacity.

Jax.TV in its current implementation has no content security built into the platform and is only a distribution platform. The very feature that makes digital content ideal for distribution, also makes it really easy to pirate. Unlike analogue content, digital content can be duplicated perfectly with almost no cost or effort to the consumer [22]. This is a major concern for content providers as they require systems that are secure against copyright violation [22].

Digital rights management (DRM) provides a method for securing against copyright violation, but as Jamkhedkar and Heileman [22] explain, the field of DRM is still in the early stages of development. The Jax.TV platform requires a security system specifically designed for the peer-to-peer distribution nature of the platform.

1.4 Objectives

The goal of this project is to create a platform that allows users to watch high quality video content across multiple devices. The platform will use Jax.TV as the primary video distribution mechanism on high speed networks, but the platform will not be limited to high speed networks and will allow users to transfer the video data through other mechanisms. These can include existing file sharing networks such as BitTorrent [23] and DC++ [24] or even portable hard disk drives.

A security system is to be designed to allow this freedom of media distribution while preventing piracy and allowing the content providers to monitor usage and statistics of video usage on the platform. In addition to the security system, a prototype video player must be implemented along with the services necessary to provide the user with a feature rich experience.

In order to achieve this goal, the following objectives were set for the project:

- Perform a literature study of video encryption techniques and existing Digital Rights Management (DRM) solutions;
- Design a security system to prevent media being played outside of the platform;
- Design and implement additional services that will assist the platform in supporting a feasible business model;
- Prototype a client application that is able to play video content and that ties into the additional services mentioned above;
- Implement the platform as a modular extension of Jax.TV.

1.5 Overview

Chapter 2 is the literature study that was undertaken for the project. The chapter begins by introducing a background to networking and specifically peer-to-peer technologies, before looking into video encryption techniques and existing DRM solutions.

Chapter 3 details the design of the project by dividing the platform into essentially two components, namely the clients and the server. The design also includes the specifications and use cases.

Chapter 4 details the implementation of the individual components that make up the server, as well as the implementation of the server as single entity. This chapter also details the encryption procedure that is used to secure the video content on the platform.

Chapter 5 covers the implementation of the client application, specifically the user interaction and the security of decrypted video data. This chapter also addresses the issues encountered in implementing a secure video player.

Chapter 6 puts the platform through its paces and describes the evaluation procedure and the results that were obtained.

Chapter 7 concludes by reviewing the objectives set out by the thesis and makes recommendations for future work.

Chapter 2

Literature Study

This chapter introduces the necessary information to provide a background on peer-to-peer networking and the Jax.TV platform. This is followed by the research on securing digital video information through encryption and existing digital rights management solutions that are currently available. In the process video coding standards are also introduced.

2.1 Networking

In order to discuss Jax.TV, a strong understanding of networking and peer-to-peer technologies is needed. Tanenbaum [25] defines the following terms in networking:

layers: A network is organised by layers that create a hierarchy.

service: A service provides an interface to the layer above, providing the layer with a set of operations.

protocol: A set of rules that define the format of messages exchanged on a layer (implementation of the service).

protocol stack: The system that is made up of one protocol per layer.

architecture: The set of layers and protocols.

2.1.1 BitTorrent

Networking relies on various architectures but we focus primarily on two architectures, namely client-server and peer-to-peer [26]. In a client-server architecture, a central server is responsible for uploading the data and has to carry the combined capacity of each client's download. Examples of these include Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP). A peer-to-peer architecture is a distributed architecture whereby each peer distributes data among the other peers and these peers are considered equal in bandwidth capacity [26]. The peer-to-peer architecture offers many advantages over the client-server architecture, such as load balancing, distribution, dynamic scaling, redundant data and fault tolerance. But perhaps the most important advantage that peer-to-peer offers over client-server, particularly for video distribution, is the fact there isn't a server that has to carry the combined capacity of the downloading nodes. An example that uses the peer-to-peer architecture is the BitTorrent protocol.

BitTorrent is a protocol for distributing files, that is designed for seamless integration with Hypertext Transfer Protocol (HTTP) and the Internet [27]. Instead of all the data being transferred via HTTP, BitTorrent only uses HTTP to transfer small files (a few kilobytes) that contain metadata about the file that is to be downloaded. Once the peer has this metadata it can perform the transfer of data using BitTorrent's peer protocol that operates above the Transmission Control Protocol (TCP). The following entities are used in the BitTorrent protocol: a web server, a metadata file (.torrent), a BitTorrent tracker, a web browser and a BitTorrent download application.

A web server is used to host metadata files for users to access and download using a web browser. It is also possible to include a search engine on the web server to help users more easily discover metadata files. Once the user has downloaded the metadata file, it can be used along with a BitTorrent download application to download the data file and act as a peer in the network. Peers are the nodes in the network that are uploading and downloading data from one another.

Trackers provide a means for peers to locate each other. This communication between the tracker and peers is done on the application layer that is layered above the HTTP protocol [28]. The peers use this communication channel to announce themselves and information about the files they have available to upload (seed) and files they wish to download. Each file is divided into pieces of equal size [15]. All pieces of a file have a SHA1 hash and are included in the .torrent metadata file. These pieces are downloaded by the peers and then seeded back into the network once they are completely downloaded and their SHA1 hash has been confirmed. Each peer, through use of the tracker, knows at any time which peers are downloading the same file. This collection of peers is called a swarm.

In order to achieve good download performance, the following piece selection modes are available [28]:

- strict priority,
- rarest first,
- random first piece; and
- endgame mode.

Each piece selection mode has its own piece selection algorithm and is best suited for the different stages of the download process.

Although peer-to-peer networks like that of BitTorrent have revolutionised the way data can be distributed [29], BitTorrent is of no use to live video streaming.

2.1.2 GoalBit

GoalBit is the first free and open source peer-to-peer system for distributing real-time video streams [15]. The GoalBit platform uses a peer-to-peer protocol like that of BitTorrent but is optimised for live video streaming. The difficulty with a peer-to-peer architecture is that the nodes are highly dynamic, allowing them to join and leave the network at any point. Live video streaming using a peer-to-peer architecture therefore has to meet far stricter requirements than regular peer-to-peer file sharing, as the selection modes have to satisfy real-time streaming. However, the advantage of a peer-to-peer architecture is the abundance of unused resources that the peers have to offer. The GoalBit platform consists of the following network components:

Broadcaster: Introduces content onto the platform from the source . This can be a capture card, webcam or another HTTP, Microsoft Media Server (MMS) or Real-time Transport Protocol (RTP) stream.

Peers: Clients that connect to a stream in order to view it and upload video data.

Super-Peers: Regular peers with the exception of having a large amount of uploading capacity.

Tracker: Responsible for the management of the peers (same role as a tracker in a BitTorrent network)

GoalBit uses a .goalbit metadata file which contains the necessary configuration data for the channel to which users wish to connect. All communication between the peers and the tracker occur over HTTP or HTTPS, while the communication among peers happens over the transmission control protocol (TCP). GoalBit takes various video compression algorithms (codecs) as input and multiplexes (muxes) these codecs into different video containers. The muxed stream is used to generate fixed sized pieces called chunks. The encapsulation of video data into pieces is defined as the GoalBit Packetized Stream [15].

2.1.3 HTTP Secure

Often it is necessary to encrypt information transferred over a network connection to prevent people other than the sender and receiver from accessing the information. Information that is exchanged over an HTTP connection is sent in plaintext and is susceptible to two types of intrusion by third parties to the conversation: passive and active intrusion. Passive intrusion occurs when the intruder listens in on network traffic (packet sniffing), whereas active intrusion means the intruder is able to modify network traffic (man-in-the-middle attacks among otheres) or perform replay attacks [25]. To prevent packet sniffing and man-in-the-middle attacks, HTTP traffic can be used over a Secure Sockets Layer (SSL) connection using the standard port 443. This is known as HTTP secure or HTTPS.

SSL occurs between the transport layer and the application layer and adds overhead due to the handshaking procedure and encryption process, but prevents other parties from accessing the sensitive data. It is important to note that a connection is secure only as long as the private keys are kept secret.

2.1.4 Jax.TV

Currently the Jax.TV [16] implementation of GoalBit is written in the Python programming language [30] and consists of a tracker and peers. The tracker operates in the same way as the tracker of GoalBit does and is implemented using Python's built-in HTTP server. The peers are invoked using the command-line interface (CLI) and are able to upload and download video data. There is at present no user interface or any video playback functionality.

Jax.TV makes use of a .jaxtv metadata file for the peer to obtain information about the video. The metadata file is JSON [31] encoded and contains the following information:

File Name: The name of the video file.

Sliding Window Length: The sliding window represents the pieces that the video player must consume in a sequential manner to reproduce the video.

Name: The title of the video.

Info Hash: An MD5 hash sum of the video data file that is used as the identifier for the video and hereon referred to as the video ID.

Chunk Size: File pieces are referred to as chunks and this parameter contains the piece size for the video.

2.2 Content Security

Digital content protection plays an important role for content providers. So much so, that a digital rights system called UltraViolet was unveiled by the Digital Entertainment Content Ecosystem (DECE) on July 20th 2010 [32]. The DECE consortium consists of major Hollywood studios, DRM vendors and other partners. Disney also has a similar approach to digital distribution called Keychest [33]. It is evident that content security is important to the content providers and in order to secure the Jax.TV platform, an assessment on video encryption techniques and the current solutions available is required to secure content for the platform.

Liu and Koenig [34] divide video applications into two categories: sensitive video applications and entertainment applications.

Sensitive video applications usually require strict security requirements that are very similar to that of text encryption. A typical example of this is video conferencing in a corporate or government environment. Sensitive video applications must be able to withstand traditional cryptographic attacks, as well as perceptual attacks. Perceptual attacks are aimed at the visible information in the video data and attempt to reconstruct the video from the encrypted data. Two ways exist in which these perceptual attacks can be executed [34]. The first is to treat encrypted parts of the video stream as damaged sequences that are caused by packet loss or bit errors. The attacker attempts to then reconstruct the original video by using error concealment techniques that rely on statistical information about the video [35]. In the second technique the attacker attempts to replace encrypted parts with random data and test whether the video data becomes visible. Both these attacks are only applicable on selective encryption algorithms where the only a select amount of video data is encrypted.

In comparison with sensitive video applications, entertainment has a lower level of security. One could speculate that content providers may be skeptical to try alternative forms of broadcasting when entertainment video applications are approached with a lower level of security.

Liu and Koenig [34] also believe that encryption for entertainment applications can be considered secure if the following is true:

1. the cost involved in breaking the algorithm is higher than the license fee for the content; or

2. the time required to break the encryption is longer than the time that the content is considered valuable.

The value of content in an entertainment application is directly related to the quality of the content and the time at which it is made available. The value of a movie, for example, drops exponentially as time passes [34].

Iwata *et al.* [36] list the features of content sharing in a peer-to-peer environment:

- The content itself is transported directly between users, so there is no intermediary.
- Content can be registered by any user.
- It is difficult to gauge how many users will have access to the content.

These points are important when implementing a security platform for video content.

2.2.1 Video Standards

Before discussing encryption, an understanding of the video standards used for video coding and the various iterations of the standards is required.

The MPEG-1 video coding standard was designed for digital storage media (DSM) such as compact disks, that did not suffer from high bit errors. Because the standard was not designed to be robust against bit errors, it was not suitable for broadcast transmission [37].

The MPEG-2 standard was designed as a generic standard, intended for a variety of audio visual coding applications [37] and adopted for digital TV and HDTV. MPEG-2 is similar to MPEG-1 in terms of compression, while providing support for higher resolutions, frame rates, bit rates and support and compression of interlaced video [38]. However, MPEG-2 was designed with transmission in mind, having the requirement to support asynchronous transfer mode (ATM) networks. MPEG-2 defined two types of streams; the Program Stream, similar to the MPEG-1 for compatibility (another requirement) and the Transport Stream, designed for noisy channels [37]. MPEG-2 compression of progressive video is achieved through the encoding of three different types of pictures within the media stream:

- Intra-pictures (I-Pictures) are coded without reference to preceding and succeeding pictures (intra-coded).
- Predicted pictures (P-Pictures) are encoded with reference to other pictures (inter-coded)
- Bidirectionally predicted pictures (B-Pictures)

One video coding standard that is ready for wide adoption is the H.264/MPEG-4 part 10[1] standard, also known as H.264/Advanced Video Coding (AVC) [39]. H.264/AVC was designed to improve coding efficiency and error robustness over the previous stands like MPEG-2, H.263 and MPEG-4 part 2 [40]. H.264 is used in high definition (HD) and standard definition (SD) digital television, HD DVD formats and Digital Video Broadcasting (DVB).

2.3 Encryption

Encryption is the process of transforming a message into cipher code using an encryption algorithm, in order to prevent unauthorised access to the message [41]. Encryption can be used to provide end-to-end security and with the help of watermarking, copyright protection can be enforced [38].

The problem encountered when encrypting video data with traditional text-based (naive) encryption techniques, is the massive volume of data contained in the video content. It is primarily for this reason that other techniques are required to meet the demands of video consumption in a real-time environment. The following considerations need to be made when encrypting video data [38]:

- the compromise between large amounts of data and encryption speed;
- the compromise between compression efficiency and encryption performance;
- the dependence of the encryption technique on the compression algorithm of the video;
- the conditions that do not allow lossy compression; and
- the requirement for special features such as format compliance, scalability and fault tolerance.

H.264 video has various encryption techniques such as pre-compression encryption, post-compression encryption and joint compression and encryption [38].

For MPEG video data, Angelides and Agius [38] propose four levels of encryption:

[1]ITU-T H.264 and ISO/IEC MPEG-4 AVC.

1. encrypting all headers;
2. encrypting all headers and I-frames;
3. encrypting all I-frames and all blocks in P and B frames; or
4. encrypting all frames.

When encrypting video there are two levels of video security available [38]:

1. The encrypted images have a low image quality, but the user can still see the original video.
2. The image is fully encrypted and the picture is not comprehensible to the user.

An encryption scheme that is secure against an adversary with an infinite amount of computational power is considered a perfectly secret encryption scheme. Unlike perfect security, computational security schemes can be broken with enough time and computing power. However, under certain assumptions, it would take many lifetimes to break these schemes and this level of security is adequate for most applications.

Kerckhoff's principle states that the cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience [42]. Replacing a key is far easier than replacing the encryption algorithm. Katz and Lindell [42] clearly state that security through obscurity is a dangerous practice.

According to the sufficient key space principle, a secure encryption scheme must have a key space that is not vulnerable to exhaustive search [42]. This principle is however, only true when the number of values the key can attain is smaller than the number of values the message can attain.

2.3.1 MPEG Encryption Techniques

Qiao and Nahrstedt [43] performed a comparison of MPEG encryption algorithms using the encryption speed, security level and stream size as metrics for performance. The following algorithms were evaluated in their article:

Naive: Encryption of the entire MPEG stream using traditional text-based encryption schemes like DES.

Selective: Encryption of the only I Frames in the MPEG stream. P and B frames are meaningless without the corresponding I frame, therefore encrypting the P and B frames would be irrelevant.

ZigZag-Permutation: Encryption becomes part of the the compression process of MPEG video.

Video Encryption: The Video Encryption Algorithm (VEA) relies on the different statistical behaviour of compressed video for encryption

Pure Permutation The Permutation Algorithm encrypts the video byte stream by permutation.

Table 2.1: Comparison of MPEG encryption techniques (reproduced from [43]).

Algorithm	Security	Speed	Size
Naive	High	Slow	No change
Selective	Moderate	Fast	Increase
ZigZag Permutation	Very Low	Very Fast	Big Increase
Video Encryption	High	Fast	No Change
Pure Permutation	Low	Super Fast	No Change

From Table 2.1 it would appear that naive encryption is not the best choice because of the slow speed. However, naive encryption has the benefit that it is not codec dependent when encrypting on the byte level of the video file. What is lost in speed, is gained in compatibility resulting in an encryption algorithm that has excellent security and no change in file size.

2.3.2 Pseudorandomness

The one-time pad, discussed in Section 2.3.3, requires pseudorandom number generation and therefore a basic background to pseudorandomness is explained. A distribution $\mathcal{D}$ is pseudorandomly distributed over strings of length ℓ if $\mathcal{D}$ is indistinguishable from the uniform distribution of length ℓ [42]. Stated more loosely, a distribution is said to be pseudorandom if it appears to have a uniform distribution.

Although pseudorandomness is not true randomness, it does have a few advantages. Firstly, if ciphertext appears random to the adversary, then the

the adversary has no information about the plaintext. Secondly, a long pseudorandom string can be recreated using a short seed. However, this seed must be chosen at random.

2.3.3 One-Time Pad

Originally patented by Vernam in 1917, Vernam's Cipher (better known today as the one-time pad) is a cipher that is able to obtain perfect secrecy [42]. Being a symmetric-key encryption scheme, the same key used to encrypt is also used to decrypt. The one-time pad performs a bitwise exclusive OR (XOR) of each bit from the message with each bit of the key (also known as the pad). The result from the XOR operation is the cipher text. An example of a bitwise XOR is given in Figure 2.1. The XOR operation can be replaced with other operations like modular addition or multiplication, but the XOR operation has become associated with the one-time pad [42]. The benefit that the XOR operation has over other operations is the speed with which it encrypts and decrypts data.

message:	1 0 1 1
key:	0 1 0 1
cipher:	1 1 1 0
key:	0 1 0 1
message:	1 0 1 1

Figure 2.1: One-time pad example.

The one-time pad is not without disadvantages. Firstly, the key has to have a minimum length of that of the message. Having long keys makes it difficult to the store the keys. Often, the length of the message is not known in advance and there is no upper bound. Lastly, the one-time pad is only secure if the key is used only once.

2.3.4 A Streaming Encryption Example

Text based encryption techniques are not suitable for the high volume of data associated with video [44]. Lan *et al.* [44] proposed a peer-to-peer architecture for live streaming with DRM. There they chose to use a selective chaotic encryption algorithm [45].

A portion of important data is selected from the video, such as the quantisation step, then a sequence is generated through the logistic chaotic mapping sequence:

$$x_{n+1} = \mu \cdot x_n(1 - x_n) \tag{2.3.1}$$

where $x_n \in (0, 1)$ and $\mu \in (0, 4)$. The initial values x_0 and μ are the secret keys. The sequence that is generated in (2.3.1) is then similar to that of a stream cipher and is XORed with the data selected in the beginning. The XOR is a native machine instruction and therefor performed extremely fast. This encrypted data is then fed into the streaming feed.

Chaotic systems are extremely sensitive to disruption and slight variances in initial values cause the values to differ greatly [46].

2.4 Digital Rights Management

Digital Rights Management (DRM) refers to a technique which enforces policies to protect content throughout the distribution and consumption process. DRM imposes restrictions based on the usage rights assigned to the content. Cohen [47] states that “At the simplest form, DRM systems impose direct restrictions on what individuals can do in the privacy of their own homes with the copies of works they’ve paid for”. Examples of these restrictions include an expiration date, a date before which the content cannot be played, the number of playback times, operations that may be performed on the media and device restrictions. A well known application of DRM is the OMA DRM system that Nokia implements on their Symbian Series 60 handsets for content purchased from the Nokia OVI store [48].

DRM functionality typically includes [49]:

- controlling tracking and distribution;
- protecting content against tampering during transmission;
- protection against unauthorised use;
- defining ways in which the content is consumed;
- facilitating the distribution of content across different paths, both offline (CDs, DVDs) and online (internet, peer-to-peer networks);

- monitoring content usage; and
- support for payment gateways.

DRM aims to be resistant against break-once, break everywhere (BOBE) [21]. If an attacker is able to circumvent security of a client in a DRM system and the same attack is applicable to other clients in the system, then the system is referred to break-once, break everywhere. DRM provides a maximum set of features that can ensure content protection. However, analogue signals always provide the final loophole for content piracy.

2.4.1 Microsoft PlayReady

Microsoft PlayReady is a DRM platform that applies business models to the distribution and use of digital content. PlayReady is able to protect music, video, ringtones, images and games, through purchase, subscription, rental, pay-per-view and preview [50]. PlayReady is backward compatible with Microsoft Windows Media DRM.

Entities in the PlayReady platform include content packaging servers, distribution servers, license servers, domain controllers and metering servers.

2.4.2 UltraViolet

On 20 July 2010, the Digital Entertainment Content Ecosystem announced UltraViolet, a platform for digital video content [32]. Hollywood movie studios are seeing a decline in sale of physical media (for example DVD and Blu-Ray) as consumers are moving to video-on-demand solutions [51].

Instead of a DRM, vendor or platform lock-in, UltraViolet is offering a platform that is open to various retailers, devices and DRM solutions. The purchase of any content that supports UltraViolet is stored in the digital library and gives the consumer rights to the content. The consumer is then able to either download this content or stream it from the UltraViolet digital library.

The UltraViolet platform consists of, the UltraViolet digital library, UltraViolet retailers, devices that are compatible with UltraViolet and the consumers [51].

UltraViolet also defined a file format for the platform called Common File Format and supports any of the DRM systems that are compatible with Ultra-

Violet. UltraViolet does not serve as a new DRM system to protect content, but rather provides a platform which supports multiple devices, DRM solutions and retailers, with the aim of giving consumers more freedom in how they purchase and consume media.

2.4.3 Irdeto ActiveCloak for Media

Irdeto [52], a company which develops content security for pay media companies, released ActiveCloak for Media in March 2011 [53]. ActiveCloak is a security solution that is based on Irdeto's core Cloakware technology which is used to combat software piracy.

Hackers typically have two approaches to circumvent software-based video security [54]. The first approach is to reverse engineer the source code in order to obtain sensitive information. This sensitive information can include cryptographic keys or certificates. The second approach is to tamper with the source code to modify the behaviour of the software with the outcome having access to the secured content.

ActiveCloak for Media protects against these attacks through a three step process. The first step is to provide a strong initial resistance, and this is achieved through techniques that attempt to protect and obfuscate code against reverse engineering. In the event of a breach the magnitude of the breach is reduced by diversifying the software. Irdeto implements this by developing software that is functionally identical but structurally different. Releasing diverse software means breaches only impact small selection of users, as apposed to the entire user base. Finally, if a security breach occurs, ActiveCloak for Media is able to recover from this and patch the software.

ActiveCloak for Media allows the developers to focus on the application and lets Irdeto handle the security. Furthermore, ActiveCloak for Media allows the developer to use other DRM solutions with ActiveCloak for Media. ActiveCloak for Media secures the content in a secure store and provides a single API to access it [55].

ActiveCloak for Media focuses on the application security, not only the content security. This is different to DRM which assigns rights to the usage of the content.

2.5 Summary

In a white paper published by the Convergence Culture Consortium at the Massachusetts Institute of Technology, de Kosnik [56] states that no legal site makes all popular TV shows available. She goes further to state that pirating TV shows has the following advantages over the legal alternative:

- a single point for searching;
- simple sorting;
- a single application and interface to learn;
- file portability;
- access to a global base of TV shows;
- personal archives; and
- low cost and free of commercials.

She finishes off to say that most individuals choose to pirate, not because it is free, but because it is the easiest manner in which TV shows can be watched online, with added features.

This should be kept in the back of our minds as we discuss the system and detailed design in the next few chapters. Although this platform is for legal purposes, it should encompass as many of the benefits that piracy offer. By doing this you provide a platform that, to the user, is as easy and feature rich as pirating but remains fully legal. This project does not set out to design the business case for the platform, but it does need to provide security to support a viable business case to operate the platform.

Chapter 3

Platform Design

The previous chapter described the background research required to design a security and containment platform for peer-to-peer media distribution. This chapter details the approach to designing this platform. The chapter provides an overview of the platform and lists the components that make up the platform, before detailing the design of each component and the design decisions that were made.

3.1 Design Approach

When designing a platform of this nature, we believe it is best to make the video object an integral part of the platform with the security and services wrapped around the video object.

The design of the security tries to find a balance between security and usage restrictions placed on the user. Two points to consider when designing the security are the following: firstly the video that is rendered on the screen will always provide a means for users to capture the analogue image, and secondly it can be assumed that it is easier for a user to obtain an illegal copy of the video from a video piracy service, than what it is to attack the video encryption scheme that this project proposes. Hence it is not feasible to create a security system that is impossible to break. As with sufficient security we can discourage almost all users of the platform from attacking the system's security.

We define a security system that fully (naively) encrypts the video file and allows the file to be distributed on and off the platform. By fully encrypting the video file we sacrifice efficiency and performance for the best security[1] and are resistant against perceptual attacks[2]. In order to decrypt the video file, the video file must be opened with the platform application and connected to the platform servers. With the user authenticated, the keys can be streamed over a low bandwidth Internet connection to the application and allow the video to be decrypted for playback in real time. Each key decrypts a segment of the video and requires a connection to a key server for the duration of the video playback. It is possible to place a server on a local network where no Internet connection is available and where is it feasible to do so. An example of this would be darknets.

3.2 Platform Overview

The platform can be divided into two components, the client-side applications and the servers. The client-side application is the application that the end user will interact with, through the use of the graphical user interface (GUI). The servers provide functionality to the client applications and are used to monitor the platform. An example of this functionally is the usage tracking server which is responsible for monitoring video playback on the platform. In Figure 3.1 an overview of the platform is illustrated with the various high level components. Although the various servers are shown as separate logical entities, they may reside on the same physical server. This is described in further detail in Chapter 4.

User

The "user" component in Figure 3.1 represents the client application that the end user will make use of to; search for downloadable content available on the platform, start downloads and play encrypted video content.

Peer

The "peer" components in Figure 3.1 represent other users on the platform that are possibly watching videos, downloading videos, as well as seeding media onto the local network.

[1]Refer to Table 2.1
[2]Refer to Section 2.2.

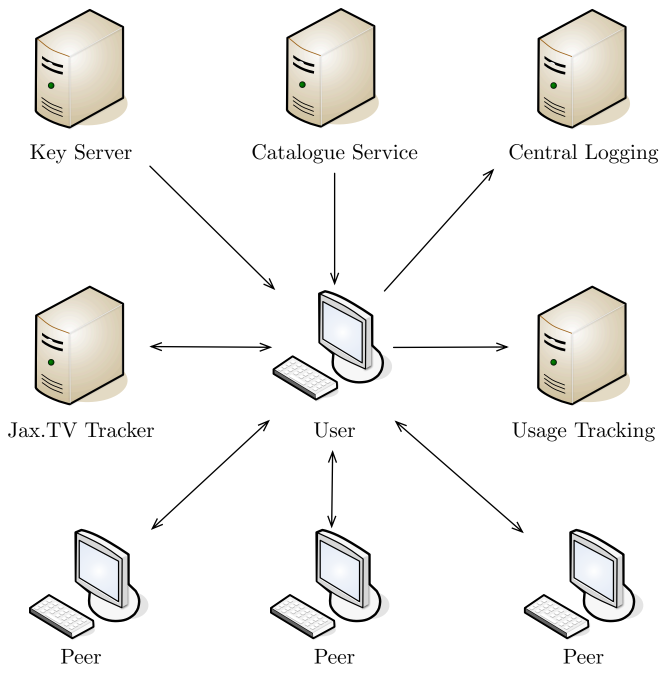


Figure 3.1: High level component overview of the platform.

Catalogue

The catalogue service maintains a collection of metadata about the videos on the platform. Typical metadata includes the title, rating and duration of the video. The catalogue server aids the user in discovering content on the platform. Additional information (that is not available to the user) is also stored on the catalogue server for use by the client application.

Central Logging

When working with a distributed platform such as this, debugging becomes tedious with the multiple remote client applications. The traditional manner of monitoring log files on various machines is replaced with a central logging service. If logging is required on one or multiple client applications, their logs are sent to a central server that is able to analyse, filter and aggregate the logs. This service is primarily for the development and testing phases of the project and therefore will only be used for informative log messages during production phase.

Usage Tracking

The usage tracking service collects information from the platform and separates it into two categories; statistical data that is collected over a long period of time and real-time information. The statistical data that is stored over a long period of time includes user login, playback and search information. The real-time information collected provides administrators with the ability to monitor, in real-time, which users are logged in, what videos the users are watching, and which videos the users have cached on their devices.

Jax.TV Tracker

The "tracker" component in Figure 3.1 is the tracker service developed as part of the Jax.TV implementation [16]. As described in Section 2.1.4, it is responsible for tracking which peers are seeding media onto the network and providing downloading peers with a list of peers that are seeding the files being requested for download.

Key Server

The key server is responsible for streaming the keys to the client application in order for the application to decrypt the encrypted video. All communication to this server has to be encrypted to prevent eavesdropping and secured to prevent unauthorised access to the keys.

3.3 System-wide Design Decisions

A set of system-wide design considerations were defined at the outset of the project. These include systems specifications, programming language and software standards.

3.3.1 Specifications

The following specifications were set out for the platform:

Modular in Design

The software design of the platform is intended to be orthogonal by decoupling the various modules. This means changes in future do not require a complete platform redesign.

Cross-platform

The client application must be designed to be cross-platform. The platform is targeted at the mainstream desktop operating systems; Microsoft Windows 7, Ubuntu Linux and Mac OS X.

Codec Independence

The platform should be designed to be video codec independent with relation to the encryption scheme. More specifically any codec can be used, provided that the application video player can support the codec.

Real-Time Tracking

The platform must be designed to track, in real time, the video content that is being consumed by the end users. This feature needs to provde platform managers the ability to monitor the acceptance of new media by the users, provide auditing and allocate license fees. The platform cannot support any business models without real-time tracking.

Multiple Distribution Modes

Although Jax.TV provides the primary mechanism of video file distribution on the platform, the platform is designed for African countries where Internet bandwidth is expensive. For this reason the platform should allow multiple distribution mechanisms, and be specifically designed to allow media to be distributed through flash drives and external hard drives. This takes full advantage of the benefits that piracy offers, such as file portability and personal archives [56].

DRM Avoidance

The platform should be designed not to use any existing DRM techniques, but rather be an alternative to DRM.

Data Security

The most important asset on the platform is the video content. Although it is listed last, securing the video content is the most important specification for the platform. During distribution, encrypted content is secure from the user. However, during playback the video has to be decrypted in order for the video playing application to render the audio and image sequences. Playback and the decryption of the video is therefore the weakest point in the platform. Attention has to be given to preventing the user from gaining access to the decrypted video data. With the video data in a decrypted form, the user is not restricted to the platform client application to view it and can distribute the video in its decrypted form. The user must thus be prevented from gaining access to the decrypted video data. It is because in this form, the user is free do with the video content as they please. This platform intends to provide a legitimate commercially viable alternative to piracy, not an aid.

3.3.2 Programming Language

Python [30] was chosen as the programming language to facilitate Jax.TV integration, to enable rapid prototyping and for its cross-platform capabilities. JaxTV was written in Python and using the same language for this platform makes integration easier. Python is well documented and widely supported, and because of this there are many libraries available on the Internet, making Python a prime candidate for rapid prototyping. Python is cross-platform and runs on Microsoft Windows, Linux and Mac OS X [57].

The disadvantage of Python is its speed, because it is considerably slower than other high level languages, such as Java, and cannot compete with C and other lower-level languages. However, it is possible to extend Python with C and C++, if necessary.

In addition to Python, the Qt framework was chosen to develop the GUI and, in particular, the video player. Qt (not to be confused with Apple Quick-Time or QT) is a cross-platform application and UI framework that includes bindings for a wide range of programming languages including Python [58]. The Qt framework is primarily developed by Nokia and is used for the development of applications for Symbian phones, but is also used to develop applications for desktop operating systems, including Microsoft Windows, Linux and Mac OS X. Through the use of the PyQt Python bindings, the Qt framework can be used with Python. Because of the number of operating systems that Qt supports, we believe it is most suited for this project. The Qt framework is designed with rapid prototyping in mind, therefore the Software Development Kit (SDK) includes an IDE and drag-and-drop UI designer. Qt includes various modules, such as a GUI module, networking module, SQL module, to name a few.

When selecting programming languages and frameworks, it is important to look for active development. Both Python and Qt are actively developed at present, which results in improvements and bug fixes to the language and framework.

3.3.3 Software Standards

Git [59] was chosen for distributed version control of the source code and configuration files, with a remote repository on GitHub.com. Data representation for both object parameters and stored files, was done using JavaScript Object Notation (JSON) where possible [31]. Python Fabric [60] was used to distribute the source code to the server and the clients.

3.4 Use Cases

A use case diagram for the platform is shown in Figure 3.2. Here the roles of two actors are defined: the administrator and the end user. The administrator is able to add new media to the platform through the encryption service; add, remove and edit metadata, revoke media from the platform and monitor the platform. The end user is able to search, download, watch and share media. Although the platform provides a search and download mechanism, users are able to directly share media with each other using any other distribution mechanism, such as flash drives and external hard drives. An important point to note is that the users can only decrypt and watch the video through the platform application.

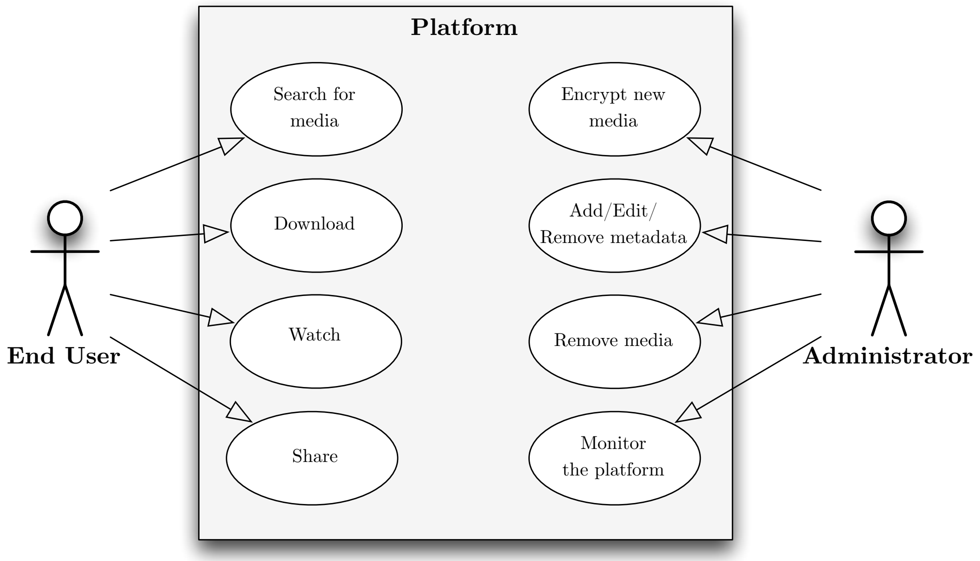


Figure 3.2: UML use case diagram for the platform.

3.5 Platform Communication

In order to integrate the various modules on the platform, a standard communication protocol is required. Representational State Transfer or better known as the RESTful style of architecture for networking systems [61], is best suited for the platform, instead of designing a protocol from scratch. REST is not a standard but rather a style, even though it does make use of various Internet standards, such as HTTP, URL and HTML. The REST interface provides a standard channel with which the clients can communicate with the server. This design allows for standard communication between the client and REST server, and allows the REST server to interface on various other protocols with the other server side services. These include the MySQL database, MongoDB database and Sphinx search engine. This is illustrated in Figure 3.3. The other advantage of this is that the REST server can standardise the presentation of data to the clients. JSON was chosen as the format to represent data when communicating between different components.

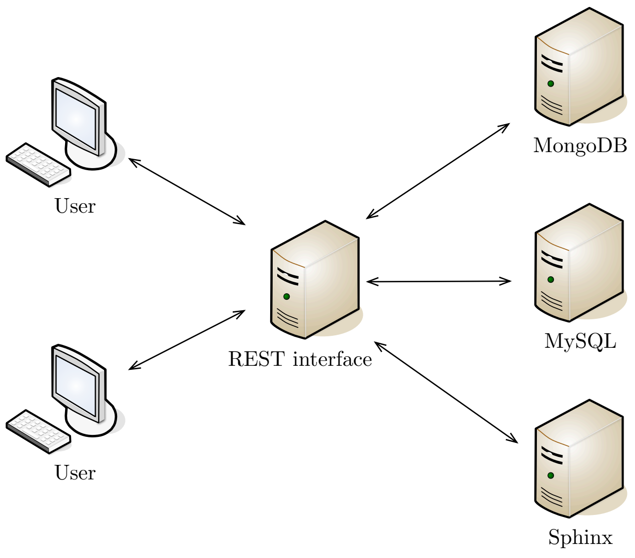


Figure 3.3: REST interface design.

An important attribute of the REST style architecture is that it is a stateless architecture, therefore the server does not maintain information about the clients that it is serving. The clients are responsible for sending all the necessary information for the server to process the request. REST also allows proxy servers, cache servers and gateways to be placed in between the client and the server to improve security and performance [62].

Django [63] was chosen for the REST interface, as it subscribes to the model-view-controller (MVC) software architecture. Other MVC frameworks are available, such as Ruby [64], however selecting Django allows the code base to remain in Python.

Security

One of the disadvantages of REST is that the Django server will respond to any valid HTTP requests, much like a request from a web browser. Traffic to the web server needs to be restricted to that of the application. The other disadvantage of this communication is that everything is transmitted in plaintext. To address these issues, the following security measures are put in place.

Firstly, all communication is encrypted with a Secure Sockets Layer (SSL) connection (HTTPS). This solves the problem of eavesdropping and man-in-the-middle attacks, but does not prevent anybody from anonymously accessing the web server. This is where user based authentication plays a role in preventing anonymous requests to the server. The user has to be registered with the platform in order to login in and access information. The problem with REST service and user based authentication is that the user is able to view the page information on the web server through their browser using their known credentials. The user would be able to view the web page that displays the video keys with their browser and obtain the keys to the video. In order to prevent this, a custom user agent string is created for the application.

A user agent string is included in the header of a HTTP request, that is used by the server to identify the web browser. Typically in HTTP traffic the user-agent includes the operating system of the browser, as well as the version of the browser. There are many user agents as different devices are running different browsers, and those running the same browser are often running at different versions or on different operating systems.

The web server of the platform is designed to only accept requests containing the user agents in a whitelist. The client application uses a unique user agent and the web server accepts every request it receives, checks the user agent in the header and rejects those not on the whitelist. In HTTPS, everything transmitted is encrypted, including the user agent, so it is not possible to view the raw network packets in an attempt to obtain the user agent. An overview of the communication security stack is shown in Figure 3.4.

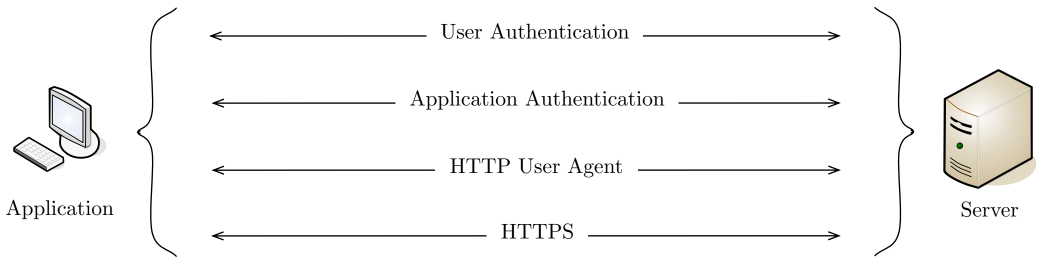


Figure 3.4: Platform security stack.

If the user agent is compromised, an update can be applied to the client applications to update the user agent, thereby preventing the Break Once, Break Everywhere problem [21]. Out-of-date versions of the application will be forced to update their user-agents to access the server.

3.6 Encryption

The encryption scheme is based upon the one-time pad, with the pad generated by a pseudorandom number generator, that closely resembles that of the unsynchronised mode stream cipher.

The original unencrypted video file is divided into blocks of identical size (with the exception of the last block that makes up the remainder of the file). Each block is treated as the plaintext message data in the encryption scheme. A seed is supplied to the pseudorandom number generator (PRNG) which produces a sequence that is the same length as the block. The seed is the key for the block and the sequence is the pad used to encrypt the plaintext.

The plaintext message data in the block is XORed with the one-time pad to produce the cipher text. The cipher text block is stored in a new file, but at the same position of the block in the original file. By repeating this process for each block in the original file, a complete encrypted file can be produced from the original. This encryption process is illustrated in Figure 3.5.

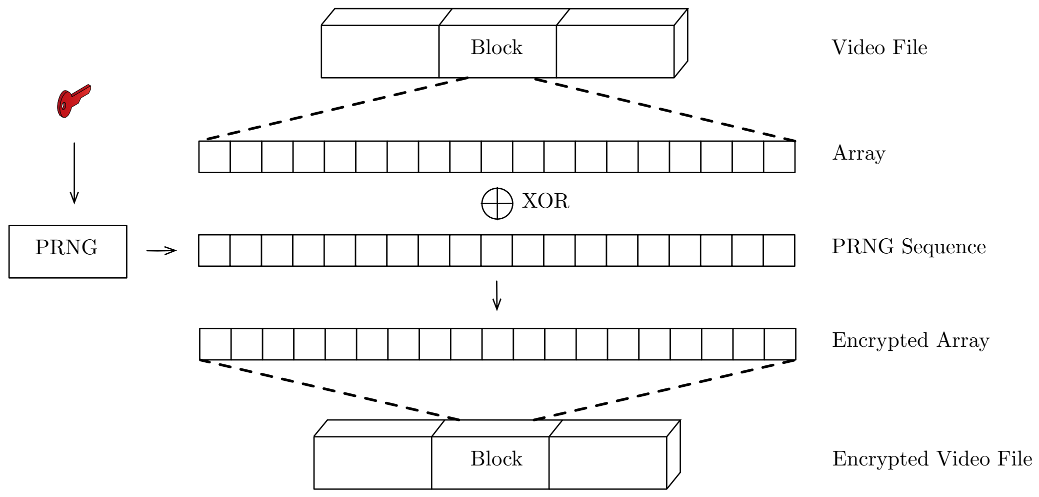


Figure 3.5: Design of the video encryption scheme.

Although this approach is seen as a naive form of video encryption, it has various advantages:

- The first is that the XOR is a native machine instruction on most processors, and this allows the XOR to be calculated quickly.

- The bandwidth required to stream the keys is minimal in comparison to streaming the entire video.
- This approach encrypts the binary data of the video file, therefore it does not restrict encryption to a specific video codec.
- The video is fully encrypted so that it is not comprehensible if played back to the user.
- The entire video file is encrypted, not just the header or I-frames, therefore the approach is robust against perceptual attacks[3].

This design meets the requirements set out by the project.

3.7 Video Player

The video player is the weakest point in the security design as this point requires the content to be decrypted in order for it to be rendered. Various approaches are available in designing the video player and are briefly described here.

It is possible to stream video on the local host using VLC media player [65]. Unfortunately, any data streamed on the local host is accessible by any program running on that host computer.

Another approach using VLC is to feed decrypted video data to VLC through a Unix named pipe. The problem with this approach is that the decrypted data is not secured but rather obscured. Also, Unix named pipes, as the name suggests, are only available on Unix-based platforms. Although Windows supports named pipes, using this approach would inherently mean having to develop software for Unix and Windows operating systems with different code bases. This would make the cross-platform functionality harder to implement and difficult to implement within the time frame of the project.

A higher level approach is to use third party libraries like PyMedia [66], pyglet [67] and ffvideo [68]. Because these libraries are high level interfaces, it is not possible to manipulate the video data before it is played back. Only a source can be specified for these libraries and the data needs to be accessible in order to decrypt it before it can be played.

[3]This is described in Section 2.2

A more complex approach would be to decrypt the video data in a parent process, and pipe the decrypted data to FFmpeg [69] running in a subprocess. This approach has many complexities and does not guarantee security through standard input and standard output.

One approach is to develop video player from scratch using OpenGL. Unfortunately, the time frame for this project did now allow this, even though it would allow for the maximum amount of security to be built into the video player. Using OpenGL, it is possible to create a custom video player that can decrypt the video data for playback in memory. By decrypting the video in memory, it is significantly harder for an attacker to obtain the raw video data.

Finally, Qt [58] was decided upon because of its support for rapid prototyping and the ability to manipulate data in memory. However, at the time of choosing Qt it was not known that Qt does not allow direct access to the media being played by the video player. This is discussed in detail in Section 5.2. It is important to remember in making this decision that the nature of the project is to develop a prototype and not a full scale production application.

3.8 Summary

This chapter described the platform and the typical use cases for it, as well as the design decisions that were made up front. It further detailed the designs of the core components, namely the server, the encryption scheme and the video player. Before continuing to the implementation of the project, it is important to note that the object is to develop a working prototype and not a full scale production platform with native applications for every operating system and device.

Chapter 4

Server Implementation

This chapter describes the implementation of the server and all the services required to operate the platform. The server can be subdivided into six components and these are illustrated Figure 4.1.

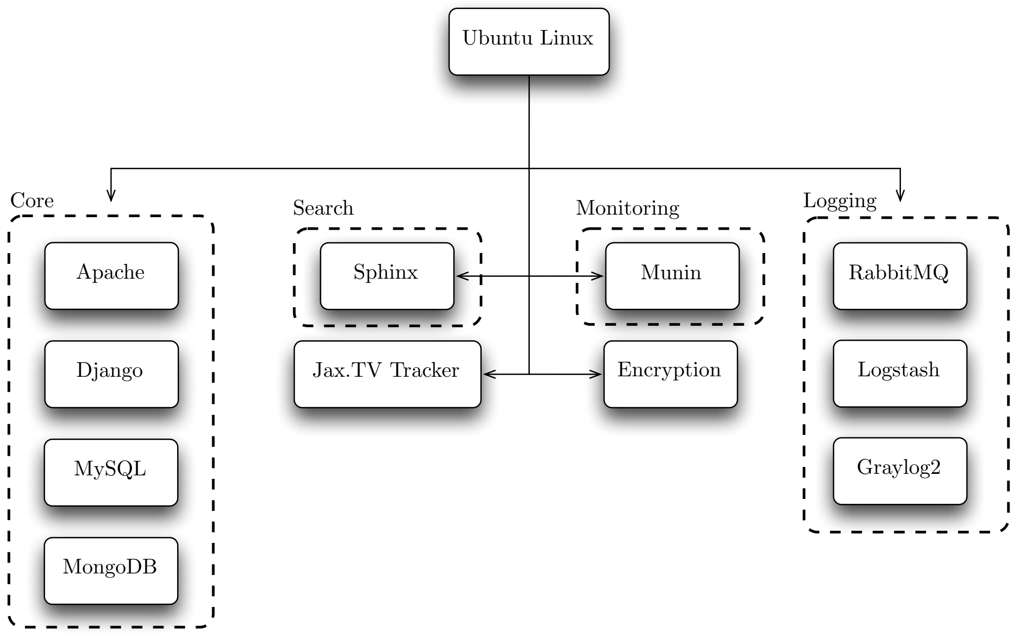


Figure 4.1: Component overview of the server infrastructure.

4.1 Hardware

The server used for the prototyping is running Ubuntu Linux Server [70] as a guest operating system on VMware vSphere [71]. vSphere is a cloud computing virtualisation operating system, that allows for multiple guest operating systems to share the same hardware resources. The guest operating system is equipped with a 2,3 GHz vCPU and 1024MB of RAM. For the purposes of prototyping all the services such as Django, the encryption and central logging are hosted on this virtual machine. This greatly simplified the implementation phase of the project, instead of hosting each service individually.

4.2 Core

The core of the server is made up of two components, the REST interface and the database systems. The REST interface is responsible for client communication with the server, while the database systems are used to store information for the operation of the platform (excluding the video data).

The RESTful interface is implemented with Django [63], which is hosted on Apache web server [72]. Django is served over an HTTPS Secure Sockets Layer (SSL) connection which prevents man-in-the-middle attacks, as well as preventing third parties from viewing the communication between the client and the server. As stated earlier, Django subscribes to the model-view-controller (MVC) software architecture, so the models defined are translated into entries in the backend database. The data is stored on MySQL server, using the Django MySQL backend engine. Django provides various database engines, and interfaces directly with the database. This allows the programmer's focus to remain on the Django models and not the actual SQL statements. MySQL can easily be replaced with another database server. In addition to MySQL, MongoDB is used to store real-time data for usage tracking. This is explained in detail in Section 4.7.

The core of the server has two roles on the platform; to provide a communication channel for the client application and for storing information about the videos. Information that is stored in the database is used by the client application to play back the video and to create a catalogue of video metadata. All this information is stored using Django models and is explained below.

4.2.1 Django Models

The Django model is a Python class that is created to represent the data, and each model's data is stored in its own table within the database. The model contains attributes and methods, while Django provides the following to the model:

- a subclass of the Django Model class;
- a database column for each attribute of the model; and
- an automatically generated API to access the database.

Even though Django creates an API for the models through which data is modified in the database, it is possible to directly interface with the database and modify rows. When Django then uses the API, the modified data is retrieved. Figure 4.2 shows the model classes and each is discussed below:

Video

The Video model is used to represent each video on the platform and stored in the *table_videos* table on the MySQL server. The attributes for the Video model include the name, availability, file name, video ID (*info_hash*), sliding window length, chunk size, number of blocks and the timer synchronization[1] of the video. The info hash attribute is the primary key for the Video model and is used as the identifier for videos on the platform.

VideoKey

The VideoKey model represents each encryption key stored in the database. The attributes for the model include the ID, the key itself, the index and the corresponding video ID. The index corresponds to the block number that the key was used to encrypt.

SphinxId

The SphinxId model stores an ID related to a video ID. Sphinx requires a primary key identifier field for each item it is required to index[2]. Because the

[1]Discussed in Section 5.2
[2]Sphinx is described in detail in Section 4.6

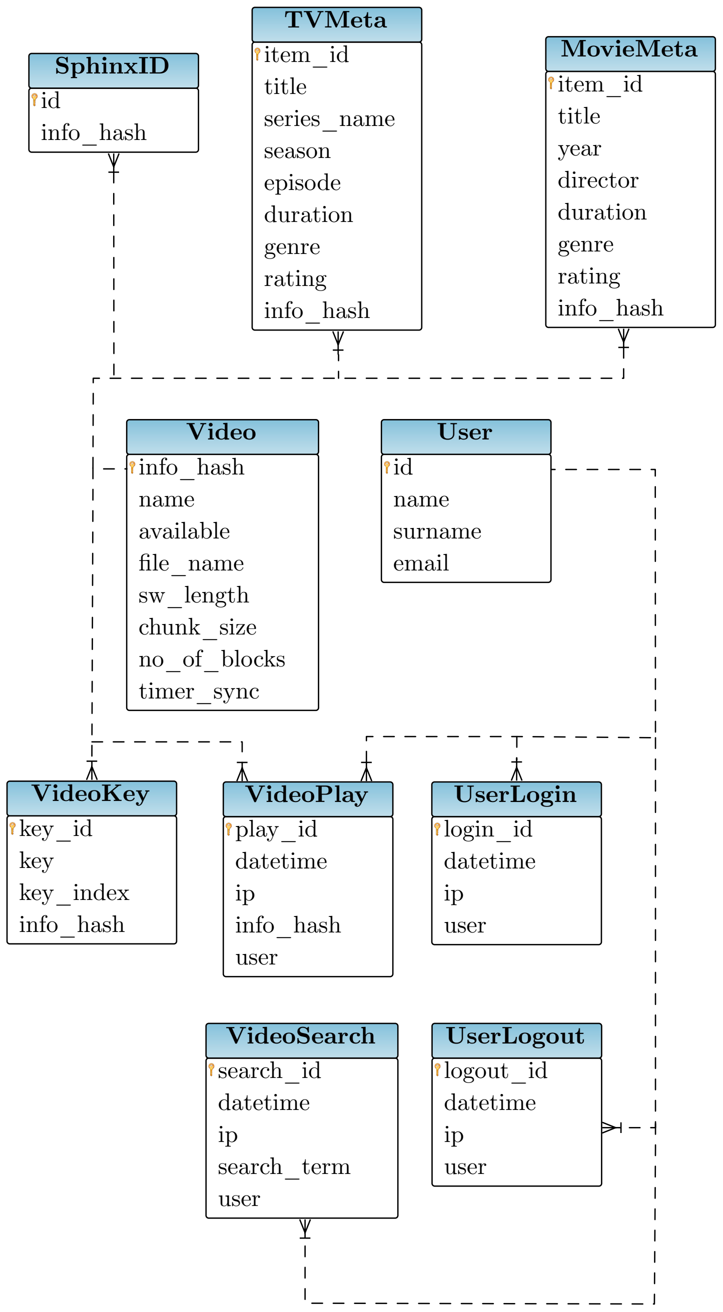


Figure 4.2: UML class diagram of all the Django models.

Video model already has a primary key in the form of the video ID, a separate model is created for the Sphinx ID. Creating a separate model for the Sphinx keeps the design modular and allows for Sphinx to be replaced with another search engine without affecting the Video model.

TVMeta

In order to store metadata for the videos two models were created, one for TV shows and one for movies. The TVMeta model includes the typical metadata representing a television show, that is, the title, series name, season number, episode number, duration of the show, genre and rating. The model also includes an identifier and the corresponding video ID.

MovieMeta

The movie metadata is similar to the TV metadata and includes the title, year, director, duration, genre and rating. Additionally an index and video ID are included in the model.

VideoPlay

This model stores information for each video that is played on the platform as part of the usage tracking. An identifier is created and stored along with the date and time, IP address where the video was viewed, the video ID, as well as the identifier of the user who watched the video.

VideoSearch

This Django model stores information for each search that is performed by a user on the platform. An identifier is created and stored along with the date and time, the IP address where the search was performed, the search term and the user that performed the search.

User

The user model is built into the Django framework and is used here to represent users on the platform.

UserLogin and UserLogout

These two models are used to store login and logout information of the users on the platform, respectively. A login or logout identifier is created and stored along with the date and time, IP address of the action and the user identifier.

4.2.2 Django Views

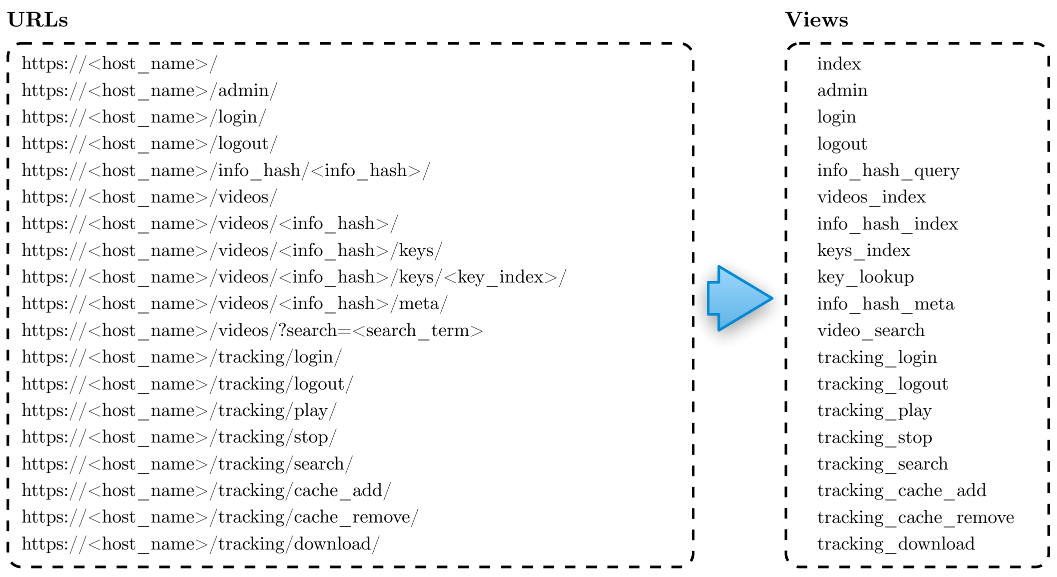
Django views, which make up the view in MVC, are responsible for returning an HTTP response to the requesting client. A list of URLs are defined and associated with a view. The view is a function that performs an operation and returns an HTTP response. The client application accesses the various views through the appropriate URL. For example, a GET request to `https://q.ml.sun.ac.za/videos/` lists all the videos on the platform by accessing the Video model API, building a list, formatting the list as JSON and returning an HTTP response. Although browsers may be the type of user agent most familiar to users, HTTP was designed to transport any type of resource that contains information and uses the Content-Type HTTP header identify the information format [62]. Because HTTP is not only limited to HTML, it was decided to use JSON to transmit information between client and server as set out in the system-wide design decisions in Section 3.3.

```
{
    "valid": true,
    "data": {
        "matches": 1,
        "videos": [
            {
                "name": "Family Guy 605",
                "file_name": "familyguy_605.mov",
                "sw_length": 10128,
                "info_hash": "f168eb05e64b2c93a3ec1f0c84d849e4"
                    ,
                "no_of_blocks": 64,
                "timer_sync": 40288,
                "chunk_size": 65536
            }
        ]
    }
}
```

Figure 4.3: A typical JSON formatted HTTP response.

The advantage of a REST interface is that information can be presented to the client in standard format, regardless of the service being accessed. The views are designed to present standard JSON format for all requests. A typical response for a video search is illustrated in Figure 4.3. Each response has a *valid* field. The views are implemented in such a manner that the valid member is always present in the response. If no valid data can be returned then the only member present is the *valid* pair with the value *false*. If however, valid data is returned the pair's value is true and the following member will be the actual data returned. Although HTTP error codes could be used to signal invalid data, this approach is more flexible because it allows the invalid response to be extended with additional information in future.

Figure 4.4 maps the available URLs to the corresponding views and a brief description of each view is given:



URLs	Views
https://<host_name>/	index
https://<host_name>/admin/	admin
https://<host_name>/login/	login
https://<host_name>/logout/	logout
https://<host_name>/info_hash/<info_hash>/	info_hash_query
https://<host_name>/videos/	videos_index
https://<host_name>/videos/<info_hash>/	info_hash_index
https://<host_name>/videos/<info_hash>/keys/	keys_index
https://<host_name>/videos/<info_hash>/keys/<key_index>/	key_lookup
https://<host_name>/videos/<info_hash>/meta/	info_hash_meta
https://<host_name>/videos/?search=<search_term>	video_search
https://<host_name>/tracking/login/	tracking_login
https://<host_name>/tracking/logout/	tracking_logout
https://<host_name>/tracking/play/	tracking_play
https://<host_name>/tracking/stop/	tracking_stop
https://<host_name>/tracking/search/	tracking_search
https://<host_name>/tracking/cache_add/	tracking_cache_add
https://<host_name>/tracking/cache_remove/	tracking_cache_remove
https://<host_name>/tracking/download/	tracking_download

Figure 4.4: Django URLs that match to corresponding Django views. Each view returns an HTTP 200 status code on a successful request.

index

The index view for the server that displays the date and time. After a successful login, a request is forwarded to this view.

admin
This is the built-in admin view from Django as described in Section 4.9.

login
This view uses an HTML template to produce a page for user to log in.

logout
This view uses an HTML template to log the user out.

info_hash_query
This view returns a *valid:true* if the video ID is available. Note that even if the queried video ID is valid on the platform, the video needs to be marked as available for the *valid* field to be returned as true.

videos_index
This view returns all the videos stored in the database but is only used for debugging.

info_hash_index
This view returns the Video model's data for the specified video ID.

keys_index
This view lists all keys, as well as the indexes for a specific video. It is only used for development.

key_lookup
This view returns the key as well as the index for the specified index of a video.

info_hash_meta
This view returns all the metadata for the specified video. The metadata can be either for a TV show or a movie, and this is determined by the view.

video_search
This view returns the results from a Sphinx search on the specified search term. Sphinx is explained in Section 4.6.

tracking_login
This view is used in conjunction with a POST request to store tracking information for a user login.

tracking_logout
This view is used in conjunction with a POST request to store tracking information for a user logout.

tracking_play
This submitting a POST request to this view, the tracking system is able to track video playback on the platform.

tracking_stop
This view is similar to the view above but is used to track when video playback stops.

tracking_search
This view is used to track search queries that are performed by the users. The client application submits a POST request to the view with the search query.

tracking_cache_add
When a client application completes a download, it submits a POST request to this view in order for the system to track where video files are cached on the platform.

tracking_cache_remove
This view is used to track when a video file is no longer cached by a client application on the platform.

tracking_download
Every time a client application starts downloading a video file, then the client application submits a POST request to this view in order for the tracking system to track a new video download.

4.2.3 Security

In order to meet the security design[3] an SSL connection needed to be guaranteed, user authentication implemented and a white list for user agents created.

In order to guarantee an HTTPS connection to the REST interface, Apache Web Server is configured to only serve the Django web framework over port 443

[3]Refer to Section 3.5.

(the default HTTPS port) with the SSL engine enabled. Refer to Chapter B.1 for full configuration details of Apache.

Django has a built-in authentication system for handling users, groups, permissions and session cookies. Enabling this system in Django provides an integrated approach to user authentication and management on the platform. By placing the *@login_ required* decorator before each view, Django will request authentication before responding to the client's request. If the user has already authenticated and the session cookie has not expired, re-authentication will not be required. The expiration time for the session cookie is set to 10 minutes.

For user-agent white-listing, a list is created in the constants.py file that contains the global constants and settings for the server. Each view receives the HTTP request as an object. This object contains a meta dictionary in which the HTTP user agent string is stored. The view checks if the user-agent exists in the global white-list. If not, a JSON response along with an HTTP 200 status code is returned. The user-agent testing is demonstrated in Figure 4.5.

```
if request.META ['HTTP_USER_AGENT'] in Constants.
   JAXTV_USER_AGENT:
```

Figure 4.5: User agent whitelisting check.

4.3 Encryption

The encryption module is the entry point for new media onto the platform. This module allows platform administrators to load new video content onto the platform and enters metadata about the video into the database. This metadata is necessary for client application to know how to decrypt the file. This section looks at the entire encryption module before detailing the encryption procedure.

4.3.1 Encryption Module

The encryption module (encryption.py) performs many operations, one of which is the actual encryption of the original video file. Figure 4.6 is a flow

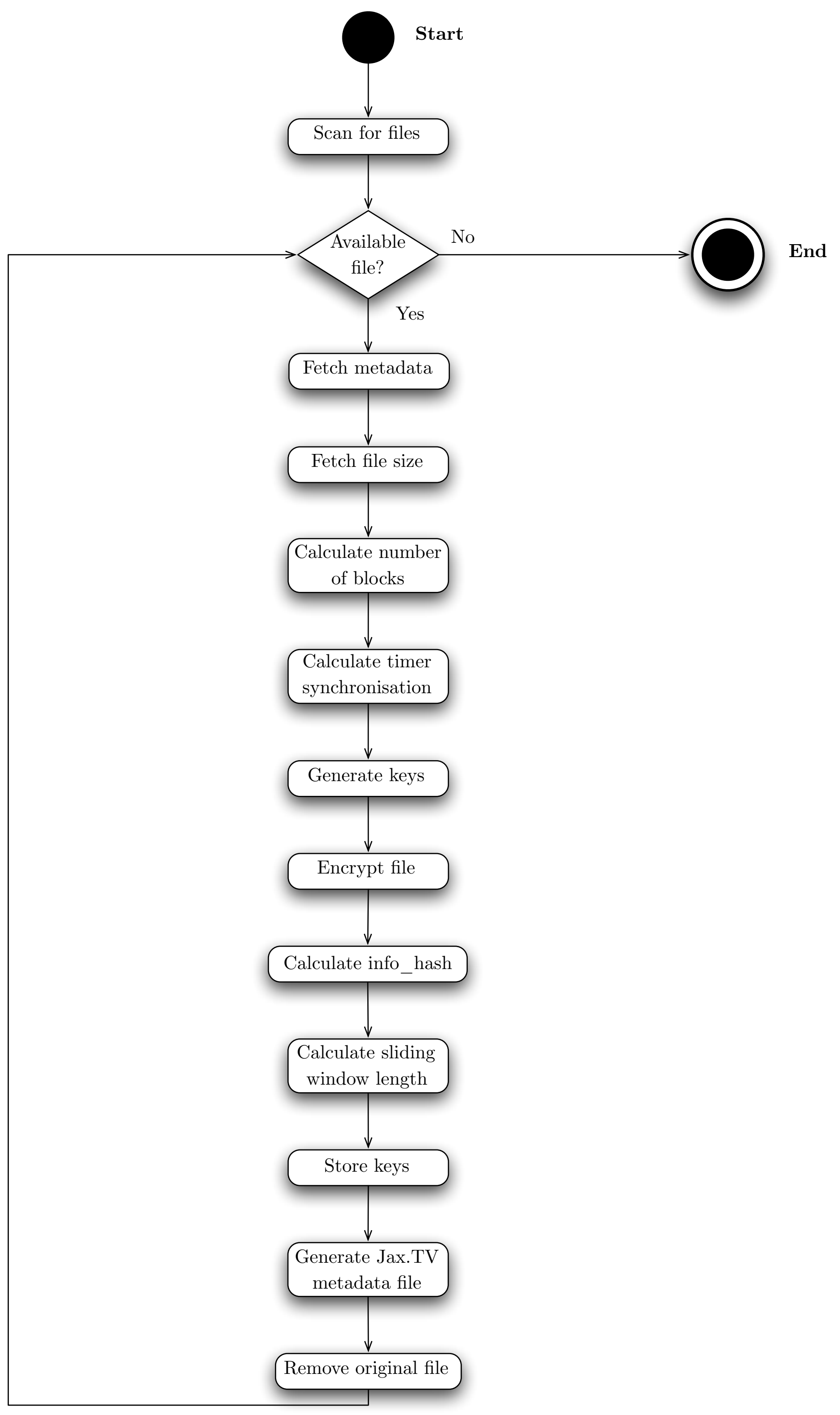


Figure 4.6: Encryption module flow diagram.

diagram of the steps that the encryption module executes.

The encryption process begins by the platform administrator copying the videos into the unencrypted folder. By creating a duplicate of the video, it is possible to revoke the video from the platform, re-encrypt it with new keys and distribute it back onto the platform. The encryption module starts by scanning for unencrypted video files in the unencrypted folder. After this, the procedure starts for each file in the directory iteratively. For each file the metadata information is determined, this includes the file name and length of the original video, the chunk size (which is set to platform specification, as listed in the server constants file). Next, the file size and number of encryption blocks are determined. By dividing the file size by the encryption block length, it is possible to calculate the number of encryptions blocks required to encrypt the file. This encryption block length is fixed for the platform and also specified in the constants file. Finally the timer synchronisation for playback[4] is calculated by analysing the metadata of the video and retrieving the total length of the video clip in seconds. This is multiplied by 1000 and divided by the number of encryption blocks to provide a timer synchronization in milliseconds.

A key is then generated for each block of the video and with all the information above determined, the actual encryption can proceed and is detailed in full in Section 4.3.2. Once the encryption of the file is complete, the MD5 hash sum can be calculated on the encrypted video file. This hash sum will be stored as the video ID (*info_ hash*) and serve as the identifier of the video on the platform. This makes it easy for any device to validate a video on the system by calculating the MD5 hash sum and querying the server with the video ID. The last attribute to calculated is the sliding window length. This is done according to the GoalBit specification.

With all the additional information calculated, the information along with the keys can be securely stored in the database and the Jax.TV metadata file can be created for the uploader. All information that is written into the database makes use of the Django model API. Finally, the video file in the unencrypted folder is erased from the server.

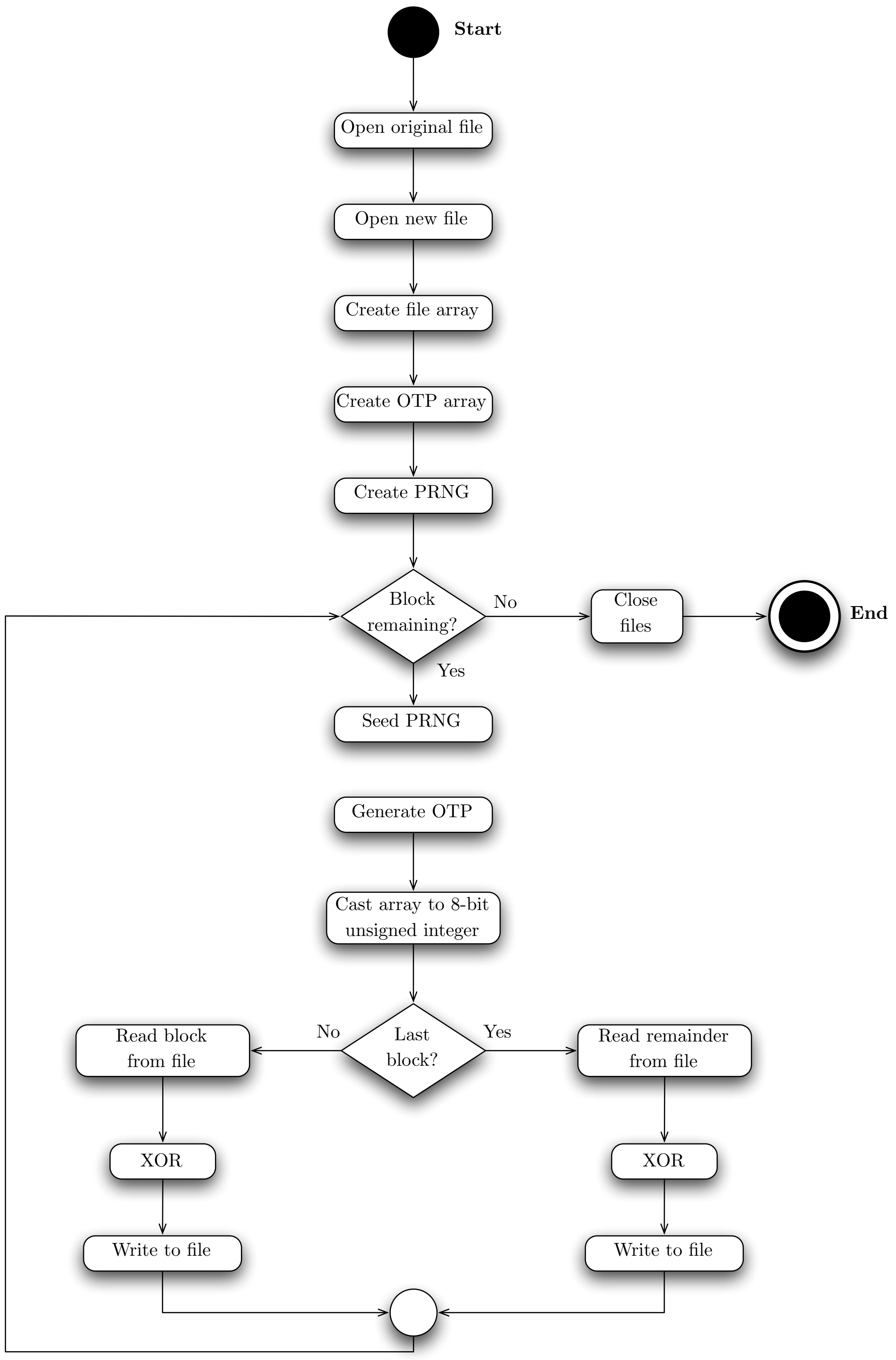


Figure 4.7: Flow diagram detailing the implementation of the encryption procedure.

4.3.2 Encryption Procedure

The encryption procedure relies heavily on the NumPy Python library [73]. NumPy is a package for scientific computing on Python with a powerful toolkit of numeric operations. Numpy features an N-dimensional array object, although for the requirements of the project it will only be used as a one-dimensional array.

NumPy's random sampling routines include a random number generator that uses a Mersenne Twister [74] pseudorandom number generator (PRNG). Although the Mersenne Twister is not cryptographically secure [75], it is faster than other PRNGs. The Mersenne Twister produces a linear recurring sequence, and allows the present state of the Mersenne Twister to be determined by monitoring a sufficiently large output [74]. For the purposes of the platform and way in which the Mersenne Twister is used, it does not create a security vulnerability. This is valid for the following two reasons: firstly, the key used to seed the Mersenne Twister is encrypted and not accessible by the user, secondly, the pad produced by the Mersenne Twister, as well as the original video is not accessible to the user. With only the encrypted video content, the user is unable to reproduce to the pad or the original video.

Figure 4.7 shows the flow diagram for the encryption procedure. Two files are opened, the original video file (in binary read-only mode) and the new file that is to be encrypted (in binary write mode). This is followed by the creation of two zero-filled NumPy arrays, each the length of the encryption block and using the 8-bit unsigned integer data type. Next, the Mersenne Twister PRNG object is created and from this point the encryption procedure steps through iteratively until the last block. For each block the PRNG is seeded with the key for the block.

With the PRNG seeded, the PRNG produces an array of random numbers between 0 and 255. This range is chosen to represent a byte value, in order to perform a bitwise XOR with the corresponding byte from the original file. The length of the array produced is that of the encryption block except for the last block, here it is remaining length of the final block of the file. With the array produced it has to be cast from a 32-bit unsigned integers to 8-bit unsigned integers. The encryption block from the original file is read into an array as 8-bit unsigned integers. With the data in both arrays, NumPy

[4]Discussed in Section 5.2

performs a bitwise XOR operation on both arrays and writes the result to the corresponding location in the encrypted file.

The NumPy PRNG can be seeded with a 32-bit unsigned integer and this is then stored as the key for the specific encryption block in the video file.

4.4 Jax.TV Tracker

The Jax.TV tracker has two roles on the network. The first is to maintain a list of peers along with each video that the respective peers is seeding onto the network. The second role is to provide a new downloader with a list of peers that are available to upload the requested video.

When a peer starts uploading content onto the network it announces itself to the tracker. During this communication the peer sends its unique client identifier, as well as the video ID of each video it has available to upload.

The client identifier is made up two characters for the client identifier, four digits for the version number and followed by random numbers. The client identified is unique for each client and an example of one is JT0001-619197444203.

The Jax.TV tracker (jax-tracker.py) is implemented using the Python basic HTTP server and is executed from the command line.

4.5 Super-Peer

The super-peer is a regular peer with the exception of having a high upload bandwidth and is responsible for seeding the newly encrypted video content onto the network. After the video is encrypted by the encryption server, the encrypted video data, along with the Jax.TV metadata files must be copied to the super-peer. The super-peer is running Jax.TV client in uploading mode only. The Jax.TV announces the newly encrypted video files to the tracker and is able to immediately start seeding the new video files onto the network.

The Jax.TV client is invoked by running the jaxtv.py command-line interface (CLI) application. Upon launching the CLI application, it scans the metadata folder for Jax.TV metadata files. These are used by the Jax.TV client to indicate the presence of the video file and the video ID is then announced to the tracker.

4.6 Video Search

Video discovery is said to be an important feature for users on a video entertainment platform. Netflix attributes 75% of videos watched on their platform to their video recommendations [76]. Torrents make use of a website that allow users to search for torrents [27], and DC++ clients have the ability to search for files within the application [77]. The same applies to legal services like the iTunes store or Netflix. Without a way in which the user can discover new videos, the user will have to solely rely on friends to share videos from the platform. For this reason the platform requires a full text search server.

Sphinx [78] is as a full text search server that is supported on various platforms and is open source. Sphinx can run on the mainstream operating systems such as Linux (Ubuntu and Red Hat), Windows and Mac OS X, as well as FreeBSD and Solaris. Sphinx has many features that make it the ideal choice for the platform. Sphinx is able to index and search data both on the fly or data that is stored in databases. Sphinx is also not limited to SQL databases and can search and index NoSQL storage as well. Sphinx allows searches to be limited to a single field or a subset of fields in a database table.

Sphinx supplies three Application Programming Interfaces (APIs), SphinxAPI, SphinxSE and SphinxQL. SphinxAPI includes a set of the client API libraries for languages like Python, Ruby and PHP in order to access the Sphinx search daemon. SphinxQL allows the client to query Sphinx directly with MySQL statements. The Python libraries are ideal for the integration with Django.

4.6.1 Configuration

Sphinx is configured to index the data directly in the database where Django stores the model data. In order for Sphinx to index a table, it requires that first column contain the primary key ID field. Because the Django video model already contains a primary key (the video ID), a separate Django model was created for the Sphinx ID. This model only contains the ID primary key in the first column and the video ID with a one-to-one relationship with the video model. By separating the Sphinx ID from the Video model, it allows the platform to be modified later by replacing Sphinx with another search engine if the need would arise. Because of the modular approach, removing

Sphinx would have no impact on the Video model. In order to create a SQL fetch query for Sphinx with a split table design, a join operation is done on *table_ids* and *table_videos*.

In Figure 4.8 the main fetch query for Sphinx is defined. Sphinx is configured to only indexes the name column in the Video model. For the complete configuration refer to Appendix B.2.

```
SELECT `table_ids`.`id`, `table_videos`.`name` FROM `
    table_videos` INNER JOIN `table_ids` ON `table_videos
    `.`info_hash`=`table_ids`.`info_hash`
```

Figure 4.8: The SQL statement used by Sphinx to index the database.

4.6.2 Searching

A wrapper class (SphinxClass) was created for the extensive Sphinx API. This class is stored in `sphinxwrapper.py` within the `db` sub directory. The wrapper configures and makes use of only the necessary components within the Sphinx API and includes additional functionality required to integrate with Django.

The output from the Sphinx API is given in Figure 4.9 and inspection of the results will show that the only video data returned is the ID and the fields that were indexed. No additional video data is returned in the search results. Sphinx can be configured to allow additional column data to be returned but this is strongly discouraged for performance reasons. Instead, the Sphinx manual recommends that an additional SQL lookup be done with the row identifiers. The wrapper class uses the returned identifiers to lookup the video data on each identifier returned. This information is structured and returned as JSON to the calling function.

In order to perform a full text search on the query, the client makes a request to the video_search Django view. The Django view uses the wrapper class and returns the data, in JSON format, to the requesting client.

```
{
    'status': 0,
    'matches':
        [
            {
                'id': 2,
                'weight': 1,
                'attrs': {}
            }
        ],
    'fields':
        [
            'name'
        ],
    'time': '0.000',
    'total_found': 1,
    'warning': '',
    'attrs': [],
    'words':
        [
            {
                'docs': 1,
                'hits': 1,
                'word': 'family'
            }
        ],
    'error': '',
    'total': 1
}
```

Figure 4.9: The typical output of search results produced by the Sphinx API.

4.7 Usage Tracking

Usage tracking is an important part of the platform for both content providers and system administrators. It provides system administrators with a gauge to measure usage on the platform and it enables content providers to monitor video consumption and the adoption of new videos. It also allows billing systems to later be integrated into the system.

For the specific tracking that was required by the platform, no off-the-shelf solutions were available. A custom tracking system needed to be developed. Tracking on the platform is divided into two categories, namely real-time statistics and long-term statistics.

Tracking is performed by the client making the HTTP POST request to the REST interface using the tracking URLs defined in Section 4.2.2. Every time the client application performs an action that needs to be tracked it accesses the respective view on the REST server. The client application does not expect any response, but through using an HTTP POST request, the server is able to track a specific action.

Real-Time Statistics

Real-time statistics represents information at any point in time on the platform. Real-time data tracks users that are logged onto the platform, videos that are currently playing and video files that are cached on the network.

All real-time statistics are stored in collections within MongoDB. MongoDB is a high performance, scalable, NoSQL database that is open source [79]. MongoDB stores JSON-style documents in collections. For the platform, three collections exist, *logged_in*, *playing*, and *cache*. When there is no available information, e.g. no users are logged in, then the collection will be empty.

Each entry in the *logged_in* collection is associated to a user, and includes a list of IP addresses where the user is currently logged in. An entry in the *playing* collection is associated to a video ID and includes a list of users currently watching that video. Finally, each entry in the *cache* collection is associated to a video ID and includes a list of all the IP addresses where the video file is cached.

All the statistics listed above are available through the MongoDB shell accessible via the MongoDB server. If a document has no valid data in it then it is removed from the collection. For example, if user1 is not logged into the platform then there will be no document for user1 in the *logged_in* collection. An example of the MongoDB collections is given in Figure 4.10.

Long-term Statistics

Certain statistics are required to last for long durations, even though some of these are included in the real-time statistics. These statistics include user login and logouts, video playback, searches performed by users and video files that are downloaded by users.

All long-term statistics are stored in MySQL with the help of Django models (described in Section 4.2.1). Unlike the real-time statistics which are concerned with the number of users logged in, the long-term statistics track the

```
logged_in
{user: "user1", ip: ["10.10.11.1"]}
{user: "user2", ip: ["10.10.11.2", "10.10.11.3"]}

playing
{info_hash: "abcd1234", user: ["user1", "user2"]}
{info_hash: "efgh5678", user: ["user3"]}

cache
{info_hash: "abcd1234", ip: ["10.10.11.1"]}
{info_hash: "efgh5678", ip: ["10.10.11.1", "10.10.11.2"]}
```

Figure 4.10: The structure of documents within the MongoDB collections.

date, time and the IP address where the user logged in. These records accumulate over time as videos are played or users log in and out of the system.

The statistics gathered of videos that are searched for and videos that are downloaded, enable content providers the ability to monitor the popularity and demand for video content on the platform.

4.8 Central Logging

A central logging system is important when developing a platform that has applications executing on multiple hosts. Logging is not only useful debugging information when developing software, but also for applications that are being executed on servers. Having a central point to monitor logs means that it is not necessary to log in remotely to hosts and monitor individual log files. Instead, logs can be collected centrally and allow for filtering and alerts to be placed on the items, improving efficiency in development.

In order to design a central logging system, a combination of various technologies are required to achieve the overall central logging system. The design uses RabbitMQ message queueing, Logstash and Graylog2. The central logging design is illustrated in Figure 4.11.

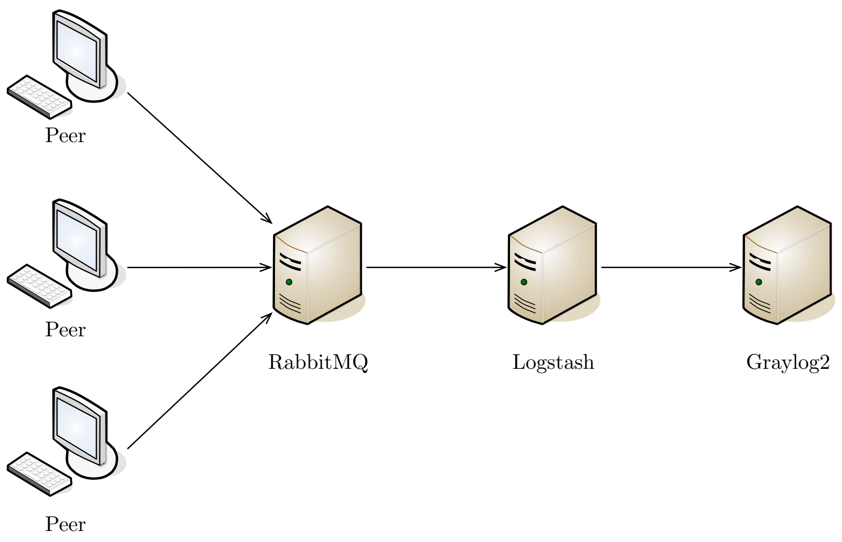


Figure 4.11: An overview of the central logging system.

RabbitMQ

The first objective is to provide a central point where messages can be sent to. Here RabbitMQ [80] provides the equivalent of a post office, allowing the remote applications to send their logs to a central point, from which it can be routed. RabbitMQ is an open source message broker service that uses the Advanced Message Queueing Protocol (AMQP) [81]. RabbitMQ consists of the following components: a producer which creates the message and sends it to the designated exchange, the exchange which uses routing keys provided with the messages to route the message to the correct message queue, the message queue which implements a first in, first out (FIFO) process, whereby consumers remove items one-by-one from the message queue. This is illustrated in Figure 4.12

The logs are sent as JSON encoded messages containing the host name of the remote machine, the IP address and the actual log message as illustrated in Figure 4.13.

Logstash

Logstash is used as as a RabbitMQ consumer to collect messages off the RabbitMQ queue. Logstash [82] is used to manage logs through parsing and storing the log events. Although Logstash can search and display logs, the user in-

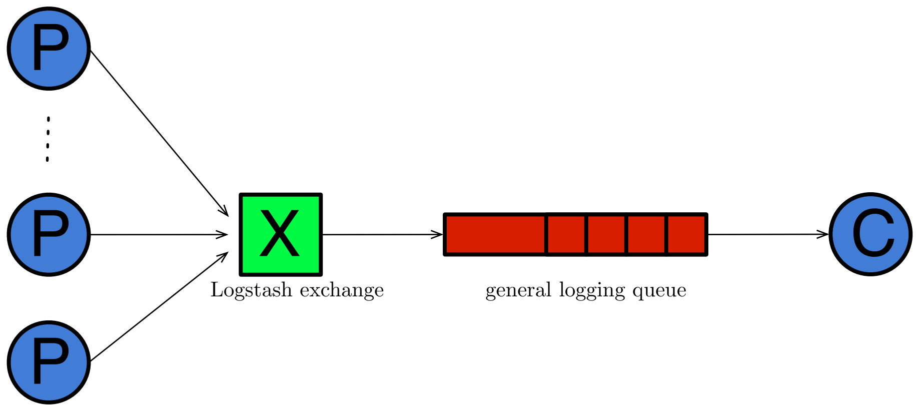


Figure 4.12: An overview of the RabbitMQ queuing system.

```
{
    "machine": "host name",
    "ip_address": "ip address",
    "message": "the actual log message"",
}
```

Figure 4.13: JSON encoded log message.

terface is simplistic and has very few features, therefore Logstash is only used to parse and mutate the logs. The visual representation will be performed by Graylog2. Logstash is able to take multiple inputs, like AMQP, stdin and files, perform various filters on the various inputs and output the logs to multiple outputs[5]. Logstash outputs the logs to Graylog2 on the platform but it can be extended to other log visualisation tools like Graphite or stored in MongoDB. The configuration for Logstash is given in Section B.3

Graylog2

Graylog2 is a tool used for analytics, alerts and monitoring through the use of logs [83], with the ability to search through logs. Graylog2 receives the log entries from Logstash and stores them in a MongoDB collection and makes the logs available through a web interface. Graylog2 allows the creation of streams by defining rules for which Graylog2 filters the main stream of logs. Various streams are configured in Graylog2 and these include the Jax.TV

[5]Refer to the documentation on the Logstash website for all the inputs, filters and outputs.

tracker log, the Django info log, the Apache error log, the encryption log and most importantly the logs of the client applications.

4.9 Admin Portal

With all these server side services running, a convenient landing page was created to help redirect administrators to the various administration and monitoring tools. For security, all the administration and monitoring tools (with the exception of Logstash) require administrator authentication. A screen shot of the admin portal is shown in Figure 4.14.

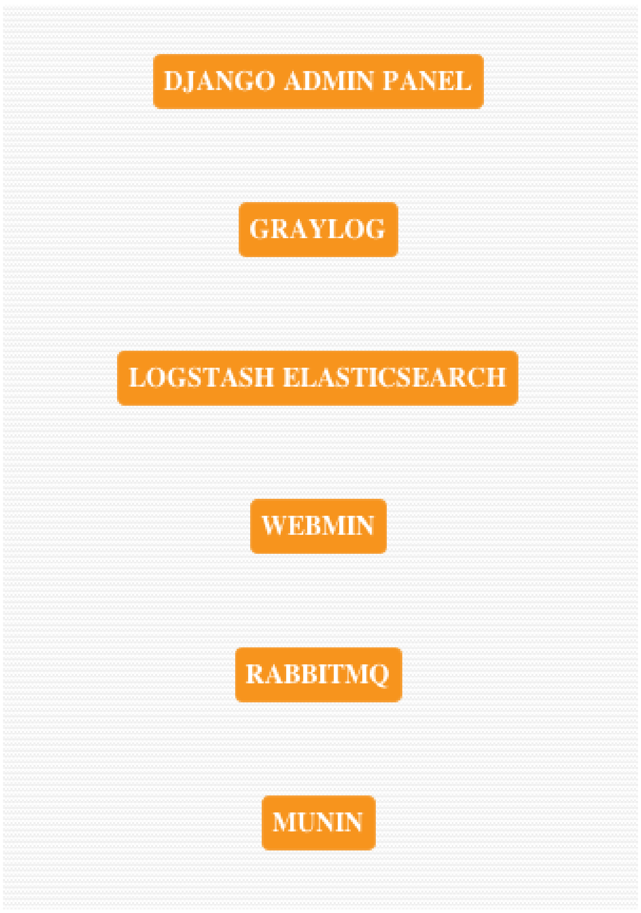


Figure 4.14: A screen shot of the admin portal.

Django Admin Panel

Administrators on the platform are able to login to the Django administration panel. Here administrators can add and remove users, video meta data, videos and make videos available and unavailable on the platform.

Graylog2

The Graylog2 dashboard is used to monitor logs, streams and alerts. This page is password protected and allows for multiple users to be created for it. Graylog2 provides real-time analytics and server health, through visual indicators such as the current message throughout and queue lengths.

Logstash

Although Logstash is used for filtering and mutation, it too has a search engine that allows searches to be performed on the logs. Logstash does not store logs, it is only an intermediary. Graylog2 is responsible for storing the logs in MongoDB database. Therefore searches can only be done on Logstash since the last downtime.

Webmin

Webmin is a web interface for system administration on Unix based operating systems. Webmin provides an alternative to the command line interface of Ubuntu Server.

RabbitMQ

RabbitMQ provides a web-based user interface to manage connections, queues, exchanges and monitor throughput on the system. The web-interface also requires user authentication in order provide access to only the platform administrators.

4.10 Summary

This chapter described the implementation of the design decisions what were made in Chapter 3. The server implementation that was developed met the requirements that were set out by the design. Importantly, even though implementation is developed on one server, the design and modular construction of the server allows it to be segmented onto separate servers if the need is requirement in a production environment. It should be evident that the server can be further extended if more features are required for the platform.

Chapter 5

Client Application

The client application is used by the end user to gain access to the platform. Through the application, the user is able log on to the platform, search for available media on the platform, and download and watch video content.

The client application is responsible for decrypting the video data in real-time and presenting it to the user without compromising security. At the same time the client application has background tasks, such as seeding downloaded video content, tracking video usage and logging application behaviour.

This chapter details the implementation of the client application and includes the challenges that were faced during the implementation of the video player. An overview of the high level components that make up the client application are illustrated in Figure 5.1.

5.1 Graphical User Interface

Before looking at the components in the background, we inspect the user interface. This is the primary point of interaction that the user has with the application. The GUI is written in PyQt using the Qt framework. The QtGui module is extensively used for the GUI implementation and the Phonon module is used for the video playback.

5.1.1 Login Screen

When the application starts, the user is presented with a login screen (Figure 5.2). Here the user is required to enter their credentials correctly in order

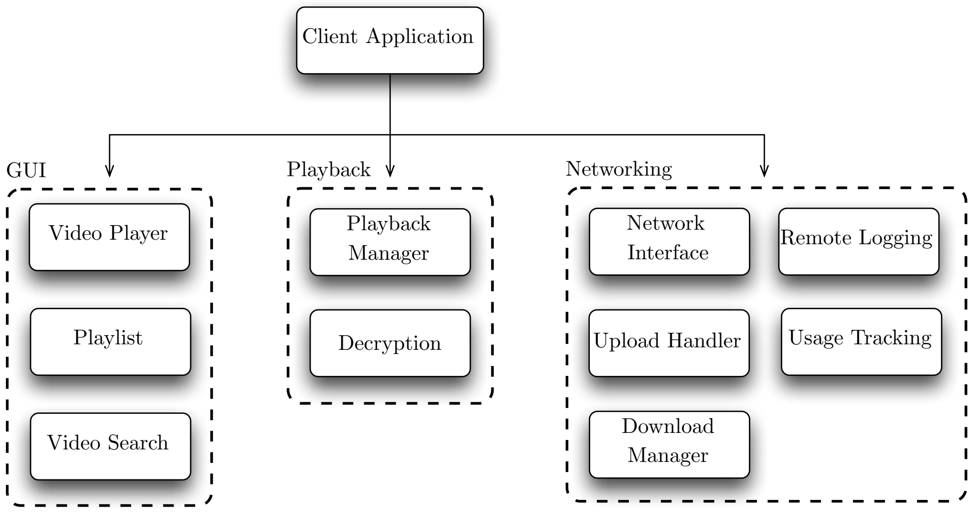


Figure 5.1: Component overview of the client application.

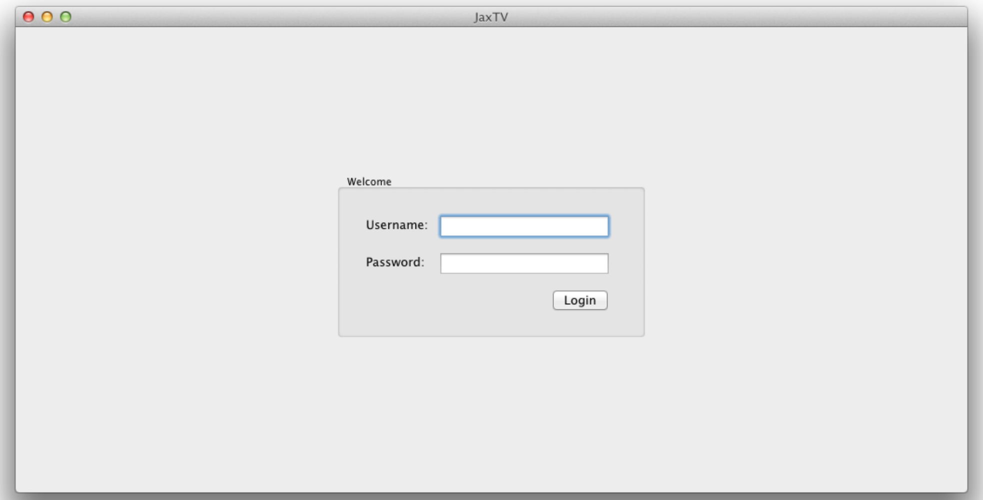
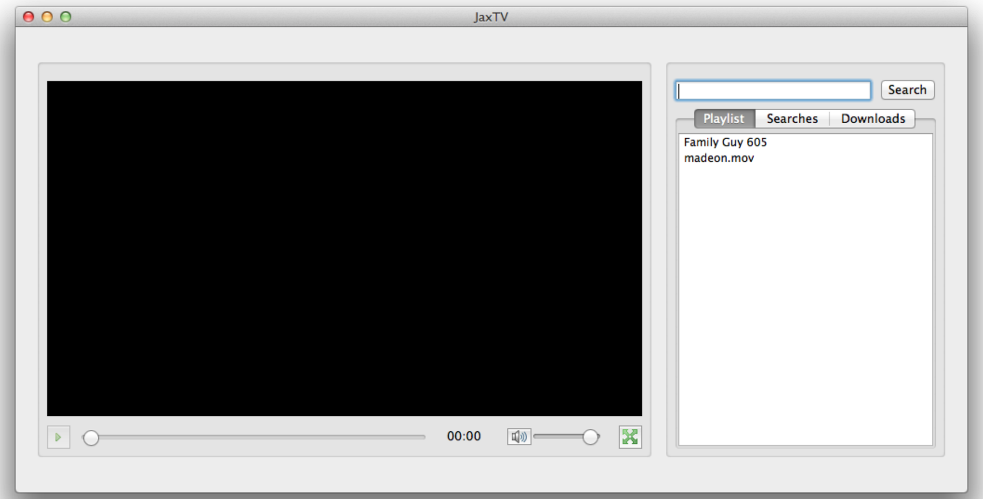


Figure 5.2: A screenshot of the application's login screen running on Mac OS X 10.7.4.

to progress to the video player screen.

5.1.2 Video Player Screen

Once the user has successfully logged in, he or she is presented with the video player screen as illustrated in Figure 5.3. The player screen is essentially the main screen of the application. It contains the video player (most of the screen space) and the tabbed widgets on the right hand side. By default the playlist tab is displayed, but the search results tab displays when a search is performed and the downloads tab is displayed when a video is added to the download queue.



Family Guy 605
madeon.mov

Figure 5.3: A screenshot of the application's main screen running on Mac OS X 10.7.4.

Video Player

The video player includes the basic functionality, such as play, pause, volume control, mute and fullscreen, as well as displaying time and position of playback. Unfortunately, the Phonon VideoWidget by default does not enter or exit fullscreen on the mouse double click events. This has become a standard feature amongst most video players and needed to be implemented in order to provide the same feature rich experience as seen in current video players. The Phonon VideoWidget was extended with the CustomVideoWidget class to provide this feature. The class also ensures that the video widget displays with an aspect ratio of 16:9. The QtGUI and Phonon module objects are illustrated in Figure 5.4.

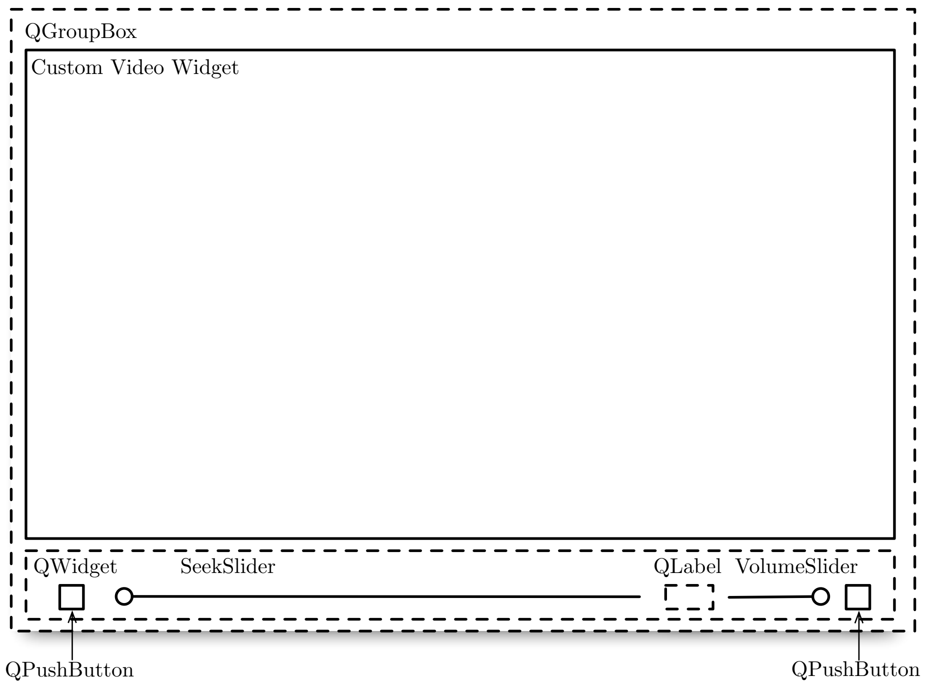


Figure 5.4: Qt component sketch of the video player.

Playlist

The playlist displays media that is available for playback, as well as the meta-information about the media. The meta-information is displayed as a tooltip by hovering the mouse over the item. Playback of any item in the list is achieved by double clicking on the entry in the playlist.

The PlaylistUpdater class is responsible for populating the playlist by checking for files in the media folder. This is done by calculating the MD5 hash sum of all the files in the folder and these MD5 hash sums are queried against the server for video ID validity. Files that have been downloaded completely will have a valid video ID (*info_hash*). A file that is not complete will not provide a valid video ID, and neither would any files that are not part of the platform. This means that the application will not play any content other than the content provided on the platform. This discourages the use of pirated content by restricting playback to legal content from the platform. Rejecting invalid videos from the application also provides video integrity within the platform and prevents users replacing video files with inappropriate or explicit material. Kohl *et al.* [84] list integrity as a tool to safeguard users with digital content libraries.

The application must be responsive to new content added, be that through downloads or from file sharing from another user. In order to achieve this, the application regularly updates the playlist, removing entries that are no longer present and adding entries that have recently been acquired. The main window class is responsible for requesting the PlaylistUpdater to perform updates on a regular interval. This interval is defined in the application's configuration file as PLAYLIST_UPDATE_PERIOD. The PlaylistUpdater class runs in its own thread and performs the first scan as soon as the application starts. This is to improve responsiveness of application, both through running in a separate thread and populating the list before the user is logged in.

The problem with performing video ID lookups on the server before the user is authenticated, is that the application's request will be denied by the server. To overcome this, the application is given a user account on the platform. Every request made to the server that queries the validity of a video ID uses this user to authenticate. The username and password is stored in the constants.py file along with the other configuration information. All the client applications share this user name.

The second problem with calculating the MD5 hash sum of a file is that the operation is hard disk intensive. A typical SD movie video file exceeds 700MB, and having to scan these multiple times is inefficient and creates a bottleneck during playback (which is also hard disk intensive). To overcome this, two solutions are implemented:

During playback the playlist cannot be updated. The main window class checks to see if video playback is occurring; if so, the update is skipped. In order to solve rescanning files, a cache file is created. If a valid video file is discovered, it is entered into the cache using the file name, along with the video ID and the last modified time of the file. The cache file is written to disk and stored as JSON format. A typical entry is given in Figure 5.5. If the file is scanned again it is first determined whether the file is in the cache. If so, the modification time of the file is checked and matched against the time stored in the cache. This is to ensure that although the file name has remained the same, the file has not been modified, or that another file has replaced it with the same name. The PlaylistUpdater class also checks for files that are no longer present or that have been modified, and removes them from the cache. Once changes are made to the cache it is written to the hard disk. The benefit of this is that, when the application is opened again, files that were previously hashed do not need to be hashed again.

```
{
    "sonyhddemo.mov": {
        "info_hash": "ab8564d200d870993d217d27be494087",
        "mtime": 1342100320.0
    }
}
```

Figure 5.5: An entry in the JSON formatted video cache file.

Search Functionality

Video search is provided through the search field above the playlist. Here the user is able to search for any video available on the platform. The user enters the search term into the search field and clicks on "search". The tabbed widget automatically switches over to the search tab and displays results received from the server. Video meta-information is displayed in the same way as the playlist through the use of tooltips. It is important to note that only videos marked as available in the Django admin panel, are returned to the client.

Download Manager

The client application contains a download manager that queues download requests made by the user. The user is able to use the search functionality of the client application to query the server and start a download by double-clicking on the entry in the search list. This entry is placed on the download manager's queue and the tabbed widget automatically switches over to the downloads tab. Here the queue is displayed with the order in which each video will be downloaded. The download manager sequentially downloads each item from the queue, and removes the item from the queue when the download has finished.

5.2 Playback

Video playback is by far the most complex procedure on the platform, unlike the encryption phase which does not need to be completed by a hard deadline. Decryption, on the other hand needs to occur quickly enough so that the video player does not play through unencrypted data faster than the rate at which data can be decrypted. Along with this, it needs to be ensured that the user cannot access the decrypted video data.

The initial implementation was met with issues and required a reimplementation. The rest of this section describes the issues and how they were overcome.

5.2.1 Implementation Issues

Using Qt's Phonon module and the MediaSource class, it is possible to use any class that inherits the QIODevice interface class as a media source. These sources include local and remote file access, network streams and memory storage. The choice was made to use the QBuffer class to store the decrypted video data for playback. The QBuffer provides a QIODevice interface to the QByteArray in memory. Using this class, the decrypted data is never written to the hard drive. This makes the decrypted video data out of reach of the end user.

The problem with using memory storage for decrypted video data, is that it is not feasible to store the entire video in memory (in particular HD content). In order to resolve this, the video data is decrypted block by block during video playback and written to the QByteArray, storing only a few blocks in memory. Unfortunately, once a QBuffer is set as a video source, any data written to the QBuffer (or directly to the QByteArray) after that point is not played. Let us for example say you load 5% of the decrypted video data in the QBuffer, you set this QBuffer as the video source, start playback, and then load more decrypted video into the QBuffer. In this case, the video player will stop playback when the 5% mark is reached. This problem is not documented by Qt and after investigation on various forums online it was established that the problem is attributed to the various backends Qt uses for playback on the different operating systems.

To circumvent this issue, the blocks in the original file were converted into individual video files of approximately the same size. These individual video files, each with their own header information, were used together to form a playlist. These files were each decrypted separately and loaded into individual QBuffers, and the QBuffers added to a playlist with a transition time of zero. This was in order to implement seamless playback between the QBuffers.

Although this implementation looked promising, there were noticeable artifacts on the audio when transitioning between QBuffers. These artifacts presented themselves as dropouts in the audio, even though the transition time was set to zero. This was especially noticeable when the video contained music and these audio artifacts are not acceptable for a video player.

5.2.2 Compromise

With the above mentioned implementations unable to offer a usable video service, another implementation was required. Although, maximum security is always the aim, by sacrificing a tolerable amount of security, we are able to achieve a secure prototype system for the specified criteria.

Instead of decrypting the video file in memory, a duplicate temporary file is created on the hard drive. This temporary file allows the encrypted video data to be decrypted in place within the file in small amounts to allow playback. Once decrypted data is used for playback it is erased with random data and at the end of playback the temporary file is removed from the hard drive. For the duration of playback, the file size remains the same and makes it difficult for the attacker to monitor changes to the file.

In the next two subsections, the playback and decryption procedures are explained in detail. Two classes are created for handling playback. The first is the PlaybackManager class which the main GUI thread uses to control the video playback. The second is the ThreadedDecryption class which is responsible for the decryption of encrypted video data and the erasing of decrypted video data.

5.2.3 Playback Manager

The playback manager provides controls to the main GUI thread to start, pause and resume video playback. When a video entry in the playlist is double clicked, a playback manager is created for that video. The playback manager is responsible for triggering the decryption class.

The playback manager uses the timer synchronisation value (*timer_sync* in the video metadata) to initialise a timer. The timer synchronisation is the time required to play out a block of video data. The Phonon module in Qt has events for video playback, unfortunately these events (start, stop, nearing the end of the video) are associated with the entire video file. Because the the playback manager is working with blocks within the video file, it requires events to notify the decryption class to decrypt the next block. The timer synchronisation aids the playback manager with this information. When the timer times out, the playback manager signals for the next block in the video to be decrypted.

Unfortunately, the QTimer in Qt does not support pausing. Therefore, starting the timer resets the timer to the initialised value. This is a problem for video playback where the user often needs to pause playback. To allow for this, the playback manager extends the functionality of the QTimer class to allow for pausing and resuming.

5.2.4 Decryption

The decryption of the video file proceeds in iterations through the blocks in the encrypted video file. When the procedure starts, a copy of the encrypted file is created in the temporary storage directory. All operations are then carried out on the duplicate file by seeking to the position of the file where the operation must occur. Initially the first two blocks of the file are decrypted. This is the preloading stage and the amount of blocks that are preloaded is specified by the *preloadIndex*. The preloading creates a buffer between the loading block and the playback block in case that the timer is delayed. For each block the key is fetched from the server (using the NetworkInterface class explained in Section 5.3) and then using the *load* method the video data is decrypted. The decryption is the same procedure as that of the encryption, the only difference is the pad that is generated from the key, is XORed with the encrypted data.

Once the preloading stage is completed the, ThreadedDecryption class notifies the PlaybackManager class that preloading has finished and playback can begin. The *playbackIndex* maintains the index of the block that is currently playing out and the *loadIndex* maintains the index of the block that is being loaded. The key is fetched a block ahead of the *loadIndex* and maintained with the *keyIndex*. The entire process is visually described in Figure 5.6

Because the temporary file is potentially accessible by the end user, the ThreadedDecryption class erases blocks that have already been played. Random data is written into blocks that have already been played and the erasing position is the maintained by the *eraseIndex*. Random data is written into the file instead of zero blocks because it does not provide a trail for attackers to monitor.

The ThreadedDecryption class runs in its own thread and is managed by the PlaybackManager class.

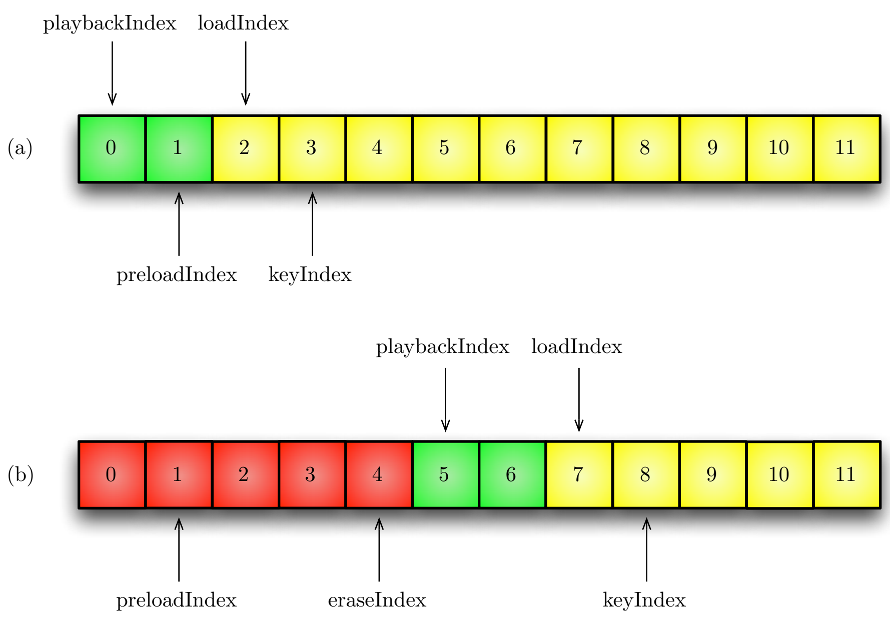


Figure 5.6: A visual representation of indices used to manage playback.

5.2.5 Future Work

Although the first implementation offered higher security, it lacked usability and another approach was required. This approach does strike a suitable compromise but lowers the security of the platform. However, this is suitable for a prototype application for this platform. Despite the implementation issues, the design of the system remains valid and requires future work to be done on implementing a client application that decrypts and stores video data in memory.

This will most likely require the client applications to be written natively for each operating system from the ground up. This is simply not feasible within the time frame of this project and must be left for future work.

5.3 Networking

The client application consists of various networking modules to complete the functions that the application offers. Each module is explained in detail below.

5.3.1 Network Interface

The NetworkInterface class is responsible for all communication to the REST server. The class maintains the base URL for requests, that is the `https://<host_name>/` portion of the URL. The host name of the REST server is located in the constants.py file as REST_HOST. The class also contains a cookie processor, as well as the user's credentials. The server uses cookies to maintain a session with the client application and the cookie processor is used to maintain the cookie. The session cookies allow the user to login once and use the session cookies to prove identity until the cookie expires.

The NetworkInterface includes the custom application HTTP user agent string in all the requests and automatically performs a login if the session has expired with the server, whilst the user is still logged into the application. All the requests that are made to the server are HTTPS GET requests with the information encoded in the URL. The NetworkInterface class has the following methods:

authenticate

The method sets the user name and password for the communication.

login

This method performs the actual login of the user on the server. The method performs the login and handles the Cross-site request forgery prevention that Django enforces. The HTTPS request is made to the login template.

listVideos

The listVideos method lists all the videos on the server by accessing the videos_index view. It is only used for development.

queryInfoHash

This method collects the information about the specified video. The HTTPS request is made to the info_hash_query view and the video ID is encoded in the URL.

listKeys

The listKeys method displays all the keys for the specified video and is only used for development purposes.

queryKey

This method is used by the ThreadedDecryption class to request the key from the server. The video ID and the index of the key need to be specified for the request to be made.

videoSearch

This method is used by the search list to look up videos on the video catalogue for the specified search term. The request is made to the video_search view on the server to perform a look up using the Sphinx search engine.

videoValidTest

The videoValidTest method is used by the PlaylistUpdater class to check if the videos in the media directory are valid. If the MD5 hash sum is valid, the server will respond with true.

fetchMetaData

This method collects the meta data for the video to be used with either the playlist or the search list.

5.3.2 Upload Handler

The upload handler is responsible for seeding available video content onto the network. The upload handler is a component of the original Jax.TV implementation and designed to search for .jaxtv metadata files. The presence of the meta file indicates to the upload handler that the the video file is available for uploading. Once the uploader handler is aware of all the available files to upload, it announces the client to the tracker.

The playlist updater creates metadata files in the metadata directory when new entries are created in the playlist. After the playlist is populated for the first time, the uploader handler thread is started. Every time the playlist is updated, the uploader handler rescans the metadata directory and announces the client to the tracker again. This way the tracker remains up to date with the clients on the platform.

5.3.3 Remote Logging

The remote logging module is responsible for transmitting the log messages to Graylog2 via the RabbitMQ server. The module consists of two classes, the RabbitMQProducer and the RabbitMQLogHandler.

The RabbitMQProducer class runs in its own thread and maintains a connection with the RabbitMQ server. The class is a producer for the RabbitMQ server, as illustrated in Figure 4.12. The RabbitMQProducer class uses the python Queue class and sends any message placed on the queue to the RabbitMQServer. The connection to RabbitMQ is closed when the thread is deleted.

The RabbitMQLogHandler class allows the components within the application to send log messages to the logging server. The RabbitMQLogHandler class is a logging handler for the Python logging library and extends the Python logging Handler class. Each log is a JSON object with the log message, the local IP address and the host name of the client. The RabbitMQLogHandler places this JSON object on the queue as a message for the RabbitMQProducer.

A logger instance *general_logger* is created in the main GUI thread, along with the logging level set in constants.py. The RabbitMQLogHandler is then added to the logger instance as a handler. This instance is used in all the modules to send log messages to the Graylog2 server.

5.3.4 Tracking

Tracking allows system administrators and content providers to monitor the platform through the collection of usage statistics. The tracking module consists of two classes, the TrackingBroadcaster and the TrackingHandler.

The TrackingBroadcaster class is used to transmit tracking instructions to the REST interface. Each tracking instruction is achieved by creating a HTTPS POST request to the REST interface. The TrackingBroadcaster implements a queue system and removes each entry off the queue, determines the type of tracking instruction and creates the POST request.

Each type of tracking instruction has a unique URL. For example, to track a user login, a POST request is made to https://<host_name>/tracking/login with POST data. It would be possible to use one URL and include the specific tracking instruction within the POST data and allow the server to decide. This route will not scale well as network traffic increases. By distributing the decision making (albeit small) to the peers, the load of the server is reduced and maintains the peer-to-peer architecture of the platform.

The TrackingHandler provides a logging handler for the Python logging system by extending Python logging Handler class. The TrackingHandler can be used by any module to provide tracking and constructs a dictionary entry which consists of the tracking instruction, IP address, the user that is logged in, as well as the date and time. The TrackingHandler places this on the queue in the TrackingBroadcaster.

5.4 Summary

This chapter described the client implementation with regards to the design of the platform. Although implementation issues were encountered because of a problem with the Qt framework, a workaround was created that still meets the design brief. This workaround is acceptable for a prototype application, however for a production environment it would be best for the client applications to be developed natively for their respective operating systems.

Importantly, the prototype application served as a proof of concept and confirms that the design of the platform is solid with respect to encryption and video playback.

Chapter 6

Platform Evaluation

This chapter documents the evaluation of the platform, beginning with the individual tests before proceeding to the integration tests and finally the platform test. For each test the objective, protocol, results and discussion is given. The chapter also details the set up used to conduct the tests.

6.1 Test SetUp

A standard configuration was used for the majority of the tests performed during the platform evaluation. The standard configuration consists of the various hardware, operating systems, applications and software libraries. Each of these are detailed below:

Primary Server

The primary server is used to host the REST interface (Apache, Django, MySQL), central logging system (RabbitMQ, Logstash, Graylog2), the video encryption module, the Jax.TV tracker and the usage tracking (MongoDB along with the REST interface). The server consists of a virtual machine running on VMware vSphere, using Ubuntu Linux Server as a guest operating system. The virtual machine is equipped with a single 2,3 GHz vCPU and 1024 MB of RAM. A comprehensive list details the server in Section A.1.

This server is hosted on the same LAN as the client devices, and in order to simulate low bandwidth Internet conditions, Apache is configured to limit each connecting client to 512 Kbps. All traffic on the LAN is unaffected except the traffic to the REST interface. For more detail on the Apache configuration refer to Appendix B.1.

Monitoring Server

The monitoring server is used to monitor the primary server and a client device. Additionally, the server is also acting as a super-peer on the network. The server consists of a virtual machine running on the same VMware vSphere host as the primary server, with Ubuntu Linux as the guest operating system. The virtual machine is equipped with a single core 2,3 GHz vCPU and 4096 MB of RAM. Again, a complete list of details is given in Section A.2

Macintosh Client

A 13 inch 2010 Apple Macbook Pro was used as development device, as well as the device used for the majority of the tests listed below. The Macbook Pro includes a 2,4 GHz Intel Core 2 Duo CPU, with 8 GB of DDR3 memory, a NVIDIA GeForce 320M graphics card with 256 MB of memory and 7200 RPM hard disk drive. The operating system used is Mac OS X Lion 10.7.3.

Linux Client

The Linux client environment is running Ubuntu Linux 12.04 in a virtualised environment with 1024 MB of RAM.

Windows Client

The Windows client environment is running Microsoft Windows 7 64-bit, and is also in a virtualised environment with 1024 MB of RAM.

6.2 Component Testing

This section details the tests of the individual components that make up the platform.

6.2.1 Erasing Decrypted Data

Objective

The objective of this test is to ensure that the video data that is decrypted for playback (in the temporary file) is properly erased during playback. Although the implementation uses programming libraries to achieve this, it still needs to be validated.

In order to validate that the decrypted data is properly erased, a comparison between the erased file and original unencrypted file is required. Unfortunately, available comparison tools are mostly aimed at short-length text files. In order to compare these two files, we created a test to measure if sections of the file contain unencrypted video data.

As stated in Section 5.2.4, the decrypted video data in a block is XORed with a pseudorandom binary sequence of the same length and produces a new array (erasure sequence) which is written to the same position in the temporary file. We know that XORing a statistically independent pseudorandom binary sequence with any other binary sequence will produce a new statistically independent pseudorandom binary sequence of 1's and 0's. In this sequence, the probability of a one occurring is the same as the probability of a zero occurring: $P(0) = 0.5$ and $P(1) = 0.5$. Furthermore, all symbols are statistically independent of each other so: $P(0)^N = 0.5^N$ and $P(1)^N = 0.5^N$ where N is the number of sequential symbols.

With the knowledge of what to expect in the erased file, we XOR the erased file with the original unencrypted video file. This procedure is illustrated in Figure 6.1. If the erasure process should fail, the imperfectly erased file will contain blocks of content that are statistically dependent on the original file (typically containing blocks filled with original video data). When this file is XORed with the original unencrypted video file, corresponding blocks (where erasure failed) will contain sequences of 0's.

As described in the client implementation Section 5.2.4, when the video playback stops the temporary file is removed from the hard drive and the for that reason the last block is not erased. For this test, the client application is modified not to remove the file at the end of playback. Because the last block is not erased, the last block is not used in the test.

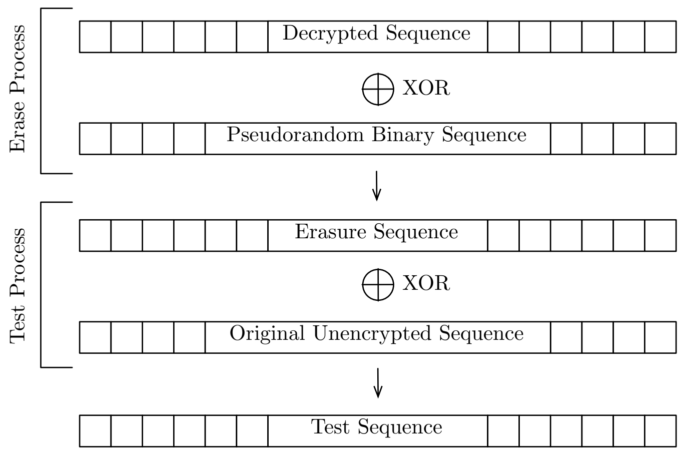


Figure 6.1: A graphical representation of the erase test procedure.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Modify the client application not to remove the file after playback.
3. Play a video using the client application.
4. Remove the last block from both the original unencrypted file and the temporary file, then perform a bitwise XOR of the two files.
5. Tally the frequency of the different zero length sequences produced from the XOR in the previous step.

Results

Figure 6.2 plots the probability of a sequence of zeros occurring versus the length of that sequence.

Discussion

The file used for the test is 49,6 MB H.264 encoded nine minute and fifteen seconds video. With a bit rate of 714 Kbps, a sequence of zeros that exceeds 714000 in length with result in at least one second of video data that is not erased.

From the Figure 6.2 it is clear that the erasing procedure is successful as the longest sequence of zero bits that occurred is 25 sequential bits. This is

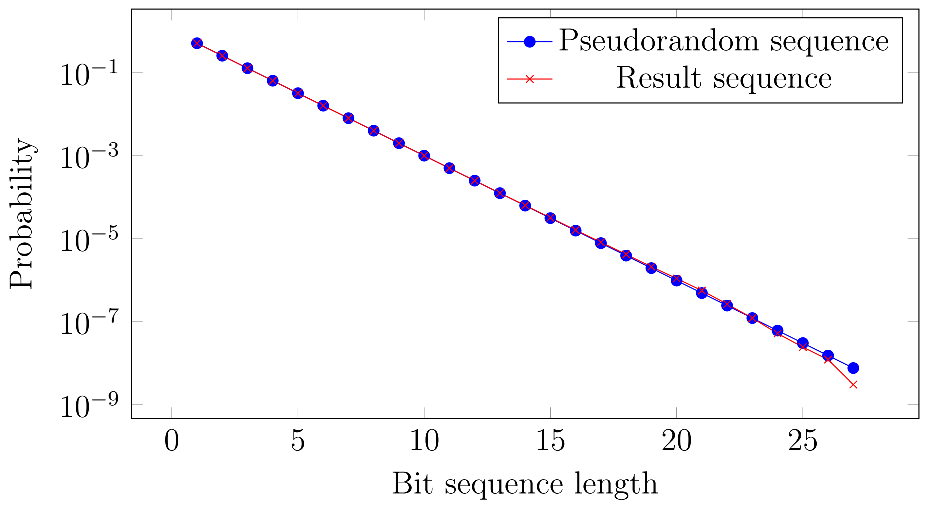


Figure 6.2: Probability of zero sequences vs the bit sequence length.

much lower than the sequence length of 714000 required to view one second of video. Additionally, the probability determined from the erased file closely matches probability of a pseudorandom sequence, as we expected with the erasing procedure.

6.2.2 Video Search

Objective

The objective of this test is to evaluate the effectiveness of the full text search server. Both the accuracy of the results, as well as the speed with which the results are returned, will be measured. The test will require a large collection of video entries to be indexed by Sphinx for meaningful results. In order to achieve this, the database was populated with a maximum of 1 million entries randomly selected from IMDB's database. IMDB [85] is an online database containing information about movies, TV series and actors, that is gathered from the studios and also submitted by users visiting the website.

Protocol

1. Perform the standard set up described in Section 6.1.

2. Define a list of all the videos indexed by Sphinx.

3. Iterate through the list using each entry as a query for Sphinx and collect the results for each query, as well the time taken to receive the results.

4. Check to see if each query was listed in the returned results and compare the latency.

Results

The results from the test are tabulated in Table 6.1. Each row represents the number of entries within the indexed database, the average time it took to perform a query with Sphinx, the average number of results each query returned and the number of errors or invalid results that are returned.

Table 6.1: Comparison of Sphinx search performance vs database size.

Database size	Average time	Average Results	Errors
1,000	38.81ms	1.549	0
10,000	42.66ms	7.915	0
100,000	50.6ms	24.478	0
1,000,000	79.06ms	90.613	1

Discussion

From the results it is clear that Sphinx performs well in a development environment where the resources are shared amongst other services such as the centralised logging. Sphinx almost always returns meaningful results, with the exception of one query. During the test with a database containing 1 million entries one error occurred. The queried term '`<-->`', which is possibly a mistake in IMDB's collection, caused Sphinx to generate a server side error. This happened despite escaping all queries using the Sphinx API. Although an error did occur, it is one query out of a million and does contain unusual characters for a video title. The average time to perform searches scales impressively as the database size increases.

6.2.3 Cross-Platform Support

Objective

The object of this experiment is to test cross-platform capabilities of the client application. Video playback within the client application relies on the Phonon module within the Qt framework. Phonon itself relies on various backends for the different operating systems. For this experiment the client application will be tested on Microsoft Windows 7, Ubuntu Linux 12.04 and Mac OS X 10.7.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Determine the default Phonon backend on each operating system.
3. Using the client application, test the video playback on each operating system.

Results

Table 6.2 lists the backends that are used by default for the different operating systems, as well as playback support using that backend.

Table 6.2: Phonon backend support on the different operating systems.

Operating System	Backend	Playback
Windows	DirectShow	✗
Ubuntu	GStreamer	✓
OS X	QuickTime	✓

Discussion

The client application is only able to provide video playback on Mac OS X and Ubuntu Linux. Using Windows Media Player it is possible to playback video using the H.264 codec that is used for the test. Unfortunately, because of problems with the DirectShow Phonon backend on Windows, it is not possible achieve video playback through the client application. It is possible to achieve playback on Windows by installing K-Lite Codec Pack [86], a third party codec utility for Windows, however the application aims to achieve native codec support without the need for additional utilities. In order to resolve this issue, native applications must be created for each operating system on which the client application needs to run.

6.2.4 Codec Support

Objective

One of the specifications of the platform is to be video codec independent. The platform design allows for this; however, we need to test whether the prototype application meets this specification. Although third party utilities allow for additional codecs to be supported, this test focuses on the codecs that work with the Qt framework and Mac OS X natively. This test will be performed on the Macintosh client, as the majority of the development was performed on this client.

Protocol

A variety of video codecs were required to test the application. Fortunately, MPlayer [87], which is an open source video player, makes a collection of videos of a variety of codecs available on their website. These videos contain codecs that are common and videos that have obscure codecs or are purposefully damaged (such as missing frames). Ninety videos were selected from the website and analysed using VideoSpec [88] for compatibility with Mac OS X. The remaining steps are detailed below:

1. Perform the standard set up described in Section 6.1.
2. Analyse each video using VideoSpec to determine the bitrate, codec and container.
3. Using standard file playback, attempt to play the video using the Qt video widget.
4. Encrypt each video that successfully plays on the Qt video widget.
5. Test the compatibility of each video on the platform application.

Results

Table 6.3 tabulates the results for this test but only lists the videos that were able to play on the Qt video widget. For each video, the number, container and codec is given. Along with this the compatibility with the Qt video widget, the platform encryption method and the platform application is given.

Only video number 76 failed to be encrypted. The reason for this is that the encryption method uses the Hachoir Python library to determine the duration of the video, and was unable to determine the duration of the video. Without the video duration it is not possible for the encryption module to calculate the timer synchronisation.

Table 6.3: Video player codec support.

No.	Container	Codec	Qt	Encryption	Jax.TV
13	MP4	H.264/MPEG-4 AVC	✓	✓	✗
16	MOV	DV (PAL)	✓	✓	✗
18	MOV	DVCPRO (PAL)	✓	✓	✗
19	MOV	DVCPRO 50 (PAL)	✓	✓	✗
22	MOV	Photo - JPEG	✓	✓	✗
27	MOV	Apple ProRes 422(HD)	✓	✓	✗
76	MOV	Component Video - YUV422	✓	✗	n/a
88	MP4	MPEG-4 Video	✓	✓	✓
97	MOV	H.264/MPEG-4 AVC	✓	✓	✓

Discussion

From Table 6.3 we can see that using standard file based playback with the Qt video widget is supported (on Mac OS X) for all the videos that are listed. However, once the videos are encrypted and played back using the platform application (that makes use of the Qt video widget), only two codecs are supported.

A possible explanation for this is that the Qt video widget might require information from the entire file or from more than the two blocks that are decrypted any any one given moment. This reiterates the need for a native application.

6.2.5 Playlist Caching

Objective

Playlist caching is put in place to improve the performance of populating the playlist because of the disk intensive MD5 hash sums that are calculated for each video. The objective of this test is to measure the performance increase that the cache provides when populating the playlist.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Launch the client application on the Macintosh client, populate the playlist with no previous cache and measure the time.
3. Launch the client application and populate the playlist again, this time using the cache created in the previous iteration and measure the time.
4. This process is repeated for playlists with 10, 20, 30, 40 and 50 videos.
5. Compare the difference between a 'cold start' and cached start.

Results

The results of the test are given in Table 6.4 where each row in the table represents an iteration with a particular playlist size. The size of the playlist is given along with the cold start duration and cached start duration in minutes and seconds. The smallest test of 10 videos contains 16.2 GB of video, while the largest test used a video collection of 80.98 GB.

Table 6.4: Comparison between 'cold start' and cached start playlist population.

Entries	Size	Cold start	Cached start
10	16.2 GB	02:51	00:02
20	32.39 GB	05:17	00:12
30	48.59 GB	08:43	00:24
40	64.78 GB	11:51	00:28
50	80.98 GB	14:47	00:30

Discussion

It is evident from the results that the playlist caching mechanism makes an enormous difference in the time it takes to populate the playlist. As we expected, the MD5 hash sum is hard drive IO intensive, as the Macintosh client reported an average data read of 90 Mbps.

The caching helps keep time to populate a playlist well under one minute for a decent amount of video data. Since some video collections can be far larger than 500 GB, this caching will scale well as the collection size increases.

6.2.6 Decryption Performance

Objective

Decryption performance is an important aspect of video playback as it enforces a real-time constraint on the procedure. Even though both encryption and decryption procedures are similar in relying on the XOR operation, the encryption process has no real-time constraint.

Two tests are carried out with the objective of evaluating the performance of the decryption procedure. The first test evaluates the performance against different bitrate videos. Here the encrypted video data needs to be decrypted in a time frame that is shorter than the time needed to watch the video. The second test compares the overall load between playing an encrypted video and the original unencrypted video through the application.

The first test determines time required to encrypt a video file. This is measured by the real time factor. The real time factor is given as

$$\mathrm{RTF} = \frac{P}{I} \tag{6.2.1}$$

where P is the time required to process the encryption and I is the duration of the video clip. In an ideal situation a RTF greater than 1 will result in the decryption lagging behind the playback of the video. Because resources have to be shared on the computer, it would be safe to assume that a RTF smaller than 0.8 will not result in any video lagging. This provides a buffer of 20% for other processes to share the system resources. The resolutions and bit rates used for the test are listed in the Table 6.5.

The second test is conducted by modifying the video player to play unencrypted video files and compare the system resource usage with that of encrypted video playback. The video used for this test is an H.264/MPEG-4 AVC encoded video with a bitrate of 10.3 Mbps and a resolution of 1920x1080 (HD). The system resources are monitored with a cloud based monitoring service called Datadog [89] that is installed on the Macintosh client.

Protocol

Test 1: Decryption speed:

1. Create a collection of encrypted files at different bitrates and store the keys on testing device.
2. Record the time required to decrypt each encrypted video.
3. Compare the RTF of each video.

Test 2: System resource usage:

1. Perform the standard set up described in Section 6.1.
2. Monitor the system resource usage using Datadag.
3. Modify the application to play regular video files.
4. Compare the resource usage.

Results

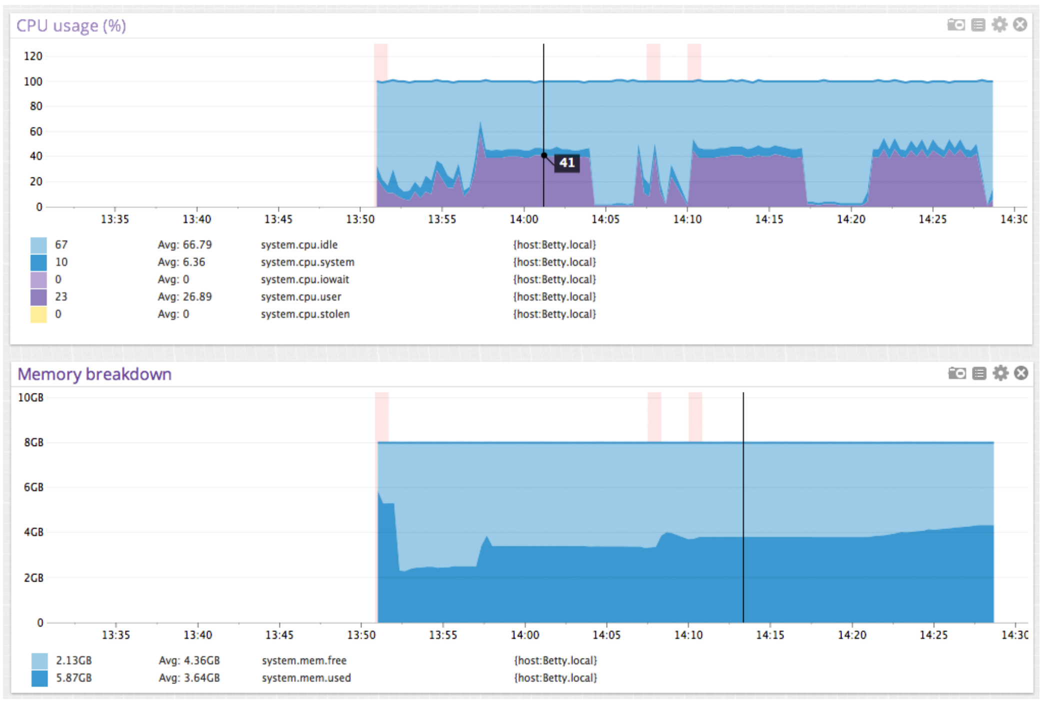
Table 6.5 lists the real-time factor for decrypting videos of different bitrates. Figure 6.3 shows the CPU usage and memory usage of the Macintosh client for playing unencrypted video content and playing video content that is decrypted in real-time. The test for playing unencrypted video starts at 14:10 in the figure, while the test for playing encrypted video content begins just after 14:20.

Discussion

For the first test it is important to bear in mind when looking at the results that these values only factor in the actual decryption procedure of reading in the data from the file, generating the sequence for the one time pad, performing

Table 6.5: Decryption real-time factor performance versus bitrate.

Bitrate	RTF
997 Mbps	9.026
96.1 Mbps	0.960
70.8 Mbps	0.707
46.7 Mbps	0.469
22.7 Mbps	0.227
18 Mbps	0.180
13.3 Mbps	0.134
8850 Kbps	0.098
4530 Kbps	0.049
1899 Kbps	0.020
893 Kbps	0.008
386 Kbps	0.006



🟦	67	Avg: 66.79	system.cpu.idle	{host:Betty.local}
🟦	10	Avg: 6.36	system.cpu.system	{host:Betty.local}
🟪	0	Avg: 0	system.cpu.iowait	{host:Betty.local}
🟪	23	Avg: 26.89	system.cpu.user	{host:Betty.local}
🟨	0	Avg: 0	system.cpu.stolen	{host:Betty.local}

🟦	2.13GB	Avg: 4.36GB	system.mem.free	{host:Betty.local}
🟦	5.87GB	Avg: 3.64GB	system.mem.used	{host:Betty.local}

Figure 6.3: System resource usage during video playback.

the XOR and saving the data to file again. However, the results do provide an indication of the maximum bitrate that is possible to decrypt in real-time.

In test two, we see that the application, when modified to play unencrypted video, has an average CPU usage of 40%. When encrypted video content is played, the graph remains around 40% but contains small spikes in CPU usage. This is attributed to the decryption process that is decrypting new blocks and erasing played blocks of video data in set intervals.

6.2.7 Network Performance

Objective

This test performs an evaluation of the network traffic transmitted between the client and server. The objective of the test is to measure the network traffic (excluding the video data) and to ensure that communication between the client and the REST interface is secured. Network packet sniffing software is used to perform this test and if the network traffic is secured, then the packet sniffing software will reveal no information about the network traffic other than the source and destination.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Set up the Wireshark [90] network protocol analyser on the client to monitor traffic between the client and the server.
3. Launch the client application and play a video from start to finish.
4. Analyse the results from Wireshark.

Results

Table 6.6 provides a summary of the network traffic recorded on Wireshark. All TCP traffic between the client and the server consisted of three protocols: HTTPS (port 443), the Jax.TV transfer protocol (port 3001) and the AMQP protocol for RabbitMQ (port 5672). It is important to note that the logging traffic over the AMQP protocol is not secured. This is an acceptable design decision, as no sensitive information such as user passwords or video keys are present in the log data.

By analysing individual HTTPS packets, Wireshark revealed that transport layer security (TLS) version 1 is used for the encryption and the entire payload is encrypted. Both traffic sent to and received from the REST interface is encrypted.

Table 6.6: Network traffic summary.

Packets	1167
Bytes	211961
Average traffic	0.007 Mbps
Average packet size	181.629 Bytes
TCP ports used	443, 3001, 5672

Discussion

Using Wireshark and filtering all the packets sent and received between the client and the server, we are able to determine all packets are sent using the HTTPS protocol. Because the packets are encrypted using TLSv1, all the traffic to the REST interface is secured and will not be vulnerable to packet sniffing and man-in-the-middle attacks. A second and very positive result from this test reveals that the bandwidth required between the client and the server is very low. With an internet connection of 1Mbps, the key streaming would result in a 0.007% utilisation. This indicates that the low Internet bandwidth requirement set out by the design criteria is met.

6.3 Integration Testing

The tests in this section evaluate the integration of the individual components and the platform as whole, highlighting important features. This test will comprise of the primary server, the monitoring server and a number of client applications running on a variety of operating systems. These operating systems include Mac OS X and virtualised Windows and Linux operating systems. Although these client applications will be running concurrently with users testing them, specific tests are performed during the platform test on a single Macintosh client.

6.3.1 Video Playback

Objective

Video playback is the primary function of the client application and the objective of this test is to evaluate the playback environment for the user.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Log in to the client application.
3. Start playing a video.

Results

The client application allows any video in the playlist to be played, and provides volume control, play, pause, mute and full screen functionality. The application indicates progress with the seek slider and video time display.

One cause for concern introduced by file playback is the delay when starting to play a video. The video player must create a temporary (duplicate) file before playback can begin. This process varies by the size of the video file but can take up to 25 seconds for a 540 MB file.

6.3.2 Video Search and Download

Objective

The objective of this test is to evaluate video search and download on the client application. The test also describes the procedure in which the user performs a search and download.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Log in to the client application.
3. In the search field enter the search query and click search.

4. The tabbed widget will switch to the video search tab and list the results. Double click on a video to start downloading.

5. Monitor network throughput during the download.

6. Wait for the download to complete and repeat steps 3 through 5 with different video files sizes.

7. Repeat steps 3 through 6 adding an additional seeding peer on each iteration.

Results

Figure 6.4 graphs the results for the download test. The figure compares the average download rates of three different files by adding peers to the network. The speeds listed in the figure are given in megabytes per second [MBps].

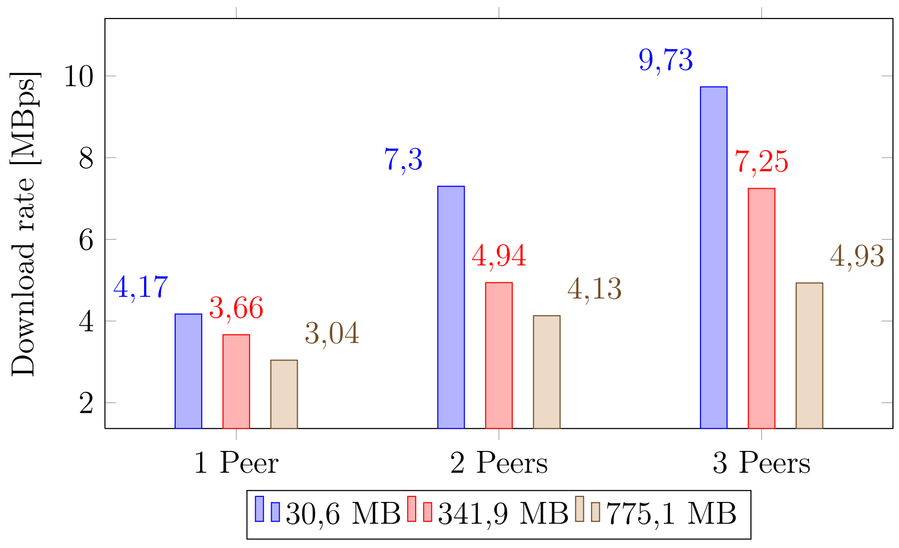


Figure 6.4: Downloading performance of three different file sizes when adding peers to the network.

Discussion

The test is performed on a gigabit local area network and the performance of the underlying Jax.TV protocol performs as expected and increases as additional seeding peers are added to the network. During a video download from one peer the download rate is significantly lower than with multiple peers.

This can be attributed to the overhead in the peer-to-peer protocol that is operating in a mode that is actually client-server based. The performance does decrease slightly as the file size increases but can be fixed with updates to the underlying Jax.TV protocol. It is important to remember when looking at the results that Jax.TV is protocol technology demonstrator and not production ready.

The download manager of the client application executed as expected, queueing video downloads and providing the user with an indication of which videos are still in the queue to be downloaded. The videos that finish downloading appear in the playlist after the playlist is refreshed every 30 seconds.

6.3.3 Alternative Video Downloading

Objective

One of the specifications of the project is to allow videos to be downloaded through other transport mechanisms such as USB flash drives or external hard drives. For this test, the client application is tested to see whether is allows videos to be loaded from other sources.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Log in to the client application.
3. Allow the playlist to populate with existing video content on the computer.
4. Once the playlist is populated, copy a video that is cached on one peer to this peer using a USB flash drive. The video must be copied into the media directory.
5. Wait for application to include the video as part of the playlist.

Results

This functionality of the application worked exactly as prescribed. Thirty seconds after the playlist was initially populated and the downloaded video was added, it appeared in the playlist.

6.3.4 Usage Tracking

Objective

Usage tracking is a specification of the platform that allow business models to be applied to the platform. This objective of this test ensure that the both real-time and long-term statistics are tracked accurately.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Log in to the client application.
3. Perform a search for a video.
4. Download a video from the search results.
5. Wait for the download to complete and play the video.
6. Logout of the client application.
7. During each one of steps monitor usage tracking.

Results

No user interface is available for the usage tracking and requires use of the command line facilities for both MonogoDB and MySQL. For the long-term and the real-time statistics, each step mentioned above was accurately recorded in the respective databases.

One issue that did arise within the statistics was date and time synchronisation between clients. In order to move as much processing as possible to clients, the date and time acquisitions are done on the client. This is posted as an entire message to the REST interface and saved directly into the database. During the test, one of the clients was accidentally configured for an incorrect time zone. For the this reason the information sent to the REST interface was off by two hours.

It is also possible that the date and time on clients may not be accurate. In order to address these issues, the REST interface must be used to acquire the date and time, as the REST interface can be configured to synchronise to time servers.

6.3.5 Central Logging

Objective

One of the key features of the platform is the ability to central view logs from servers and clients. The objective of this test to evaluate the functionality of this feature.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Monitor the logs on Graylog2 during the integration test.

Results

Graylog2 is configured on the platform to split the logs into different streams. Streams are created for the all the logs sent by the client applications, Django error logs, Django debugging logs, the Jax.TV tracker and the encryption logs.

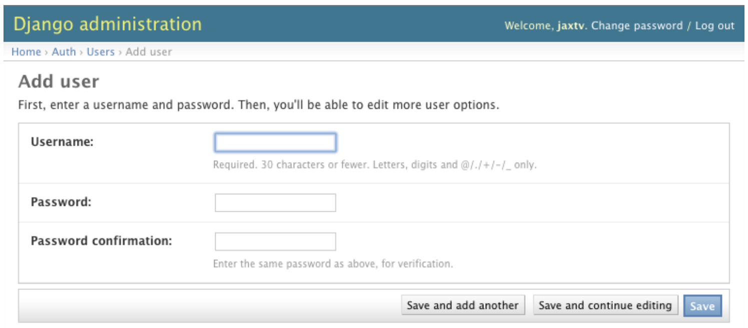
6.3.6 Modifying Video Information

Objective

An important feature for platform administrators is the ability to modify video information and mark the availability of videos. The objective of this test is to test this functionality.

Protocol

1. Perform the standard set up described in Section 6.1.
2. Log in to the client application.
3. Search for a video that is not marked as available.
4. Mark the item as available in the Django admin panel and repeat the search on the client application.
5. The process but mark the video as available directly from the MySQL database.


Django administration
Welcome, **jaxtv**. Change password / Log out
Home › Auth › Users › Add user

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:
Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Password confirmation:
Enter the same password as above, for verification.

Figure 6.5: A screen shot of adding a user through the Django administration panel.

Results

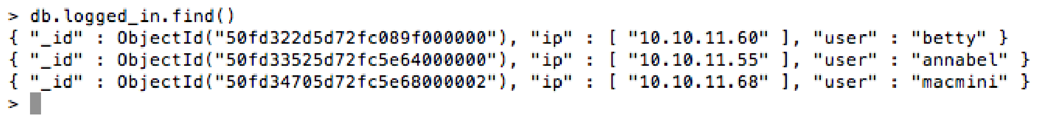
The video information that is modified through both the Django admin panel and the database directly, reflects immediately on the client application. This applies to the video metadata information as well.

6.4 Platform Testing

The final test performs a qualitative test in order to provide an overview of all the components of the project operating together as the entire platform. The test starts off by looking at the client application, before demonstrating the administrative features of the platform. This section includes screen shots of the administrative interface and the client application, however, it is difficult to provide a complete picture of the platform. For this reason a video screen recording is available on YouTube at `http://youtu.be/4U_tbCBCWKI`. Additionally, the source code for the project has been made available on Dropbox at `https://www.dropbox.com/sh/nhb5rz67ypi6jgd/5oYW2X87Q1`.

The test set up follows the procedure detailed in Section 6.1 and includes two Macintosh clients and one Linux client. The first requirement for the test is to create users for each client via the Django Admin Panel. A screenshot of adding a user is given in Figure 6.5.

With all the users created, the next step is to launch the client application and log the users into the three applications. Once the users are logged in, they are presented with an empty playlist. In order to download video content, a search is performed using the application and the results are listed in the search tab. Double clicking on any entry in the search results, places the



```
> db.logged_in.find()
{ "_id" : ObjectId("50fd322d5d72fc089f000000"), "ip" : [ "10.10.11.60" ], "user" : "betty" }
{ "_id" : ObjectId("50fd33525d72fc5e64000000"), "ip" : [ "10.10.11.55" ], "user" : "annabel" }
{ "_id" : ObjectId("50fd34705d72fc5e68000002"), "ip" : [ "10.10.11.68" ], "user" : "macmini" }
> 
```

Figure 6.6: Real-time information about users logged onto the platform.

video on the download queue. Once a few videos are placed on the download queue, each video remains listed on the downloads queue until the download is completed. The user also has the option of downloading a video, that belongs to the platform, using alternative distribution mechanisms. Examples of these include; copying videos using flash drives, or downloading videos from other users using FTP or HTTP downloads. These videos need to be placed in the media directory in order to be scanned and listed in the playlist.

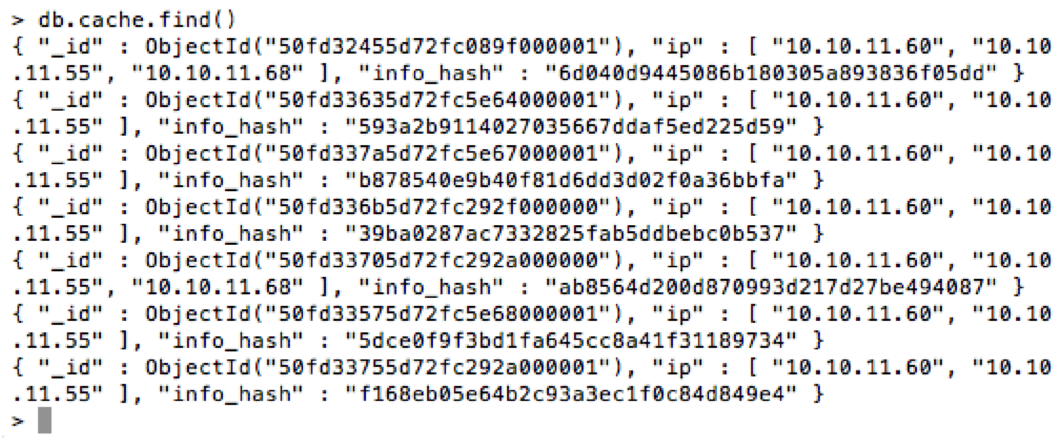
The downloaded videos appear in the playlist after they have finished downloading and the playlist is updated. Now the videos are ready for playback by the user and the user can play any video that is listed in the playlist. During playback the user has the option to pause and resume playback, enter and exit full screen mode and adjust or mute the volume of the video. Additionally, all videos that are listed in the playlist are made available for uploading to other peers on the network.

In order to monitor real-time activity of the clients, the MongoDB shell is used. All the real-time information is stored under one database using multiple collections. Listing all entries in the `logged_in` collection reveals all the users that are currently logged onto the platform and is illustrated in Figure 6.6.

Figure 6.7 provides the real-time information about the video data that is cached by the clients on the network. This information is stored in the `cache` collection within MongoDB, while the real-time information about users watching videos is stored in the `playing` collection. Figure 6.8 indicates that at the point of the screen shot during the test, two users were watching the same video, while a third was watching a different video.

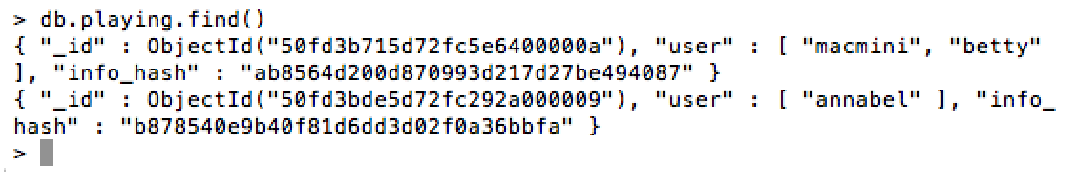
MySQL is used for gathering and retrieving information stored over a long period of time. Figure 6.9 shows the MySQL shell being used on the primary server to access video searches that have been performed by users on platform, stored in the `table_video_searches` table within the database.

In order to view the centralised logging system, Graylog2 can be accessed



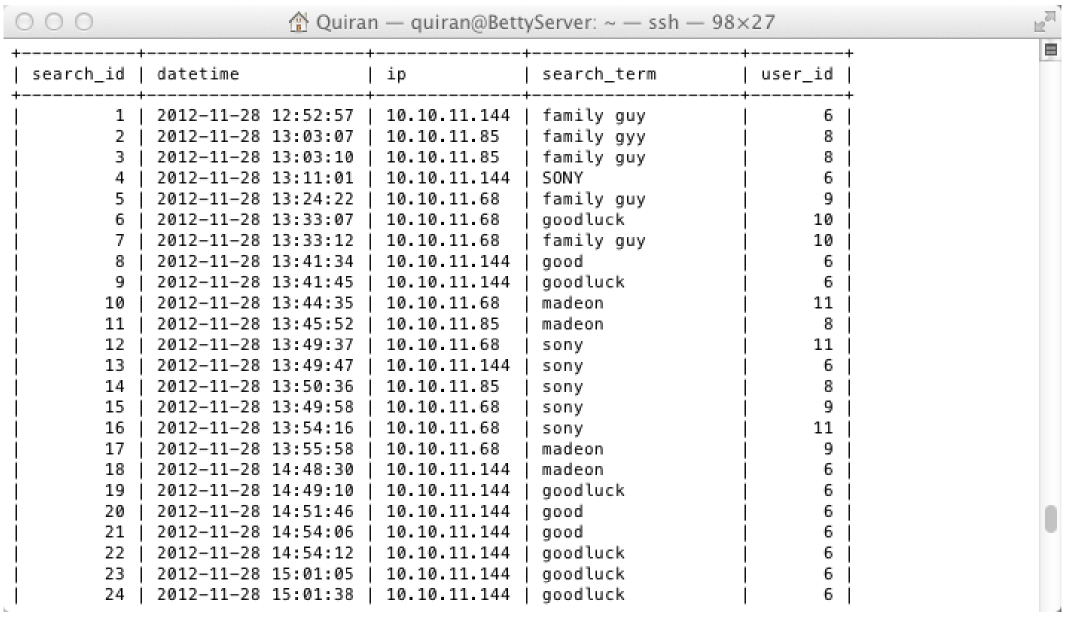
```
> db.cache.find()
{ "_id" : ObjectId("50fd32455d72fc089f000001"), "ip" : [ "10.10.11.60", "10.10
.11.55", "10.10.11.68" ], "info_hash" : "6d040d9445086b180305a893836f05dd" }
{ "_id" : ObjectId("50fd33635d72fc5e64000001"), "ip" : [ "10.10.11.60", "10.10
.11.55" ], "info_hash" : "593a2b9114027035667ddaf5ed225d59" }
{ "_id" : ObjectId("50fd337a5d72fc5e67000001"), "ip" : [ "10.10.11.60", "10.10
.11.55" ], "info_hash" : "b878540e9b40f81d6dd3d02f0a36bbfa" }
{ "_id" : ObjectId("50fd336b5d72fc292f000000"), "ip" : [ "10.10.11.60", "10.10
.11.55" ], "info_hash" : "39ba0287ac7332825fab5ddbebc0b537" }
{ "_id" : ObjectId("50fd33705d72fc292a000000"), "ip" : [ "10.10.11.60", "10.10
.11.55", "10.10.11.68" ], "info_hash" : "ab8564d200d870993d217d27be494087" }
{ "_id" : ObjectId("50fd33575d72fc5e68000001"), "ip" : [ "10.10.11.60", "10.10
.11.55" ], "info_hash" : "5dce0f9f3bd1fa645cc8a41f31189734" }
{ "_id" : ObjectId("50fd33755d72fc292a000001"), "ip" : [ "10.10.11.60", "10.10
.11.55" ], "info_hash" : "f168eb05e64b2c93a3ec1f0c84d849e4" }
> 
```

Figure 6.7: Real-time information about video data cached by clients on the platform.



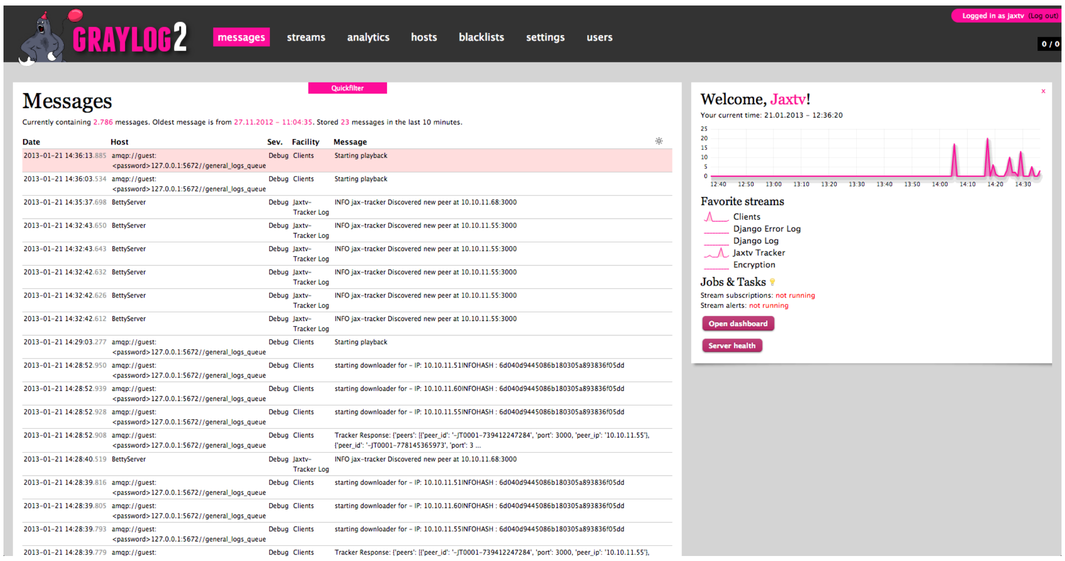
```
> db.playing.find()
{ "_id" : ObjectId("50fd3b715d72fc5e6400000a"), "user" : [ "macmini", "betty"
], "info_hash" : "ab8564d200d870993d217d27be494087" }
{ "_id" : ObjectId("50fd3bde5d72fc292a000009"), "user" : [ "annabel" ], "info_
hash" : "b878540e9b40f81d6dd3d02f0a36bbfa" }
> 
```

Figure 6.8: Real-time information about videos currently being played on the platform.



```
+-----------+---------------------+-------------+---------------------+---------+
| search_id | datetime            | ip          | search_term         | user_id |
+-----------+---------------------+-------------+---------------------+---------+
|         1 | 2012-11-28 12:52:57 | 10.10.11.144 | family guy         |       6 |
|         2 | 2012-11-28 13:03:07 | 10.10.11.85  | family gyy         |       8 |
|         3 | 2012-11-28 13:03:10 | 10.10.11.85  | family guy         |       8 |
|         4 | 2012-11-28 13:11:01 | 10.10.11.144 | SONY               |       6 |
|         5 | 2012-11-28 13:24:22 | 10.10.11.68  | family guy         |       9 |
|         6 | 2012-11-28 13:33:07 | 10.10.11.68  | goodluck           |      10 |
|         7 | 2012-11-28 13:33:12 | 10.10.11.68  | family guy         |      10 |
|         8 | 2012-11-28 13:41:34 | 10.10.11.144 | good               |       6 |
|         9 | 2012-11-28 13:41:45 | 10.10.11.144 | goodluck           |       6 |
|        10 | 2012-11-28 13:44:35 | 10.10.11.68  | madeon             |      11 |
|        11 | 2012-11-28 13:45:52 | 10.10.11.85  | madeon             |       8 |
|        12 | 2012-11-28 13:49:37 | 10.10.11.68  | sony               |      11 |
|        13 | 2012-11-28 13:49:47 | 10.10.11.144 | sony               |       6 |
|        14 | 2012-11-28 13:50:36 | 10.10.11.85  | sony               |       8 |
|        15 | 2012-11-28 13:49:58 | 10.10.11.68  | sony               |       9 |
|        16 | 2012-11-28 13:54:16 | 10.10.11.68  | sony               |      11 |
|        17 | 2012-11-28 13:55:58 | 10.10.11.68  | madeon             |       9 |
|        18 | 2012-11-28 14:48:30 | 10.10.11.144 | madeon             |       6 |
|        19 | 2012-11-28 14:49:10 | 10.10.11.144 | goodluck           |       6 |
|        20 | 2012-11-28 14:51:46 | 10.10.11.144 | good               |       6 |
|        21 | 2012-11-28 14:54:06 | 10.10.11.144 | good               |       6 |
|        22 | 2012-11-28 14:54:12 | 10.10.11.144 | goodluck           |       6 |
|        23 | 2012-11-28 15:01:05 | 10.10.11.144 | goodluck           |       6 |
|        24 | 2012-11-28 15:01:38 | 10.10.11.144 | goodluck           |       6 |
```

Figure 6.9: MySQL shell being used to access video searches performed by users.



Messages

Currently containing 2.786 messages. Oldest message is from 27.11.2012 - 11:04:35. Stored 23 messages in the last 10 minutes.

Date	Host	Sev.	Facility	Message
2013-01-21 14:36:13.885	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	Starting playback
2013-01-21 14:36:03.534	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	Starting playback
2013-01-21 14:35:37.698	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.68:3000
2013-01-21 14:32:43.650	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.55:3000
2013-01-21 14:32:43.643	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.55:3000
2013-01-21 14:32:42.632	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.55:3000
2013-01-21 14:32:42.626	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.55:3000
2013-01-21 14:32:42.612	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.55:3000
2013-01-21 14:29:03.277	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	Starting playback
2013-01-21 14:28:52.950	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	starting downloader for - IP: 10.10.11.51INFOHASH : 6d040d9445086b180305a893836f05dd
2013-01-21 14:28:52.939	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	starting downloader for - IP: 10.10.11.60INFOHASH : 6d040d9445086b180305a893836f05dd
2013-01-21 14:28:52.928	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	starting downloader for - IP: 10.10.11.55INFOHASH : 6d040d9445086b180305a893836f05dd
2013-01-21 14:28:52.908	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	Tracker Response: {'peers': [{'peer_id': '-JT0001-739412247284', 'port': 3000, 'peer_ip': '10.10.11.55'}, {'peer_id': '-JT0001-778145365973', 'port': 3 ...
2013-01-21 14:28:40.519	BettyServer	Debug	Jaxtv-Tracker Log	INFO jax-tracker Discovered new peer at 10.10.11.68:3000
2013-01-21 14:28:39.816	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	starting downloader for - IP: 10.10.11.51INFOHASH : 6d040d9445086b180305a893836f05dd
2013-01-21 14:28:39.805	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	starting downloader for - IP: 10.10.11.60INFOHASH : 6d040d9445086b180305a893836f05dd
2013-01-21 14:28:39.793	amqp://guest:<password>127.0.0.1:5672//general_logs_queue	Debug	Clients	starting downloader for - IP: 10.10.11.55INFOHASH : 6d040d9445086b180305a893836f05dd
2013-01-21 14:28:39.779	amqp://guest:	Debug	Clients	Tracker Response: {'peers': [{'peer_id': '-JT0001-739412247284', 'port': 3000, 'peer_ip': '10.10.11.55'},

Welcome, Jaxtv!

Your current time: 21.01.2013 - 12:36:20

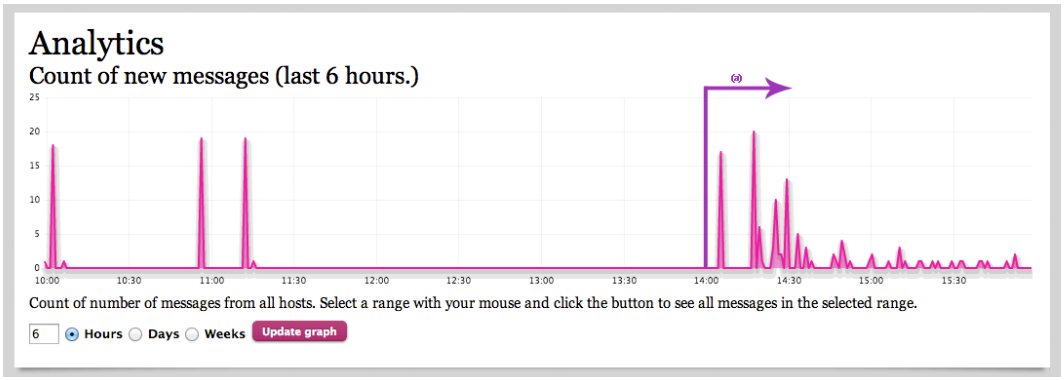
Favorite streams

- Clients
- Django Error Log
- Django Log
- Jaxtv Tracker
- Encryption

Jobs & Tasks

Stream subscriptions: not running
Stream alerts: not running

Figure 6.10: Graylog2 web interface displaying the log message stream.



Count of number of messages from all hosts. Select a range with your mouse and click the button to see all messages in the selected range.

Figure 6.11: Graylog2 web interface displaying the log message analytics with the test period indicated at (a).

via the admin portal. Graylog2 provides a authenticated, graphical interface to view and categorise log messages. All log messages are filtered into streams by Graylog2 according to their categories. The streams are visible on the right of Figure 6.10 and includes streams for all the client log messages, the Django error log messages, the Django informative log messages, the Jax.TV tracker log messages and the log messages produced by the video encryption module. Graylog2 also provides analytics with regard to message throughput on the service and this is given in Figure 6.11.

Munin was used to monitor the primary server during the final test. The graphs that Munin produces include, Apache web server volume (Figure 6.12), the CPU usage (Figure 6.13), the memory usage (Figure 6.14), and the primary server's network traffic (Figure 6.15). Although the primary server is a virtualised operating system and hosts various services on the same system, the network traffic and load on the server is very low during the test.

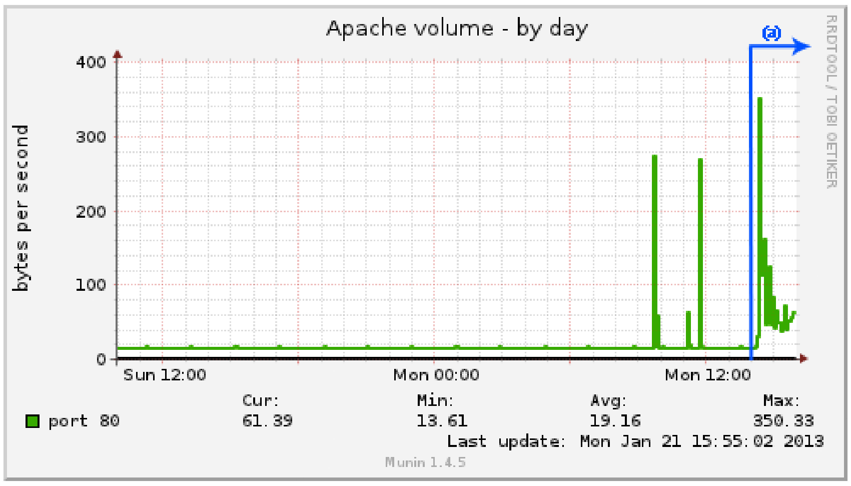
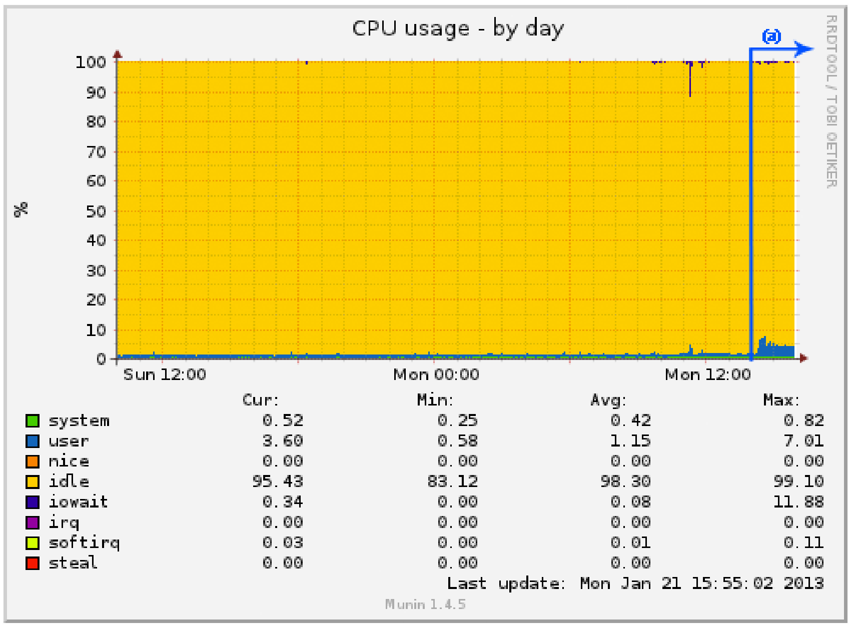


Figure 6.12: Munin displaying Apache volumes during the test period indicated at (a).

6.5 Summary

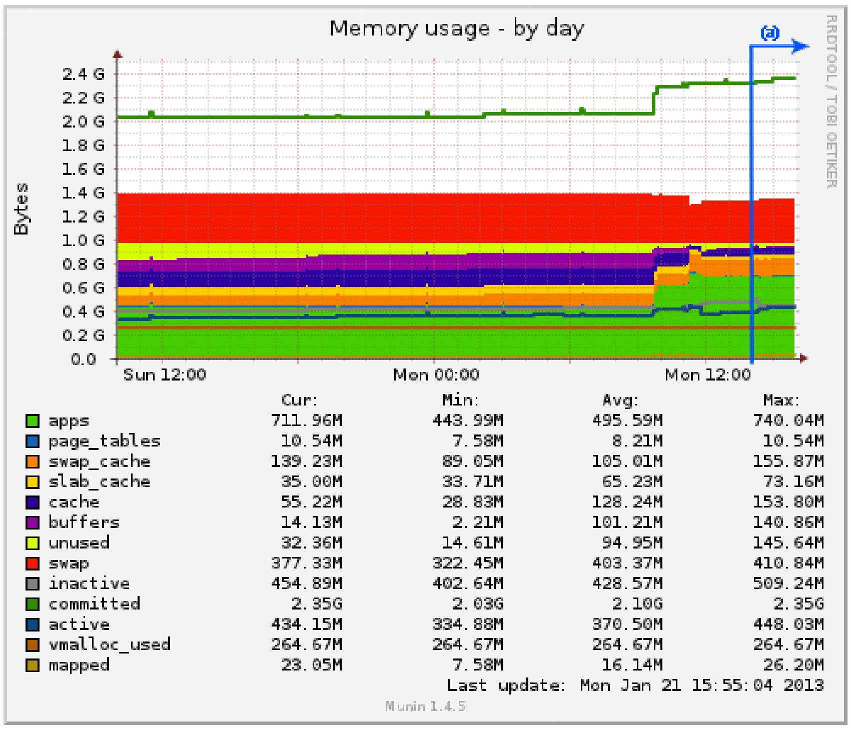
This chapter presented the results found in evaluating the core components of the platform and the prototype application. Most importantly, the security aspects of the platform design performed as expected. We are able to fully encrypt a video file, decrypt blocks of the file in real-time for playback and erase decrypted data. Furthermore, an evaluation procedure was developed to test if data is correctly erased with random data.



	Cur:	Min:	Avg:	Max:
🟩 system	0.52	0.25	0.42	0.82
🟦 user	3.60	0.58	1.15	7.01
🟧 nice	0.00	0.00	0.00	0.00
🟨 idle	95.43	83.12	98.30	99.10
🟦 iowait	0.34	0.00	0.08	11.88
🟪 irq	0.00	0.00	0.00	0.00
🟨 softirq	0.03	0.00	0.01	0.11
🟥 steal	0.00	0.00	0.00	0.00

Last update: Mon Jan 21 15:55:02 2013

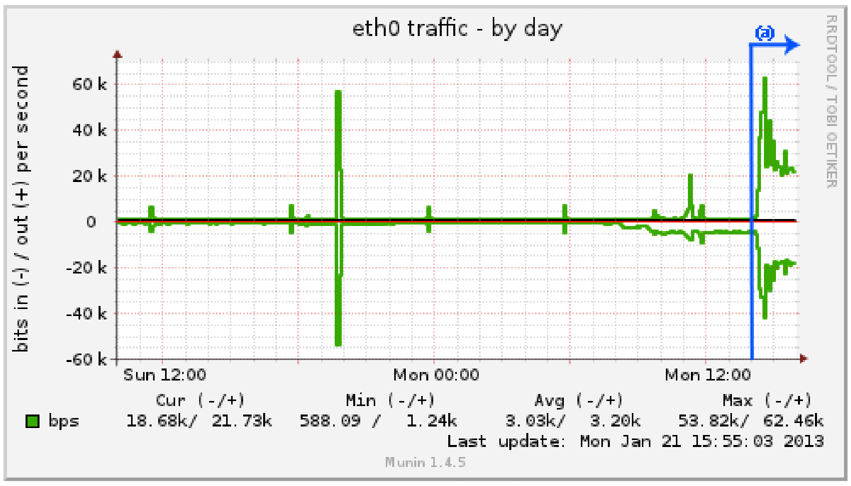
Figure 6.13: Munin displaying CPU usage during the period indicated at (a).



	Cur:	Min:	Avg:	Max:
🟩 apps	711.96M	443.99M	495.59M	740.04M
🟦 page_tables	10.54M	7.58M	8.21M	10.54M
🟧 swap_cache	139.23M	89.05M	105.01M	155.87M
🟨 slab_cache	35.00M	33.71M	65.23M	73.16M
🟦 cache	55.22M	28.83M	128.24M	153.80M
🟪 buffers	14.13M	2.21M	101.21M	140.86M
🟨 unused	32.36M	14.61M	94.95M	145.64M
🟥 swap	377.33M	322.45M	403.37M	410.84M
⬜ inactive	454.89M	402.64M	428.57M	509.24M
🟩 committed	2.35G	2.03G	2.10G	2.35G
🟦 active	434.15M	334.88M	370.50M	448.03M
🟫 vmalloc_used	264.67M	264.67M	264.67M	264.67M
🟫 mapped	23.05M	7.58M	16.14M	26.20M

Last update: Mon Jan 21 15:55:04 2013

Figure 6.14: Munin displaying memory usage during the test period indicated at (a).



```
        Cur (-/+)          Min (-/+)         Avg (-/+)         Max (-/+)
bps  18.68k/  21.73k   588.09 /   1.24k   3.03k/   3.20k   53.82k/  62.46k
                          Last update:  Mon Jan 21 15:55:03 2013
```

Figure 6.15: Munin displaying network traffic during the test period indicated at (a).

Chapter 7

Conclusion

The aim of this project was to design and implement a video entertainment platform that allows users to watch high quality video content across multiple devices. The outcome is a platform that relies on peer-to-peer sharing and a security scheme that allows this peer-to-peer nature of sharing while preventing piracy and allowing the monitoring of video usage.

7.1 Evaluation

We designed and implemented a platform that contains a peer-to-peer distribution mechanism that is based on the GoalBit platform but is also free to allow the exchange of video data through other distribution mechanisms. These can include BitTorrent and users sharing video files using external hard drives or USB flash drives.

In order to secure the video content, we created an encryption scheme that is based on the one-time pad to encrypt the video data on a bit level and uses a low bandwidth connection to a key server so that the keys can be used to unlock the video file. The encryption scheme breaks the file into blocks and encrypts each block individually. This way, if an attacker is able to crack a block or guess a key, the whole video file is not compromised. Because the platform allows the video content to freely be distributed, a real-time connection to the key server prevents the content from being unlocked and watched outside of the platform.

In order to play back encrypted content, we created a prototype client application to demonstrate video playback and additional services like search and video download. The client application provides the only means for a registered user on the platform to watch an encrypted video. The prototype application allows the user to search for available videos and download these videos. The client application also seeds video data back onto the network for other users to download.

For feasible business models to be applied to the platform, we created a way for platform administrators to manage the content on the platform and monitor the usage and adoption of video content amongst users. Platform administrators have the ability to add or revoke content from the platform. Platform administrators are also able to view real-time statistics about users logged into the platform, videos being played and video files that are cached on users' devices. Furthermore, long-term statistics are gathered about user logins, as well as video searches, downloads and plays.

Finally, the platform serves as a modular extension of Jax.TV and uses existing technologies and products (where available) to achieve this. Where existing solutions were not readily available, such as support for real-time monitoring, they were created specifically for the platform.

7.2 Contributions

This project uses existing technologies and solutions to create a video entertainment platform that:

- uses peer-to-peer to distribute video data;
- offers an alternative to existing DRM solutions;
- that is able to support business models;
- offers a video encryption scheme that is video codec independent; and
- creates a platform that addresses the need for high quality video distribution in Internet bandwidth restricted regions.

7.3 Recommendations

Although this project only focused on downloaded video using Jax.TV, future work can allow the security implementation to be extended to real-time video streaming on Jax.TV. This would require additional functionality, such as real-time video encryption capabilities on the streaming devices and real-time video decryption on the client application. Due to the design proposed in this project, a redesign of the encryption scheme is not required to implement real-time video streaming.

We also recommend that research be done into advertising mechanisms for the platform as a revenue stream. Finally, recommendation systems should be incorporated with network management to automatically download recommended videos for users when the network is idle.

7.4 Summary

The platform developed in this project provides an alternative to users in bandwidth restricted areas to high quality video that would otherwise only be able to users with high capacity Internet bandwidth. The platform provides a base to be customised and extended with features where required by the individual deployment.

Bibliography

[1] Netflix Inc.: Netflix - Watch TV Shows Online, Watch Movies Online. Available: http://www.netflix.com/, 2011. [Online: Accessed 12 March 2011].

[2] Disney: The Walt Disney Studios. Available: http://www.waltdisneystudios.com, 2012. [Online: Accessed 05 December 2012].

[3] Farivar, C.: Disney anoints Netflix as its exclusive distributor in 2016. Available: http://arstechnica.com/business/2012/12/disney-anoints-netflix-as-its-exclusive-distributor-starting-in-2016/, 2012. [Online: Accessed 05 December 2012].

[4] YouTube LLC: YouTube - Broadcast Yourself. Available: http://www.youtube.com/, 2011. [Online: Accessed: 12 March 2011].

[5] Apple Inc.: Apple - iTunes. Available: http://www.apple.com/itunes/what-is/, 2012. [Online: Accessed 05 October 2012].

[6] MultiChoice Africa (Pty) Ltd: Multichoice | History. Available: http://www.multichoice.co.za/multichoice/view/multichoice/en/page44122, 2012. [Online: Accessed 05 October 2012].

[7] YouTube LLC: About YouTube. Available: http://www.youtube.com/t/about_youtube, 2012. [Online: Accessed 05 October 2012].

[8] YouTube LLC: Our Solutions - YouTube. Available: http://www.youtube.com/yt/advertise/our-solutions.html, 2012. [Online: Accessed: 28 October 2012].

[9] YouTube LLC: YouTube Timeline. Available: http://www.youtube.com/t/press_timeline, 2012. [Online: Accessed 05 October 2012].

[10] IOC: IOC to live stream London 2012 in 64 territories on its YouTube channel. Available: http://www.olympic.org/news/ioc-to-live-stream-

`london-2012-in-64-territories-on-its-youtube-channel/166482`, 2012. [Online: Accessed 19 September 2012].

[11] MultiChoice Africa (Pty) Ltd: Video on Demand - DStv on Demand. Available: `http://ondemand.dstv.com/find-out-more`, 2012. [Online: Accessed 05 October2012].

[12] MultiChoice Africa (Pty) Ltd: BoxOffice | Frequently asked questions. Available: `http://boxoffice.dstv.com/help`, 2012. [Online: Accessed 05 October 2012].

[13] Jensen, M.: Lowering the costs of international bandwidth in Africa. *Association for Progressive Communications, San Francisco*, 2006.

[14] International Telecommunication Union: Key statistical highlights: ITU data release June 2012. Available: `http://www.itu.int/ITU-D/ict/statistics/material/pdf/2011Statisticalhighlights_June_2012.pdf`, 2012. [Online: Accessed 06 October 2012].

[15] Bertinat, M.E., De Vera, D., Padula, D., Amoza, F.R., Rodríguez-Bocca, P., Romero, P. and Rubino, G.: GoalBit. In: *Proceedings of the 5th International Latin American Networking Conference on - LANC '09*, p. 49. ACM Press, New York, New York, USA, 2009. ISBN 9781605587752.

[16] Bruwer, J.: Peer-to-peer video streaming over fast local networks. Unpublished.

[17] City of Cape Town: Gated development policy. Available: `http://www.capetown.gov.za/en/planningandbuilding/Publications/LandUseManagement/Documents/GatedDevelopmentPolicy.pdf`, November 2007. [Online: Accessed 07 October 2012].

[18] Wireless User Groups South Africa. Available: `http://www.wug.za.net`, 2012. [Online: Accessed 07 October 2012].

[19] Cape Town Wireless User Group. Available: `http://www.ctwug.za.net/content.php`, 2012. [Online: Accessed 09 October 2012].

[20] Pretoria Wireless User Group. Available: `http://www.ptawug.co.za`, 2012. [Online: Accessed 09 October 2012].

[21] Biddle, P., England, P., Peinado, M. and Willman, B.: The darknet and the future of content distribution. In: *ACM Workshop on Digital Rights Management*, vol. 6, p. 54. 2002.

[22] Jamkhedkar, P. and Heileman, G.: Digital rights management architectures. *Computers & Electrical Engineering*, vol. 35, no. 2, pp. 376–394, 2009.

[23] BitTorrent.org: The BitTorrent Protocol Specification. Available: `http://www.bittorrent.org`, 2008. [Online: Accessed 14 February 2012].

[24] DC++ your files, your way, no limits. Available: `http://dcplusplus.sourceforge.net`, 2012. [Online: Accessed 15 October 2012].

[25] Tanenbaum, A.S.: *Computer Networks*. 4th edn. 2005. ISBN 0-13-038488-7.

[26] Parameswaran, M., Susarla, A. and Whinston, A.: P2P networking: an information sharing alternative. *Computer*, vol. 34, no. 7, pp. 31–38, 2001.

[27] Cohen, B.: BitTorrent Protocol Specification. Available: `http://bittorrent.org/beps/bep_0003.html`, 2008. [Online: Accessed 05 August 2012].

[28] Cohen, B.: Incentives build robustness in BitTorrent. In: *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72. 2003.

[29] Li, J., Cui, Y. and Chang, B.: Peerstreaming: design and implementation of an on-demand distributed streaming system with digital rights management capabilities. *Multimedia Systems*, vol. 13, no. 3, pp. 173–190, 2007.

[30] Python Programming Language. Available: `http://www.python.org`, 2012. [Online: Accessed 16 October 2012].

[31] Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, RFC Editor, July 2006.
Available at: `http://www.rfc-editor.org/rfc/rfc4627.txt`

[32] DECE LLC: Digital Entertainment Content Ecosystem Unveils UltraViolet Brand. Available: `http://www.uvvu.com/press/UltraViolet_Brand_Launch_Release_07_20_2010_FINAL.pdf`, 2010. [Online: Accessed 23 September 2012].

[33] Smith, E.: Disney Touts a Way to Ditch the DVD. Available: `http://online.wsj.com/article/SB20001424052748703816204574485650026945222.html`, 2009. [Online: Accessed 23 September 2012].

[34] Liu, F. and Koenig, H.: A survey of video encryption algorithms. *Computers & Security*, vol. 29, no. 1, pp. 3–15, February 2010. ISSN 01674048.

[35] Wen, J., Severa, M., Zeng, W., Luttrell, M. and Jin, W.: A format-compliant configurable encryption framework for access control of video. *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 12, no. 6, pp. 545–557, 2002.

[36] Iwata, T., Abe, T., Ueda, K. and Sunaga, H.: A DRM system suitable for P2P content delivery and the study on its implementation. In: *9th Asia-Pacific Conference on Communications (IEEE Cat. No.03EX732)*, vol. 2, pp. 806–811. IEEE, IEEE, 2003. ISBN 0-7803-8114-9.

[37] Haskell, B., Puri, A. and Netravali, A.: *Digital video: an introduction to MPEG-2*. Springer, 1996. ISBN 0412084112.

[38] Angelides, M.C. and Agius, H.: *The Handbook of MPEG Applications*. John Wiley & Sons, Ltd, Chichester, UK, November 2010. ISBN 9780470974582.

[39] MacAulay, A., Felts, B. and Fisher, Y.: WHITEPAPER–IP Streaming of MPEG-4: Native RTP vs MPEG-2 Transport Stream. 2005.

[40] Kwon, S., Tamhankar, A. and Rao, K.: Overview of H.264/MPEG-4 part 10. *Journal of Visual Communication and Image Representation*, vol. 17, no. 2, pp. 186–216, April 2006. ISSN 10473203.

[41] Kaliski, B.: A survey of encryption standards. *Micro, IEEE*, vol. 13, no. 6, pp. 74–81, 1993.

[42] Katz, J. and Lindell, Y.: *Introduction to Modern Cryptography*, vol. 3. Chapman & Hall/CRC, 2007.

[43] Qiao, L. and Nahrstedt, K.: Comparison of MPEG encryption algorithms. *Computers & Graphics*, vol. 22, no. 4, pp. 437–448, 1998.

[44] Lan, X., Xue, J., Tian, L., Hu, W., Xu, T. and Zheng, N.: A Peer-to-Peer Architecture for Live Streaming with DRM. January 2009.
Available at: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4784956`

[45] Matthews, R.: On the derivation of a "chaotic" encryption algorithm. *Cryptologia*, vol. 13, no. 1, pp. 29–42, 1989.

[46] Filippini, a., Bergamo, P. and Mazzini, G.: Security issues based on chaotic systems. In: *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE*, vol. 1, pp. 148–152. IEEE, 2002.

[47] Cohen, J.E.: DRM and privacy. *Communications of the ACM*, vol. 46, no. 4, pp. 46–49, April 2003. ISSN 00010782.

[48] Nokia: Forum Nokia Library. Available: `http://library.forum.nokia.com/index.jsp?topic=/S60_5th_Edition_Cpp_Developers_Library/GUID-C29F822D-640C-47F0-89C3-92D17DA31D49.html`, 2009. [Online: Accessed 16 March 2011].

[49] Subramanya, S. and Yi, B.: Digital rights management. *IEEE Potentials*, vol. 25, no. 2, pp. 31–34, March 2006. ISSN 0278-6648.

[50] Microsoft: Microsoft PlayReady Content Access Technology White Paper. Available: `http://download.microsoft.com/download/b/8/3/b8316f44-e7a9-48ff-b03a-44fb92a73904/Microsoft%20PlayReady%20Content%20Access%20Technology-Whitepaper.docx`, 2008. [Online: Accessed 29 March 2011].

[51] Kalker, T., Samtani, R. and Wang, X.: UltraViolet: Redefining the Movie Industry? *MultiMedia, IEEE*, pp. 7–11, 2012.

[52] Irdeto: About Irdeto. Available: `http://irdeto.com/about-irdeto.html`, 2012. [Online: Accessed 17 November 2012].

[53] Rosenblatt, B.: Irdeto Sets Next Level in Video Content Protection. Available: `http://copyrightandtechnology.com/2011/03/07/irdeto-sets-next-level-in-video-content-protection/`, 2011. [Online: Accessed 31 March 2011].

[54] Irdeto: Irdeto ActiveCloak for Media - Core Technology The Key to Effective Content Protection. Available: `http://irdeto.com/documents/so_ac_ct_en.pdf`, 2012. [Online: Accessed 05 November 2012].

[55] Irdeto: Irdeto ActiveCloak for Media - Dynamic security for any Over-the-Top service. Available: `http://irdeto.com/documents/OV_AC_OTT_EN_L.pdf`, 2012. [Online: Accessed 05 November 2012].

[56] de Kosnik, A.: Piracy is the future of television. *University of California, Berkeley*, 2010.

[57] Python Software Foundation: About | Python. Available: `http://www.python.org/about/`, 2012. [Online: Accessed 09 November 2012].

[58] Qt - a cross-platform application and UI framework. Available: `http://qt.nokia.com/products/`, 2012. [Online: Accessed 19 August 2012].

[59] Git: Git. Available: `http://git-scm.com`, 2012. [Online: Accessed 09 November 2012].

[60] Fabric. Available: `http://docs.fabfile.org/en/1.4.3/index.html`, 2012. [Online: Accessed 25 September 2012].

[61] Costello, R.L.: Building Web Services the REST Way. Available: `http://www.xfront.com/REST-Web-Services.html`. [Online: Accessed 09 March 2012].

[62] Fielding, R.: *Architectural styles and the design of network-based software architectures*. Ph.D. thesis, University of California, 2000.

[63] Django Software Foundation: Django. Available: `https://www.djangoproject.com`, 2012. [Online: Accessed 25 September 2012].

[64] Ruby on Rails. Available: `http://rubyonrails.org`, 2012. [Online: Accessed 09 November 2012].

[65] VideoLAN: VideoLAN. Available: `http://www.videolan.org/index.html`, 2012. [Online: Accessed 25 September 2012].

[66] PyMedia: PyMedia. Available: `http://pymedia.org`, 2004. [Online: Accessed 09 November 2012].

[67] pyglet. Available: `http://www.pyglet.org`, 2012. [Online: Accessed 09 November 2012].

[68] ffvideo - A python wrapper around ffmpeg. Available: `http://code.google.com/p/ffvideo/`, 2010. [Online: Accessed 09 November 2012].

[69] FFmpeg. Available: `http://ffmpeg.org`, 2012. [Online: Accessed 25 September 2012].

[70] Canonical Ltd.: Server | Ubuntu. Available: `http://www.ubuntu.com/business/server/overview`, 2012. [Online: Accessed 29 August 2012].

[71] VMware Inc.: VMware vSphere. Available: `http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html`, 2012. [Online: Accessed 29 August 2012].

[72] Apache Software Foundation: The Apache HTTP Server Project. Available: `http://httpd.apache.org`, 2012. [Online: Accessed 21 October 2012].

[73] Numpy: Scientific Computing Tools for Python - Numpy. Available: `http://numpy.scipy.org`, 2012. [Online: Accessed 10 September 2012].

[74] Matsumoto, M. and Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[75] Matsumoto, M. and Nishimura, T.: Cryptographic Mersenne Twister and Fubuki stream/block cipher. *Cryptographic ePrint Archive*, pp. 1–17, 2005.

[76] Amatriain, X. and Basilico, J.: The Netflix Tech Blog: Netflix Recommendations: Beyond the 5 stars (Part 1). Available: `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`, 2012. [Online: Accessed 05 October 2012].

[77] DC++ Features. Available: `http://dcplusplus.sourceforge.net/features.html`, 2012. [Online: Accessed 19 November 2012].

[78] Sphinx Technologies Inc.: About | Sphinx. Available: `http://sphinxsearch.com/about/sphinx/`, 2012. [Online: Accessed 27 February 2012].

[79] 10gen Inc: MongoDB. Available: `http://www.mongodb.org`, 2012. [Online: Accessed 22 October 2012].

[80] VMware Inc.: RabbitMQ - Messaging that just works. Available: `http://www.rabbitmq.com`, 2012. [Online: Accessed 22 August 2012].

[81] OASIS: Advanced Message Queuing Protocol. Available: `http://www.amqp.org`, 2012. [Online: Accessed 19 November 2012].

[82] logstash - open source log management. Available: `http://logstash.net`, 2012. [Online: Accessed 22 August 2012].

[83] Graylog2 - Free open source self-hosted log management and exception tracking. Available: `http://www.graylog2.org`, 2012. [Online: Accessed 22 August 2012].

[84] Kohl, U., Lotspiech, J. and Kaplan, M.: Safeguarding digital library contents and users. *D-lib Magazine*, vol. 3, no. 9, 1997.

[85] IMDB.com Inc.: IMDB. Available: `http://www.imdb.com`, 2012. [Online: Accessed 26 November 2012].

[86] Codec Guide: Codec Guide: K-Lite Codec Pack. Available: `http://codecguide.com`, 2013. [Online: Accessed 09 January 2013].

[87] MPlayer. Available: `http://www.mplayerhq.hu/`, 2011. [Online: Accessed 29 November 2012].

[88] Houdini Software: VideoSpec. Available: `http://videospec.free.fr/english/`, 2012. [Online: Accessed 29 November 2012].

[89] Datadog Inc.: Datadog. Available: `http://www.datadoghq.com`, 2012. [Online: Accessed 29 November 2012].

[90] Wireshark Foundation: Wireshark. Available: `http://www.wireshark.org/`, 2012. [Online: Accessed 02 November 2012].

Appendices

Appendix A

Software

This chapter lists the various software libraries, frameworks and operating systems that were used to develop and evaluate the platform. The system made use of two servers and three clients. These systems are documented below.

A.1 Primary Server

The primary server is responsible for hosting the REST interface, the centralised logging, the database systems, Jax.TV tracker, the full text search server, usage tracking, as well as handling the encryption.

Operating System

Ubuntu Linux Natty Narwhal (11.04) 64-bit.

Software Libraries and Applications

- Apache - v2.2.17
- Django - v1.4.0
- Graylog2 - v0.9.6
- Hachoir Python Library -v.1.3.3
- Logstash - v1.1.1
- MongoDB - v1.6.5

- MySQL - v5.1.63
- Numpy Python Library - v1.5.1
- Python -v2.7.1
- RabbitMQ - v2.8.5
- Sphinx -v0.9.9
- Webmin - v1.590

A.2 Monitoring Server

The monitoring server is used for the evaluation of the platform to monitor the primary server, as well as the client devices.

Operating System

Ubuntu Linux Natty Narwhal (11.04) 64-bit.

Software Libraries and Applications

- Apache - v2.2.17
- Munin - v1.4.5
- Python -v2.7.1

A.3 Client Application

The section documents the client application software libraries.

- Numpy Python Library -v1.6.1
- Pika Python Library -v0.9.5
- PyQt - v4.9.1
- Requests Python Library -v0.14.1

Appendix B

Configuration

This chapter provides the configuration settings for the services running on the primary server. The configurations provided are for Apache web server, Sphinx full text search server and Logstash log management.

B.1 Apache Web Server

```
# /etc/apache2/httpd.conf

# FOR DEVELOPMENT ONLY
#MaxRequestsPerChild 1

WSGIPythonPath /home/quiran/workspace/masters/src/server/
   jaxtv

<VirtualHost _default_:443>
    ServerName q.ml.sun.ac.za:443
    ServerAdmin qstorey@ml.sun.ac.za
    DocumentRoot /home/quiran/www

    # Turn on bandwidth limitation
    BandwidthModule On
    ForceBandWidthModule On
    # 512Kbps
    Bandwidth all 524288
    MinBandwidth all -1
    ErrorLog logs/q.ml.sun.ac.za-error_log
    TransferLog logs/q.ml.sun.ac.za-access_log
    SSLEngine on
    SSLCertificateFile /etc/apache2/ssl.crt/BettyServer.
       crt
    SSLCertificateKeyFile /etc/apache2/ssl.key/BettyServer
       .key

    WSGIScriptAlias / /home/quiran/workspace/masters/src/
       server/jaxtv/jaxtv_site/wsgi.py
    Alias /static/admin/ /usr/local/lib/python2.7/dist-
       packages/django/contrib/admin/static/admin/

    <Directory /home/quiran/workspace/masters/src/server/
       jaxtv/jaxtv_site>
        <Files wsgi.py>
        Order deny,allow
        Allow from all
        </Files>
    </Directory>

</VirtualHost>
```

B.2 Sphinx Full Text Search Server

```
#################################################
## data source definition
#################################################
source src_mysql
{
  # data source type. mandatory, no default value
  # known types are mysql, pgsql, mssql, xmlpipe, xmlpipe2
     , odbc
  type            = mysql
  #######################################################
  ## SQL settings (for 'mysql' and 'pgsql' types)
  #######################################################
  # some straightforward parameters for SQL source types
  sql_host          = localhost
  sql_user          = jaxtv
  sql_pass          = jaxtv
  sql_db            = jaxtv
  sql_port          = 3306  # optional, default is 3306
  # UNIX socket name
  # optional, default is empty (reuse client library
     defaults)
  # usually '/var/lib/mysql/mysql.sock' on Linux
  # usually '/tmp/mysql.sock' on FreeBSD
     sql_sock         = /var/run/mysqld/mysqld.sock
  # main document fetch query
  # mandatory, integer document ID field MUST be the first
      selected column
  sql_query         = \
        SELECT `table_ids`.`id`, `table_videos`.`name`
           FROM `table_videos` \
        INNER JOIN `table_ids` ON `table_videos`.`
           info_hash`=`table_ids`.`info_hash`
  # document info query, ONLY for CLI search (ie. testing
     and debugging)
  # optional, default is empty
  # must contain $id macro and must fetch the document by
     that id
  sql_query_info     = SELECT * FROM videos WHERE id=$id
}
#################################################
## index definition
#################################################
index index_jaxtv
{
  # document source(s) to index
  # multi-value, mandatory
  # document IDs must be globally unique across all
     sources
  source       = src_mysql
  # index files path and file name, without extension
  # mandatory, path must be writable, extensions will be
     auto-appended
  path       = /var/lib/sphinxsearch/data/index_jaxtv
  # document attribute values (docinfo) storage mode
  # optional, default is 'extern'
  # known values are 'none', 'extern' and 'inline'
  docinfo      = extern
  # a list of morphology preprocessors to apply
  # optional, default is empty
  #
  # builtin preprocessors are 'none', 'stem_en', 'stem_ru
     ', 'stem_enru',
  # 'soundex', and 'metaphone'; additional preprocessors
     available from
  # libstemmer are 'libstemmer_XXX', where XXX is
     algorithm code
  # (see libstemmer_c/libstemmer/modules.txt)
  #
  # morphology   = stem_en, stem_ru, soundex
  # morphology   = libstemmer_german
  # morphology   = libstemmer_sv
# morphology     = none
```

```
    morphology      = stem_en
  # charset encoding type
  # optional, default is 'sbcs'
  # known types are 'sbcs' (Single Byte CharSet) and 'utf
     -8'
  charset_type    = sbcs
}
#################################################
## indexer settings
#################################################
indexer
{
  # memory limit, in bytes, kiloytes (16384K) or megabytes
      (256M)
  # optional, default is 32M, max is 2047M, recommended is
      256M to 1024M
  mem_limit     = 32M

}
#################################################
## searchd settings
#################################################
searchd
{
  # hostname, port, or hostname:port, or /unix/socket/path
      to listen on
  # multi-value, multiple listen points are allowed
  # optional, default is 0.0.0.0:9312 (listen on all
     interfaces, port 9312)
  #
  # listen        = 127.0.0.1
  # listen        = 192.168.0.1:9312
  # listen        = 9312
  # listen        = /var/run/searchd.sock
    #listen                    = q.ml.sun.ac.za:9312
    listen                    = localhost:9312
  # log file, searchd run info is logged here
  # optional, default is 'searchd.log'
  log           = /var/log/sphinxsearch/searchd.log
  # query log file, all search queries are logged here
  # optional, default is empty (do not log queries)
  query_log      = /var/log/sphinxsearch/query.log
  # client read timeout, seconds
  # optional, default is 5
  read_timeout     = 5
  # request timeout, seconds
  # optional, default is 5 minutes
  client_timeout     = 300
  # PID file, searchd process ID file name
  # mandatory
  pid_file       = /var/run/searchd.pid
  # max amount of matches the daemon ever keeps in RAM,
     per-index
  # WARNING, THERE'S ALSO PER-QUERY LIMIT, SEE SetLimits()
      API CALL
  # default is 1000 (just like Google)
  max_matches      = 1000
  # seamless rotate, prevents rotate stalls if precaching
     huge datasets
  # optional, default is 1
  seamless_rotate    = 1
  # whether to forcibly preopen all indexes on startup
  # optional, default is 0 (do not preopen)
  preopen_indexes    = 0
  # whether to unlink .old index copies on succesful
     rotation.
  # optional, default is 1 (do unlink)
  unlink_old       = 1
  # MVA updates pool size
  # shared between all instances of searchd, disables attr
      flushes!
  # optional, default size is 1M
  mva_updates_pool   = 1M
}
```

B.3 Logstash

```
# Input
input {
    amqp {
        host => "127.0.0.1"
        name => "general_logs_queue" # Name of the queue
        exchange => "logstash_exchange"
        key => "general_logs_key"
        exclusive => false
        durable => true
        auto_delete => false
        type => "general-logging-input"
        format => "json"
    }
    file {
        type => "apache_error"
        path => [ "/home/quiran/workspace/masters/src/
           server/jaxtv/logs/error.log"]
    }
    file {
        type => "apache_log"
        path => [ "/home/quiran/workspace/masters/src/
           server/jaxtv/logs/jaxtv_django.log"]
    }
    file {
        type => "jaxtv_tracker"
        path => [ "/home/quiran/workspace/masters/src/
           server/jaxtv/logs/tracker.log"]
    }
    file {
        type => "encryption"
        path => [ "/home/quiran/workspace/masters/src/
           server/jaxtv/logs/encryption.log"]
    }
}

# Filters
# Output
output {
    elasticsearch {
        embedded => true
    }
    gelf {
        facility => "Clients"
        host => '127.0.0.1'
        type => "general-logging-input"
    }
    gelf {
        facility => "Apache␣Log"
        host => "127.0.0.1"
        type => "apache_log"
    }
    gelf {
        facility => "Apache␣Error␣Log"
        host => "127.0.0.1"
        type => "apache_error"
    }
    gelf {
        facility => "Encryption␣Log"
        host => "127.0.0.1"
        type => "encryption"
    }
    gelf {
        facility => "Jaxtv-Tracker␣Log"
        host => "127.0.0.1"
        type => "jaxtv_tracker"
    }
}
```