Scalable Media Transcoding for Cross Platform Streaming

Honours Dissertation

Andrew Beveridge: H00013703
MEng Software Engineering
research@andrewbeveridge.co.uk

Supervisor:
Peter King
P.J.B.King@hw.ac.uk

Second Reader:
Sandy Louchart
S.Louchart@hw.ac.uk

Declaration

I, Andrew Beveridge, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date:

Abstract

Video of all kinds is being consumed on a wider range of devices than ever before, and consumers expect a high quality experience on every device without being made to wait. This demand for quality and speed across such a variety of platforms creates demand for better ways of managing and serving video files. There are many challenges to overcome, and while current solutions have already solved these in a variety of ways, there is always room for improvement since existing solutions are not perfect. This research investigates current solutions in depth, presents a new solution centred around streaming transcoded video on demand, and evaluates the cost-effectiveness and user experience gain from this method.

Acknowledgements

This research would not have been possible without my personal support network.

I'd like to offer a few thanks to the mentors, colleagues, friends and family who helped me along the way: my supervisor Peter, for not giving up on me despite having very good cause; Hans, for helping me remember my passion for this topic in the first place; Leif, for asking the difficult questions which helped trigger problem-solving thoughts, and debating document structure at midnight; my parents, for trying to check up on me despite getting almost no response from me, and my fiancée Marta, for bringing me cups of tea, helping me focus and giving me space.

Abbreviations

AJAX Asynchronous JavaScript and XML
AVI Audio Video Interleave
ES Elementary Stream
IEC International Electrotechnical Commission
ISO International Organization for Standardization
MKV Matroska Multimedia Container
MP4 MPEG-4 Part 14
MPEG Moving Picture Experts Group
MSE Mean Squared Error
MUX, DEMUX Multiplex, Demultiplex
MVC Model, View, Controller
ORM Object Relational Mapper
PHP PHP: Hypertext Preprocessor
PSNR Peak Signal-to-Noise Ratio
SSIM Structural Similarity
SVC Scalable Video Coding
VCS Version Control System
VLC VLC Media Player

Glossary

Asynchronous JavaScript and XML

A group of web development techniques used to create asynchronous communication between client-side and server-side code without interfering with the display and behavior of the existing page

Audio Video Interleave

A multimedia container format introduced by Microsoft in November 1992 as part of its Video for Windows technology

Codec

A device or program that compresses data to enable faster transmission and decompresses received data

Elementary Stream

The output of an audio or video encoder, containing only one kind of data, e.g. audio, video or closed caption. Often referred to as "elementary", "data", "audio", or "video" bitstreams or streams

Generation loss

The loss of quality between subsequent copies or transcodes of data

International Electrotechnical Commission

A non-profit, non-governmental international standards organization that prepares and publishes International Standards for all electrical, electronic and related technologies

International Organization for Standardization

An international standard-setting body composed of representatives from various national standards organizations

Linux

A generic term referring to the family of Unix-like computer operating systems that use the Linux kernel

Matroska Multimedia Container

An open standard free video container format

Mean Squared Error

A quantifier of the difference between values implied by an estimator and the true values of the quantity being estimated

Model, View, Controller

A software pattern which divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user

Moving Picture Experts Group

A working group of authorities that was formed by ISO and IEC to set standards for audio and video compression and transmission

MPEG-4 Part 14

A video file format specification created in 2003 by MPEG as an extension of the ISO base media file format

Multiplex

The combination of more than one stream of data into one signal. Also *Demultiplex*, the reverse of this process to produce two or more discrete streams

Object Relational Mapper

A framework which abstracts away from the manual conversion of data between the type system in a relational database and an object-oriented programming language

Peak Signal-to-Noise Ratio

An engineering term for the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation

PHP: Hypertext Preprocessor

A server-side scripting language designed for web development but also used as a general-purpose programming language

Scalable Video Coding

A video compression standard which uses a high quality stream with multiple subset streams to allow for quality variation with varying bandwidth

Stream

Data which is read or written in a continuous flow

Structural Similarity

A full reference metric for measuring the similarity between two images

Transcode

The direct analog-to-analog or digital-to-digital conversion of a file from one encoding to another, such as with video or audio files

Version Control System

A tool for managing a collection of files which provides three important capabilities: *reversibility*, *concurrency*, and *annotation*.

VLC Media Player

Originally standing for *VideoLan Client*, VLC is a portable, free and open-source, cross-platform media player and streaming media server written by the VideoLAN project

Contents

Abbreviations **i**

Glossary **ii**

1 Introduction **1**
1.1 Problem Overview . . . 1
1.2 Aims & Objectives . . . 1
1.3 Document Structure . . . 2

2 Background **3**
2.1 Digital Video . . . 3
2.1.1 File Structure . . . 3
2.1.2 Container Formats . . . 4
2.1.3 Compression Formats . . . 6
2.2 Related Work . . . 8
2.2.1 Distributed Transcoding . . . 8
2.2.2 Computing Power vs Storage Space . . . 9
2.2.3 Scalable Video Coding . . . 9
2.3 Previous Systems . . . 11
2.3.1 On Demand Transcoding . . . 11

3 Requirements **12**
3.1 Testing Platform . . . 12
3.1.1 Overview . . . 12
3.1.2 Functional Requirements . . . 12
3.1.3 Non-Functional Requirements . . . 13
3.2 Transcoding System . . . 13
3.2.1 Overview . . . 13
3.2.2 Functional Requirements . . . 14
3.2.3 Non-Functional Requirements . . . 14

4 Design **15**
4.1 Testing Platform . . . 15
4.1.1 Potential Challenges . . . 16
4.2 Transcoding System . . . 17
4.2.1 Potential Challenges . . . 17

5 Implementation **19**
5.1 Tools Used . . . 19
5.1.1 Git . . . 19
5.1.2 LAMP Stack . . . 19
5.1.3 Symfony . . . 20
5.1.4 jQuery . . . 21
5.1.5 FlowPlayer . . . 21
5.1.6 FFmpeg . . . 22
5.2 Process . . . 22
5.2.1 Transcoding System . . . 27

6 Testing and Evaluation **33**
6.1 Bandwidth Measurement . . . 33
6.2 Transcoded Video Quality . . . 34

7 Discussion **37**
7.1 Limitations 37
7.2 Conclusion 37
7.3 Future Work 38

8 References **39**

Appendix A Client Bandwidth Measurement **41**
A.1 Server Side 41
A.2 Client Side 41

Appendix B Main Controller **43**
B.1 Transcode Action 43

1 Introduction

1.1 Problem Overview

Video is being created, transported and consumed at a higher rate than ever, with consumers demanding ever higher qualities and less time spent waiting. The range of devices capable of playing video increases every year, and users expect the same experience on any device.

The most widespread method of solving this problem involves transcoding source files into multiple formats in advance, and simply serving the most suitable file for each platform from disk when a user makes a request. However transcoding video is processor-intensive work, so producing all these output formats introduces a significant delay between the time when the source file becomes available and when it is ready to watch. This delay can range from minutes to days depending on the service providers infrastructure, which is frustrating for content producers.

On top of this inconvenience, a large portion of videos stored by large streaming media services are not very popular (Liu et al., 2012), but are still transcoded and stored indefinitely, so they are available to serve in case the video is requested by a specific combination of platform and quality setting. This causes a large amount of storage space to be wasted by transcoded files which are never, or rarely accessed (Kathpal et al., 2012).

This project will investigate various methods of solving this problem, try to propose a new technique for serving high quality video to many platforms and evaluate the implementation performance.

1.2 Aims & Objectives

The aim of this research project is to investigate and evaluate methods of streaming video, to find a method which allows for a significant improvement in user experience or a decreased cost for service providers.

The primary objectives are:

- Provide future developers with valuable information regarding the effectiveness and scalability of various video transcoding technologies
- Develop a system to test and evaluate various methods of streaming high quality video to multiple platforms on demand
- Investigate whether streaming transcoded video on demand is possible, and measure scal-

ability to assess whether it is viable for use in real world scenarios

- Calculate economic cost of on demand transcoding and compare with current methods of serving video to multiple platforms

1.3 Document Structure

To effectively communicate the work performed and contribution of this research in the area of video streaming, this document is structured as follows.

- Chapter 1 briefly explains the problem which forms the basis of this work, outlines the project objectives and describes the structure of this document
- Chapter 2 explains the basic concepts and background information about video required for comprehension of this document, reviews previous academic and commercial work in the field of video compression, transcoding and streaming.
- Chapter 3 details the design process which led to the final solution, etc..

2 Background

2.1 Digital Video

2.1.1 File Structure

To store video data as a file on a digital storage medium such as a hard disk or flash drive in a useful way, a specification for the layout and content of this data must be defined formally. This formal specification of stored video data is the definition of the general term *video format*. To complicate matters, the data structure and various elements of metadata are collectively named the *container format*, whereas the actual video data inside this file is a *video stream*, which is encoded using a *video codec* to a *video compression format*. Other data streams such as audio, subtitles etc may also exist inside the container file, structured according to the specification of the container format.

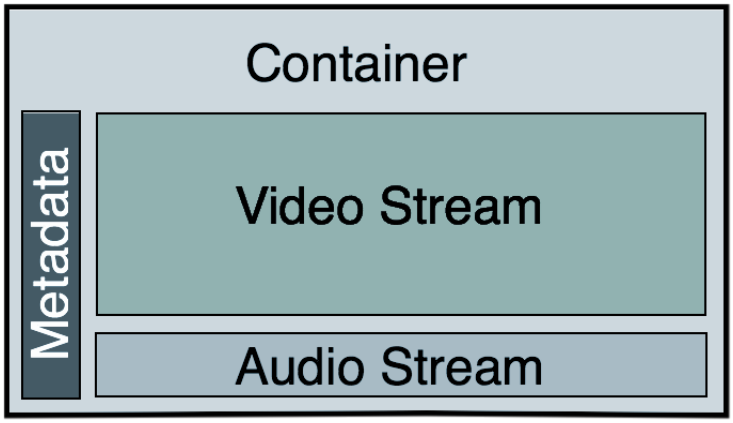

Figure 1: Structure of a video file.

Each stream in a video file is encoded using a *codec*. A codec is a program which can both *encode* and *decode* a stream of data, usually for the purpose of compressing it to take up less space on disk. This compression can be done in many different ways with a huge variety of parameters. A codec can be *lossless*, where the output from decoding an encoded file is identical to the uncompressed data, or *lossy*, where the compression algorithm tries to remove as much data as possible without lessening the perceived quality too much. As clients usually have limited bandwidth, all video formats used on the web are lossy. Finding a good set of parameters to keep the balance between smaller file size and loss of perceived quality at an acceptable place is crucial.

2.1.2 Container Formats

There are a wide variety of container formats available, each of which is suited to some purposes more than others. This section provides a brief description of some of the most popular formats, and details any relevance to this project.

Name	**Extension**	**Released**	**Supported Compression Formats**
Audio Video Interleave	AVI	1992	Any
Matroska	MKV	2002	Any
MPEG-4 Part 14	MP4	2001	H.263, MPEG-4 ASP, H.264
Flash Video	FLV	2003	H.263, VP6, H.264
WebM	WEBM	2010	VP8, VP9

Table 1: Popular video container formats

Audio Video Interleave was invented by Microsoft in a time when computers were just starting to be capable of playing video. It was the most widely used container format for over a decade, but by the mid 2000s it was declining in popularity rapidly in favour of formats with more modern features. It does not officially support many things which are now considered basic, such as embedded metadata or subtitles. It does not even officially support most of the modern compression formats in use today, but because of its simplicity almost any combination of video and audio streams can be multiplexed into an AVI file and play in any video player which supports the compression formats used. However, as it is not supported natively by any web browser and it is now quite out of date, it is mentioned in this research purely for reference and comparison.

Matroska Multimedia Container is a container format developed as an open source project by the Matroska Development Team. The primary goal of the project is to "Create and document a modern, flexible and cross-platform Audio/Video container format, in combination with an open codec API to form a free and open media framework". As a format for storing and playing video streams, it is very useful in many different situations as it can contain almost any combination of compression formats. For this research it is very useful for testing and comparison, as it supports live playback and seeking, allowing lots of compression formats to be evaluated for streaming purposes which would otherwise be difficult to test. However, as no browsers have native support for the format, an external application such as VLC Media Player must be used to play files in this format.

MPEG-4 Part 14 is the most ubiquitous container format used for video today. With native playback support in over 80% of web-enabled devices, it is used by almost all major video distributors, and it is the only format supported natively by Apple devices. Unfortunately, due to the way the container format stores structured metadata, it is not suited to on-demand streaming without the entire file being encoded in advance.

WebM is a comparatively new container format, created by a collaboration between Google, Mozilla and Matroska as a limited subset of the Matroska container format with only a few supported codecs to make web support easier to implement. Supported by all recent versions of the Google Chrome, Mozilla Firefox and Opera web browsers, it is a significant contender in the formats available for web video streaming.

Flash Video is a proprietary format created by Adobe to play videos on the web from the Flash Player browser plugin. For many years this was one of very few container formats which were used to stream video online, however as Flash is now declining in popularity it is becoming less useful. As some browsers which do not support native video playback may still have the Flash plugin installed, it is still useful as a fall-back option for some situations.

2.1.3 Compression Formats

The number of compression formats available is even larger than the number of container formats, however for the purpose of this research, only the two most widely used, modern formats are investigated. These are H.264, most commonly found in the MP4 container file and VP8, created exclusively for the WebM container format. These two formats collectively provide support for all major platforms, whereas native support for almost all other compression formats is very limited on all platforms. The following platform support for these two formats is shown below (Q-Success, 2014b) .

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
							2.1		
							2.2		
					3.2		2.3		
					4.0-4.1		3.0		
8.0		31.0			4.2-4.3		4.0		
9.0		32.0			5.0-5.1		4.1		
10.0	27.0	33.0			6.0-6.1		4.2-4.3	7.0	
11.0	28.0	34.0	7.0	20.0	7.0	5.0-7.0	4.4	10.0	10.0
	29.0	35.0		21.0					

Support:	61.42%
Partial support:	19.74%
Total:	81.16%

Figure 2: Platform support for MP4 container format with video in H.264 compression format.

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
							2.1		
							2.2		
					3.2		2.3		
					4.0-4.1		3.0		
8.0		31.0			4.2-4.3		4.0		
9.0		32.0			5.0-5.1		4.1		
10.0	27.0	33.0			6.0-6.1		4.2-4.3	7.0	
11.0	28.0	34.0	7.0	20.0	7.0	5.0-7.0	4.4	10.0	10.0
	29.0	35.0		21.0					

Support:	38.86%
Partial support:	21.12%
Total:	59.98%

Figure 3: Platform support for WEBM container format with video in VP8 compression format.

The process of converting a file from one combination of container and compression formats to another is called transcoding. A simplified diagram of the purpose of a transcoder is shown below.

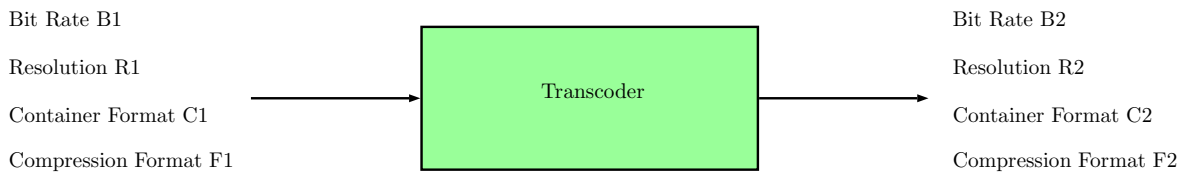

Figure 4: System diagram of a black box transcoder.

2.2 Related Work

Much work has already been done in the field of video transcoding. At a low level, there is constant research into new compression formats capable of ever higher compression ratios. There is commercially-sponsored research into more efficient techniques for handling, processing and serving video to consumers for large scale services. There is extended analysis of existing systems and data produced from the daily running of such services. In this section, a few of these existing research topics are addressed and a summary of recent academic work in these fields is offered.

2.2.1 Distributed Transcoding

Many large consumer-facing services serve videos daily to millions of viewers. To make this possible, these services transcode source videos to the ideal format for multiple platforms after the source content is added. As video transcoding is very processor-intensive, finding efficient ways of scaling up this transcode requirement is crucial. As such, several research projects have already tackled and, for the most part, solved this problem. A PhD project by Deneke (2011) implements a solution by distributing the transcoding process over a cluster of processing units using the MPI framework. Another study by Sanson et al. (2012) provides a similar distributed transcoding system, with the addition of a new parallel processing framework to reduce processing times even further. These projects are very much relevant and useful to this research, as the problem being tackled is similar and the technical implementations offered by both are well thought out. However, while prior research into distributed transcoding solves the problem of transcoding many files to many formats simultaneously, this is only part of a larger problem of providing video to consumers in the ideal quality; these systems assume that they will be told which files to transcode and into which output formats by another part of the program. This research project implements a new, somewhat simple implementation of distributed transcoding, but could easily be used with one of these other transcoding architectures. The core concepts in this research tackle a slightly different problem, attempting to reduce the amount of transcoding by only transcoding files when needed rather than automatically transcoding everything into every format.

2.2.2 Computing Power vs Storage Space

The common method of transcoding videos in advance and storing files in multiple formats is evaluated and compared with transcoding video on demand for each format in this paper by Kathpal et al. (2012). The conclusion that "trading storage with compute can be cost effective" is encouraging, as this research project is centred around trying to balance computation power with storage capacity. They define a metric called "Elimination Metric" based on the cost of computation time, data transfer and data storage over time. Using this value a content management system can decided whether a particular transcode should continue to be stored, or eliminated and left to be transcoded again on demand when requested. However, the paper did not actually implement a system to actually manage, transcode and serve videos; instead the paper focuses on manually measuring results from transcode processes executed by the researchers and providing insights into the cost-effectiveness of transcoding on demand vs. storing all of the transcoded versions of files. This is very useful information, but crucially as part of the "Future work" section of this paper it mentions user perceived latency with eliminated transcodes as a problem. This problem is significant, as transcoding a source video to a desired format can easily take as long as, or longer than, the duration of the video. Latency to this extreme is unacceptable, as user engagement is crucial when serving videos to consumers, so expecting a user to wait 3 minutes before watching a 3 minute video is useless (Krishnan and Sitaraman, 2012). This research project attempts to tackle precisely this problem, to make on-demand streaming of a file possible while it is still being transcoded.

2.2.3 Scalable Video Coding

provides a standardized extension to the H.264/AVC compression format to add a Video Coding Layer (VCL) and a Network Abstraction Layer (NAL), with the objective of enabling the encoding of a high-quality video bit stream that contains multiple subset bit streams. (Schwarz et al., 2007) These subset streams cumulatively allow for very high quality reproduction of the source file, but can be truncated at any arbitrary rate and still be playable at a lower quality. The base layer provides basic quality, successive layers refine the quality incrementally, allowing one file to be used regardless of bandwidth restraint with graceful degradation. A proof of concept system using custom playback applications on desktop and mobile devices was implemented by Goh et al. (2011), however this requirement of custom players limits the useful scope dramatically for use on any public website where consumers do not have any player capable of playing SVC

content installed. Huang et al. (2011) implements a service to transcode a high quality video stream to on demand (dubbed *CloudStream*); however it does not mention what device was expected to play such a stream. While seems like the most elegant solution to the general problem of serving high quality video to many different devices with different bandwidth restraints, it is still mostly theoretical; the technology is not yet very mature and very few players exist which can actually decode and play an SVC stream. As such, this research focuses on solving the problem using technologies which are actually usably in the mainstream at this moment in time.

2.3 Previous Systems

2.3.1 On Demand Transcoding

Cloud Transcoder (Li et al., 2012) is an API which allows a content provider to abstract away from transcoding entirely by sending mobile video consumers a URL which uses their service as a proxy to transcode a source file on the fly to serve to whichever device requested it. *Amazon Elastic Transcoder* is another transcoding API, provided as part of Amazon Web Services, which offers media transcoding in the cloud. It is described as a "highly scalable, easy to use and cost effective way for developers and businesses to transcode media files from their source format into versions that will playback on devices like smartphones, tablets and PCs." With presets for popular output formats, a content provider does not need to do their own research of make decisions about which settings work best on particular devices, rather they simply specify the device to transcode for when making an API request and the output file will be the ideal format for that platform. These systems are relevant and useful to this research in that they have essentially already solved the problem of scalable on-demand transcoding for multiple platforms. However, as mentioned earlier this is only part of a larger problem of providing video to consumers in the ideal quality; the point of this research is to tackle a slightly different problem, reducing the amount of transcoding which needs to be performed in the first place. Also, while both of these services are very useful when operating under the traditional assumption that a source video must be transcoded to many different formats and then stored before being usable, they are less useful for on-demand streaming, as they will not provide access to the output file until the transcode process is complete.

3 Requirements

The core requirements of this project are: provide insight into whether streaming transcoded video on demand is possible, and measure the scalability of this functionality if it is actually possible. To achieve these objectives it is necessary to develop a testing platform, in order to evaluate different methods of streaming high quality video to multiple platforms on demand. As such the requirements of this project are divided into two subsections, a testing platform which enables the research into different techniques, and the actual transcoding system.

3.1 Testing Platform

3.1.1 Overview

The testing platform should be a web-based system which enables development, testing and evaluation of video streaming functionality. It should be simple and functional, allowing a developer or researcher to change parameters passed to a transcoding system and view the results as a video stream in a web browser.

3.1.2 Functional Requirements

- Provide a web interface to launch transcode processes and view results in a video player
- Allow selection of video from one or more test file(s) to transcode
- Allow for selection of static file to test playback without transcoding
- Allow entering a URL to a video stream elsewhere on the internet to test playback
- Provide methods for manually altering the value of transcode system parameters:
    1. Container and compression format of transcoded video stream
    2. Bit rate of transcoded video stream
    3. Time position in video file to start transcoding from
- Allow compatibility testing of various video formats from any web browser on any device
- Allow testing of transcoding system effectiveness from any web browser on any device

3.1.3 Non-Functional Requirements

- Use only software libraries and tools licensed under an open source license
- Have very little computational overhead so as to not interfere with test results
- Have thorough documentation in the form of commented code to ensure maintainability

3.2 Transcoding System

3.2.1 Overview

The basic function of the transcoding system is to enable streaming of a high quality source file in one format to a client which may not otherwise support playing that file directly.

Firstly, an assumption is made that even if the client system could play the file directly using a dedicated video player such as VLC Media Player, only the capabilities of the web browser they connected with should be taken into account. This assumption is centred around the fact that video consumption often commands low user engagement (Krishnan and Sitaraman, 2012). As such, if the user is required to install a new application on their system, or even open a separate application which they already have installed to facilitate playback of the video stream, they may lose interest and leave rather than do so.

A second assumption is made that the video stream must be playable without needing to wait for a download process to occur first; we can see from previously mentioned research that the percentage of abandoned views and startup delay are positively correlated when startup delay is above 3 seconds, (Krishnan and Sitaraman, 2012) so we aim for startup delay to be no more than 3 seconds. Three seconds is not long enough to download the entire source video unless the source video is extremely short and the user has a fast internet connection, so in almost all cases downloading the video before playing is not a feasible option.

With these assumptions, the client browser may be unable to play the source file directly due to two specific reasons: insufficient bandwidth, which would force the client to wait for more than 3 seconds; or the source file may be in a format which the client browser does not support.

Making the video playable with a delay of less than 3 seconds in either of these cases is the primary purpose of the transcoding system, solving the latency issue described in previous research in this field by Kathpal et al. (2012).

3.2.2 Functional Requirements

- Measure client available bandwidth
- Identify client supported container and compression formats
- Calculate the optimal format and bit rate to stream video to the client
- Start transcode process to create optimal transcode file from selected source file
- Stream this transcode file to the client immediately
- Provide seeking capability by starting a new transcode process with specified start point

3.2.3 Non-Functional Requirements

- Use only software libraries and tools licensed under an open source license
- Have thorough documentation in the form of commented code to ensure maintainability
- Be self-contained and portable, allowing simple use in other video streaming systems

4 Design

4.1 Testing Platform

As the testing platform is only intended for use by a developer or researcher, the graphical design and user interface layout is not of great importance. However in order to meet the functional requirements outlined earlier in section 3, the following design is proposed.

On initial load of the test system front page, simple options to select a test file should be presented. The test files should be a selection of open source short films, stored on the server unmodified, in the highest quality format they are available in. These particular short films provide diversity in content type and activity level which is crucial for fair testing of compression format performance with different source files, and add no licensing complications.

There should be an input to enter a relative file path to a source file on the server to load into the transcode system and play; this simplifies testing the system with files other than the previously mentioned open source movies. Finally, this front page should have an input to specify a full URL to a publicly accessible video file elsewhere on the internet - when provided, this video stream should be played directly rather than transcoded. This again makes testing specific files easier.

After selecting a test file to transcode, a new page should appear which shows a video player to play the transcoded stream, and some basic controls to manually modify the transcoding parameters and update the stream. Upon page load, it may be necessary for the user to wait for up to 3 seconds for the video to load. During this waiting period, a moving loading image should be displayed to show the user the video is loading. By default, the parameters of the transcode process should be calculated and detected automatically by the transcode system, and displayed in the control fields so the user can see what values the system calculated as optimal.
Control fields should be created to allow modification of the following parameters:

- Video start position (in seconds)
- Transcode bit rate (in kilobytes per second)
- Transcode container format
- Source video file path

Upon modification, these values should immediately update - this should cause the video which is currently playing to change to a new transcode, produced according to the new pa-

rameters. This will allow for subjective testing of how well the transcoding system works and allow simulation of what the video quality would look like for a user accessing the system from platforms which support different formats or have less available bandwidth.

The video player itself should take up most of the screen, have basic play, pause, volume and time bar controls and display video duration in minutes and seconds, allowing for seeking to a new playback location by clicking a point on the time bar.

4.1.1 Potential Challenges

The basic testing platform should be relatively straightforward to implement as it is primarily just a video player which plays a specified video URL. However, the following potential challenges have been identified as likely causes of extra work or complication.

- Quickly manipulating the video stream which is being played or replacing it with another may be somewhat complex, depending on what method of playing video in the browser is used; a player with an extensive API may be easy to work with, but given the requirement that all software used must use an open-source license, it may be difficult to find a player with such an extensive API.

- The design specifies that the player should display the video duration. However as the video which is being played is still being transcoded, the metadata which contains information such as the video duration does not yet exist. Finding a solution to this problem may prove difficult due to the nature of how video metadata is stored in common container formats.

- As the ability to seek is required, yet the aforementioned metadata which contains the file duration does not exist, this may also cause seeking using the time bar to not function without additional work.

4.2 Transcoding System

The transcoding system has a fairly simple purpose; take a request to play a high quality source file and create a new file which can be immediately played by the client web browser with whatever limitations that may impose. However this is technically complex, with several challenges to overcome. The design of the system this project implements is documented below

First, the system should measure the bandwidth available to the client browser. This gives the system an estimate of what target bit rate the transcode should be started with, which essentially allows the system to create a bespoke video stream on demand which maximises the client internet connection. This should allow for a higher quality video viewing experience than if the more traditional technique of using a two or three preset video quality configurations and allowing the user to select one is used. The bandwidth measurement itself should be calculated by transferring a small file to the client and recording the time taken to transfer.

Next, the video playback capabilities of the client should be assessed; identification of video container and compression formats supported should be performed, possibly using an existing client-side library to do so. This allows the transcoder to output a file which is not only playable by the client, but in the case that multiple formats are supported, the most efficient compression format is used.

These details, including which file has been requested should be used to start a new transcode process to create the desired output file. The actual transcode process should run in the background on the server, but be kept track of to enable detection of when it finishes, allow ending the process prematurely if the client leaves or decides to play another video, etc. Once the transcode process has been started, the output file created by it should be read and streamed to the client to be played immediately.

If a seek request is made, any existing transcode processes for this client should be immediately ended and a new one created with the updated start position parameter, then streamed to the client as before. This technique should create the illusion of seeking within a file, despite the file not actually existing yet.

4.2.1 Potential Challenges

This technique for streaming video on demand while the transcoding process is still running does not seem to have been attempted in prior academic research. However there is good reason for this; it is technically challenging to implement successfully, for reasons explained below:

- As previously mentioned, video metadata is stored in different ways depending on which container format is used. However, many of the popular formats store metadata as one contiguous section of the file which is written after the file has been encoded. The technical reason for this is obvious; much of the metadata is data about the actual video stream itself - byte offsets to locate keyframes, length of each stream in bytes, various other bits of information which are only known once the video stream is complete. As such, many video players will either not play an incomplete file at all, or will play for a second or two then stop. Simply getting a file to play in any web browser while it is incomplete will be a significant challenge.

- Transcoding is a computationally difficult process. To transcode a video to an output file with a fairly low target bit rate whilst maintaining subjectively acceptable perceived quality requires a lot of processing power. However, to be able to stream a file to a client while it is still transcoding, the transcode process must be happening at a rate which is faster than the playback rate, usually 30 frames per second. As such there is a trade-off to be made between output quality and encoding speed, which is limited by a minimum encoding speed of 30 FPS. Depending on the compression format and codec used, it may be difficult to achieve an encoding rate which is consistently above 30 FPS without a large quality sacrifice.

- The range of devices which this project is aiming to support is huge: from phones, to tablets, to laptops and desktops with a variety of operating systems, there is a wide range of supported container and compression formats. However, while some platforms may support a variety of very flexible formats, other platforms may only support a single specific format which is more difficult to achieve on-demand streaming of an incomplete file. Achieving the goal of cross-platform playback may prove to be problematic.

5 Implementation

In the requirements and design sections of this research project, the proposed software development was divided into two discrete systems; a testing platform and a transcoding system. However upon development start it became apparent that separation of concerns to the point where the systems are clearly distinct would require a larger amount of time than was deemed necessary for completion of the primary research purpose. As such, this section describes the implementation of both the testing platform and transcoding system as one development process, as the resulting code is a single web application in which the transcoding functionality is a subset of the greater testing platform functionality.

5.1 Tools Used

Before detailing the method of this implementation, a description of the significant technologies chosen and explanation of each choice is provided in the following subsections.

5.1.1 Git

Git is a Version Control System (VCS); it records changes to a file or set of files over time to allow recalling or reverting to specific versions later on. This is crucial when developing a testing system as sometimes an attempted way of implementing a specific component doesn't work out and it is useful to be able to revert to a prior technique which worked. Git was chosen over other VCSs for this system because it is currently more popular with open source projects and has a greater collection of free services supporting it. All of the code written as part of this implementation was committed and pushed to a repository hosted by the GitHub online Git repository service.

5.1.2 LAMP Stack

LAMP as an acronym in the general context of open source software refers to the bundle of software which enables rich web applications to be built and served. Most traditionally meaning Linux, Apache, PHP and MySQL, with PHP interchangeable with Perl or Python and MySQL interchangeable with MariaDB or MongoDB.

In the context of this research it refers to Linux, Apache, MariaDB and PHP, which were the core software tools this implementation was built on top of.

Linux is the operating system running on the server which the system runs on and all testing was performed on. More specifically, the distribution of Linux chosen was Arch Linux, as it has a general preference for "bleeding edge" software which is of significance in a field such as video encoding where there is a very quick release cycle with high development activity.

Apache is the web server which processes the HTTP requests made by the client browser and serves the web pages which make up the testing platform. It is very mature software, with a large number of modules and configuration options, and is the most used web server (Q-Success, 2014a). Alternative servers choices are often made for higher performance (nginx, lighthttpd), for use in a non-free system (IIS) or for embedded systems, but Apache was sufficient for the limited requirements of this implementation.

MySQL is the most widely adopted open-source relational database used for web applications. It has proven to be a flexible, powerful alternative to proprietary databases and has become central to the architecture of the Internet (SkySQL Corporation, 2014b). MariaDB is an open-source, community-developed fork of MySQL, focused on maintaining direct replacement compatibility with MySQL (SkySQL Corporation, 2014a). For this project it was used primarily because it is the default implementation of MySQL used in Arch Linux.

PHP: Hypertext Preprocessor (PHP) is a server-side scripting language, originally created in 1995 by a single developer as a personal project, now produced as an open source project by The PHP Group (The PHP Group, 2014). Once intended to be embedded directly into an HTML source document to add server-side functionality to a static HTML web page, it has evolved to be a much more fully fledged programming language, offering most of the functionality expected in modern general-purpose languages. Over the past decade it has also become by far the most widely used server-side programming language on the web. (Q-Success, 2014b) This widespread usage and extensive functionality has encouraged the development of a vibrant community of open-source development centred around PHP as the core language. A wide range of extensively developed frameworks exist to make building a web application on top of PHP simpler, faster and more robust.

5.1.3 Symfony

Symfony is an open-source web application framework written in PHP, to simplify building web applications using the Model, View, Controller (MVC) software pattern (Potencier, 2014). The advantages of building upon a framework rather than starting from scratch when developing a

software project are numerous, but the most obvious is to prevent having to reinvent the wheel; when building a rich web application it can be easy to waste a lot of time developing features which have been developed previously by other people many times, whereas such features are usually built in to an application framework. High quality frameworks also add significant maintainability and upgradeability to a project, due the the underlying structure. Using the MVC pattern also increases separation of concerns, further enhancing maintainability and increasing reusability. There are many PHP frameworks available, each with its own advantages and disadvantages. Symfony was chosen for this project primarily because of the *Process* component, which makes the execution of other server-side applications (in this case, the actual transcode process) much more robust.

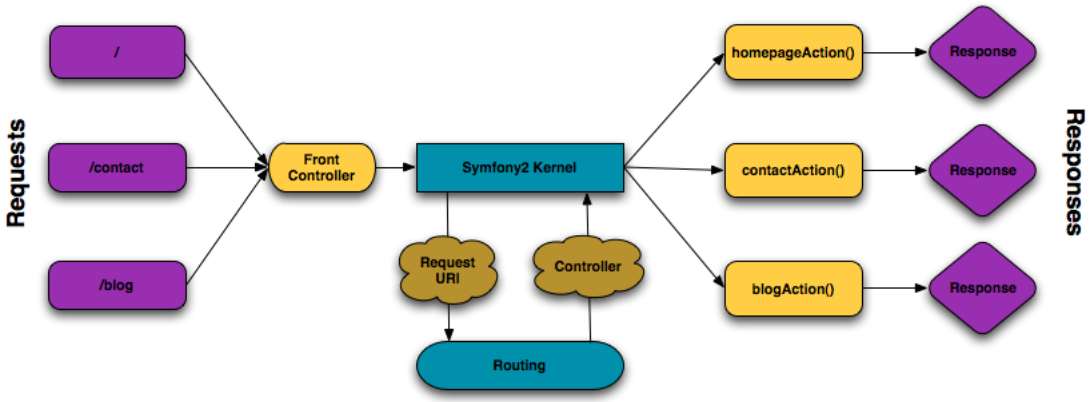

Figure 5: Architecture of a HTTP request and response in Symfony 2.

5.1.4 jQuery

jQuery is a cross-platform, widely used, open-source JavaScript library designed to simplify and add functionality to traditional client-side scripting (jQuery Foundation, 2014). Writing any significant client-side functionality in native JavaScript code without using a library such as jQuery involves considerably more work than with, so the majority of rich web applications use at least one library. jQuery specifically was chosen for this project due to well-documented integration with Symfony.

5.1.5 FlowPlayer

FlowPlayer is an open-source HTML5 video player, built on top of jQuery with a Flash fallback for browsers which do not support HTML5 video playback (Flowplayer, Ltd, 2014). With

an extensive API, it was chosen for this project due to its extensive API, after finding other video players (such as *JW Player*) lacking in specific API capabilities necessary for functionality required for this project.

5.1.6 FFmpeg

FFmpeg is the umbrella term for the open-source software project that produces several widely used libraries and programs for handling multimedia data (The FFmpeg Team, 2014b). The most widely known components of the FFmpeg project are:

- libavcodec, a library containing all the FFmpeg audio/video encoders and decoders
- libavformat, a library containing demuxers and muxers for audio/video container formats
- ffmpeg, a command-line tool to transcode audio and video formats

The important part of the FFmpeg project for this research is the *ffmpeg* command-line transcoding tool. When compiled with support for codecs which are restricted by software patents, it is a highly functional piece of software capable of doing a direct transcode of a source file in almost any format to an output file of almost any specified format, with built-in support for over 100 codecs (The FFmpeg Team, 2014a). An output file can be created with any arbitrary combination of Elementary Streams inside a container file, provided a Mux, Demuxer is available which can Mux, Demux the specified streams into the specified container. This allows for creation of "non-standard" output files; files which contain Elementary Streams which do not match the commonly expected compression formats for the container format. This flexibility is the main reason the ffmpeg tool was chosen to provide the core transcoding functionality for this implementation, as the ability to test such files on various platforms was desired.

5.2 Process

The first step of implementation involved setting up a virtual machine with the software stack described in the previous section. Once the web server was configured and accessible to the web, a new Symfony project was created using Composer, a PHP dependency manager. The next step was to create a new bundle named "TranscodingExperimentsBundle" in Symfony to contain all of the project code. A bundle is simply a directory with a well-defined structure which can contain almost anything, for example classes, controllers and web resources. A database called "transcoding" was created and details entered into the bundle configuration files to enable the

application to connect to the database and store objects. As version 2 of the Symfony framework uses the Doctrine Object Relational Mapper (ORM), writing data to the database is abstracted out as creating an instance of an entity, setting values and calling persist() to save the resulting object.

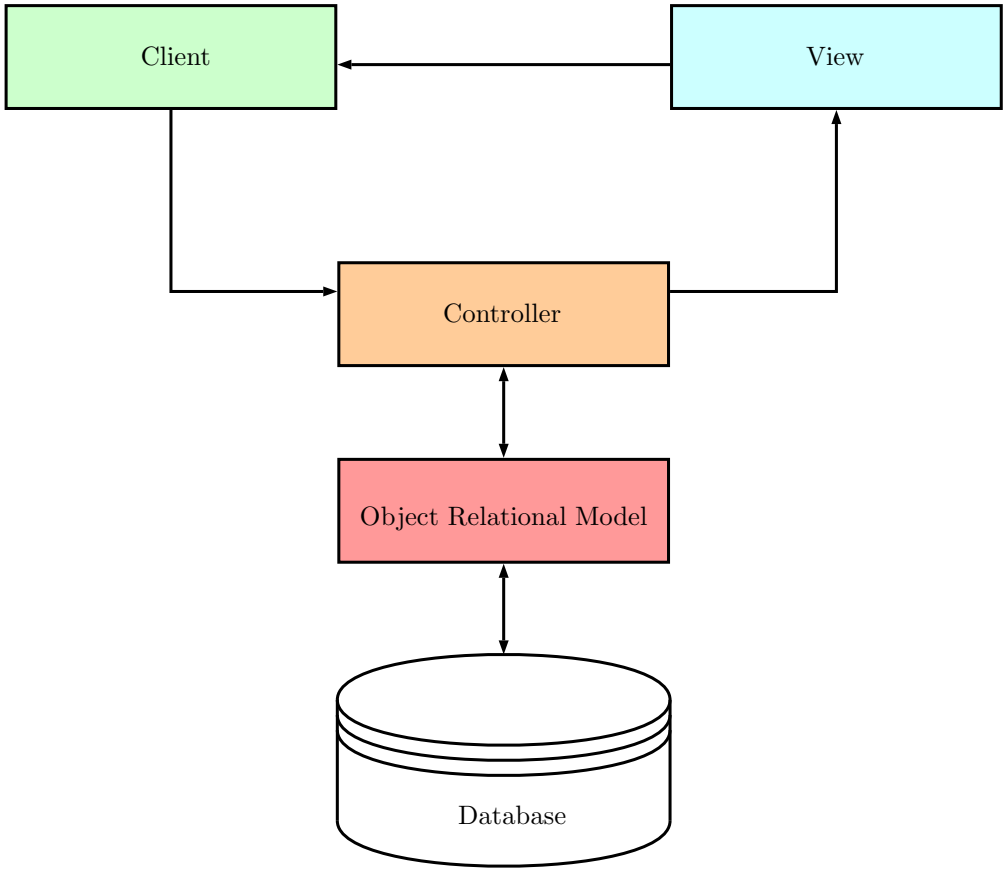

Figure 6: Structure of a web application using the MVC architecture with an ORM.

With the initial server and framework set up, the Symfony console was used to generate ORM entities for two objects, "Client" and "TranscodeProcess". The Client entity is trivially basic, with only one field: "ip", to store the remote IP address of a client. This serves the purpose of allowing for very basic tracking of a multiple clients, allowing the system to record which transcode process is running for a specific client. The TranscodeProcess entity is more significant, with fields to store every piece of information about a transcode process:

- clientID, the association with the client which requested this transcode
- status, simple state identifier (*running*, *ended*, *killed*)
- fullCommand, the full ffmpeg command which was executed with absolute paths

- processPID, the Linux process ID of the ffmpeg transcode process
- targetBitrate, the bitrate (in KB/s) of the transcoded output file
- inputFilepath, the absolute file path to the source file
- outputFilepath, the absolute file path to the transcoded output file
- duration, the duration (in seconds) of the source file

With entities created, the primary functionality of the testing platform was developed. Two routes were created, one for the front page and one for the video playback page. These routes both point to actions in the controller "DefaultController"; these actions are named "indexAction" and "loadFileAction" respectively.

indexAction is very basic, simply loading the front page view, which contains HTML and a small amount of inline CSS to apply basic functional styling; the front page contains a link to '/loadFile' for each of the test files with the file path as a URL parameter, and two small HTML forms to allow manually entering a file path or loading an external video URL to test.

Figure 7: Front page of Testing Platform.

loadFileAction implements the initial, static client side video playback/testing page itself, passing the URL parameter to the view to be used as a JavaScript variable. This view was initially fairly simple, containing a small amount of inline CSS, an empty div block named 'player' and some JavaScript to create the FlowPlayer instance and load the URL to a static video file on the server into the player. Once that was set up and working, the testing controls were created,

which consist of 5 named text input fields according to the requirements specification and one button named 'Clear' which clears any currently playing video and removes the player instance completely. At this point, the only inputs which actually functioned were the "File Path" and "URL" buttons, for which a jQuery event handler was assigned to capture form submission, remove any existing flowplayer instance and create a new one with the specified file path or absolute URL. Despite the three other inputs not functioning, the overall system functionality at this point was enough to be considered a testing platform for video streaming, as you could use it to test playback of various static video files on different platforms. Before attempting to implement the transcoding system, this testing platform was used to perform a small amount of preliminary research into which non-standard video formats may be supported on different platforms, and also confirm which standard formats are functional. These tests were performed by transcoding some test files manually using ffmpeg with output container and compression formats specified on the command line, then loading the test files using the test player on each of the platforms and recording whether the file plays successfully. The results of these tests are presented below:

Table 2: Playback support for modern container and compression formats on popular platforms

Platform	**Container Format**	**Compression Format**	**Result**
Google Chrome	MP4	H.264	✓
Google Chrome	MKV	H.264	✗
Google Chrome	WEBM	VP8	✓
Google Chrome	MKV	VP8	✗
Mozilla Firefox	MP4	H.264	✗
Mozilla Firefox	MKV	H.264	✗
Mozilla Firefox	WEBM	VP8	✓
Mozilla Firefox	MKV	VP8	✓
Internet Explorer	MP4	H.264	✓
Internet Explorer	MKV	H.264	✗
Internet Explorer	WEBM	VP8	✗

Continued on next page

Table 2 – continued from previous page

Platform	Container Format	Compression Format	Result
Internet Explorer	MKV	VP8	✗
Apple Safari	MP4	H.264	✓
Apple Safari	MKV	H.264	✗
Apple Safari	WEBM	VP8	✗
Apple Safari	MKV	VP8	✗
Android	MP4	H.264	✓
Android	MKV	H.264	✗
Android	WEBM	VP8	✓
Android	MKV	VP8	✗
Apple iOS	MP4	H.264	✓
Apple iOS	MKV	H.264	✗
Apple iOS	WEBM	VP8	✗
Apple iOS	MKV	VP8	✗
Adobe Flash	FLV	H.264	✓

As these results clearly show, there is no simple solution to achieving cross-platform video playback, as no single format works on every platform. The minimal set of formats which must exist to support all popular platforms is MP4 / H.264 and WEBM / VP8. It is worth noting that although Adobe Flash is a technology which is becoming less relevant with modern developments such as HTML5, it is still supported by a very large proportion of desktop web browsers and it supports H.264 video in the FLV container. This is an important note because in some circumstances it may be useful to have a fallback option to play videos using Flash on desktop browsers; most of the large video streaming websites are still using Flash as the primary technology for playback at this time.

After performing these tests and confirming that there is no single format which allows

playback on all platforms, the next step was to implement a system to detect the supported formats on a particular device and load the correct version of the selected file automatically. Fortunately this proved to be built-in functionality in the Flowplayer API, which meant a single API call was all that was necessary to detect the best supported container format by the client platform.

5.2.1 Transcoding System

One of the core functions of the transcoding system is to quickly identify the bandwidth available to the client platform, to enable the transcoded video stream to be tailor-made to provide the best video quality possible over the clients internet connection without waiting. The original idea for how to do this involved downloading a small file to the client and measuring the time taken to download it. However we also asserted that the client must have to wait no more than 3 seconds for the video to load. As such, upon implementation it was found that this method alone is too primitive to return useful accurate results, as the client bandwidth can range from very slow (by today's standards, anything less than 50KB/s) to quite fast (above 2MB/s). This variety in client bandwidth made choosing the size of a single file to download difficult, prompting an iterative approach. To solve this problem, a small server-side function to serve arbitrary blobs of randomized binary data was created; 'servePayloadAction', and a recursive JavaScript function was created which starts by downloading a very small file (20KB) using Ajax, adding the time taken to download that file to a variable and then recursing, progressively increasing the size of the downloaded blob until the total time taken is over 3 seconds, at which point the average speed is estimated and returned to the calling function in kilobytes per second. The code for this functionality can be viewed in *appendix A*.

This algorithm proved to be very effective, reliably returning a useful bandwidth estimate after only 3 seconds. As such, this speed test is executed automatically as soon as the video playback page loads, and an loading spinner image is displayed while it is running to minimise abandonment.

As soon as the bandwidth measurement is complete, all the required variables to build the URL to the transcode stream are available; the source file path, supported format, available bandwidth and start position, which for this case is always 0. These variables are used to build a transcode URL in the following format:

/transcode/[Bit Rate]/[Start Second]/[Source File Path]/stream.[Container File Extension]

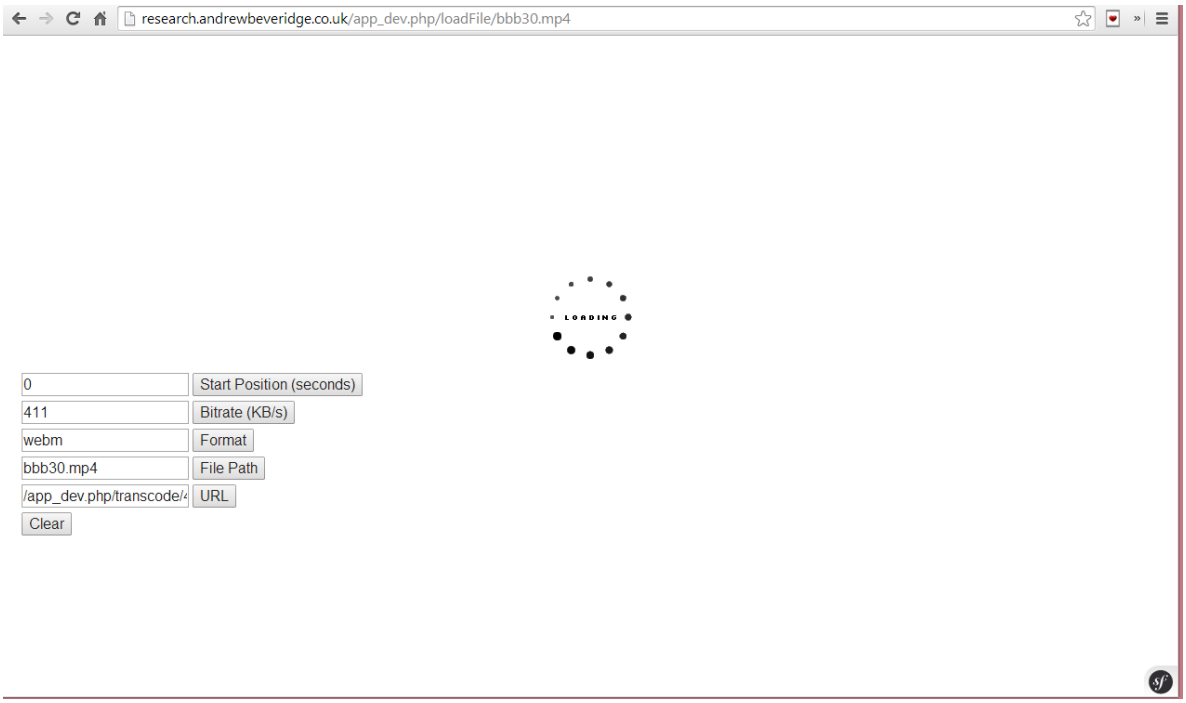

Figure 8: Playback page of testing platform during bandwidth measurement.

With the dynamically-built transcode URL complete, the loading spinner is hidden and the URL is loaded into a new Flowplayer instance. This causes Flowplayer to request that URL to stream video from, which is routed to the transcodeAction function in DefaultController.

transcodeAction is the largest and most central part of the transcoding system. It handles all HTTP requests, it creates, modifies and persists Client and TranscodeProcess entities, it starts and stops the actual transcode process which create the video stream played by the client, and it reads the content of the transcoded file and streams it to the client to be played.
The code for this function is well-commented and can be viewed in *appendix B*. However as it is quite lengthy, a somewhat simplified pseudocode is offered below.

Listing 1: Algorithm to stream a video file which is still in the process of transcoding

```
Function transcodeAction :
Inputs ( clientBitrate, startSecond, inputFilepath, containerFormat ) {

  Set up debug logs and error handling

  Decode input file path from URL parameter

  Read remote IP address of client from request headers

  Detect requested byte range from request headers

  Find any existing transcode processes owned by the client and stop them

  Build absolute path to requested input file on disk from input parameter
```

```
Execute FFprobe process to read metadata from input file, such as duration

Estimate transcoded file output size using simple calculation:
duration * bitrate

Build HTTP response headers to simulate the response from a web server
serving a static file {

  If requested range is outside sane range {

    Send HTTP code 416 (Range Not Satisfiable)

    Send Content-Range header with acceptable byte range values based on
    estimated output size

    Return HTTP response at this point with no content
  } Else {

    Send HTTP code 206 (Partial Content)

    Send Accept-Ranges header with value "bytes", to allow client to seek
    using byte-range requests

    Send Content-Length header with length value calculated from requested
    byte range start and end

    Send Content-Range header with requested byte range values and limit
    based on estimated size

    Send Connection header with value "Keep-Alive" to tell browser to keep
    reading despite delays

    Send ETag header with value generated using the md5 hash algorithm to
    allow correct caching

    Send Content-Type header with correct value for requested video format
  }

}

Send HTTP response headers to client before sending any data to allow for
clients which do range testing

Calculate point in video to start transcoding at, in seconds, from requested
byte range start

Create transcode process to transcode input file from start second to output
format at target bitrate

Store process ID of running transcode process in variable for future checks

Check for the existence of the output file and wait until it has been created

Transcode Buffer Loop {

  Read the current file size of the output file

  If the output file contains less than 40 kilobytes of data we have not sent
      {

      If transcode process with PID stored earlier is still running {

          Sleep for 1 second and loop
```

```
        } Else {

            Assume we have finished streaming the entire file, break out of loop

        }

    } Else {

        Set start byte to value of total bytes sent variable, starting at 0

        Set end byte to current length of output file

        Stream data from output file to client, starting at start byte and
        splitting read into buffered chunks of 8 kilobytes

        Add content length ( end byte - start byte + 1 ) to total bytes sent

    }

  }

  Reached the end of the Transcode Buffer Loop, return finished HTTP response

}
```

This algorithm, when implemented properly, provides a technique for serving video files which are not yet complete, enabling the streaming of a transcoded video file almost immediately after the transcode process itself starts. For this technique to work, the transcode process must be have a configuration which enables transcoding at a rate no less than 30 frames per second, which is the standard rate of playback for most video files commonly used today. Indeed, this is the sole function of the *createTranscodeProcess()* function in this implementation; this takes the specified transcode parameters and starts a transcode process with a pre-set set of configuration options for the output format which will reliably transcode at a rate faster than 30 frames per second on the test system. For example, the ffmpeg process executed when a request is made for an input file *bbb.mp4* to be transcoded to WEBM with a target bitrate of 140 kilobytes per second is:

```
ffmpeg -i "bbb.mp4"
-c:v libvpx -quality good -b:v 1120K -qmin 10 -qmax 42 -maxrate 1120K
            -bufsize 2240K -threads 14
-c:a libvorbis -qscale:a 3
-f webm -y bbb.mp4-$currentDateTime.webm
```

The command for the ffmpeg process to serve a request for an input file *Sintel.mkv* transcoded to MP4 with a target bitrate of 430 kilobytes per second is:

```
ffmpeg -i "Sintel.mkv"
-c:v libx264 -b:v 3440K -profile:v high -threads 14
-c:a libfdk_aac -vbr 3 -ac 2
-f mp4 -y Sintel.mkv-$currentDateTime.mp4
```

Note that while elsewhere in the implementation the client bitrate and target bitrate use *kilobytes* per second as the unit, ffmpeg uses bits as the unit for specifying bitrate and as such the values are multiplied by 8.

The end result from this transcodeAction function is that within a few hundred milliseconds of a request being made, the client platform receives a data stream containing video which is in the ideal format, using all of the available bandwidth to provide them with as high quality a video stream as supported by their internet connection.

From a user perspective, they click on a file to play, wait 3 seconds and it starts playing; regardless of the format, quality or filesize of the stored file on the server.

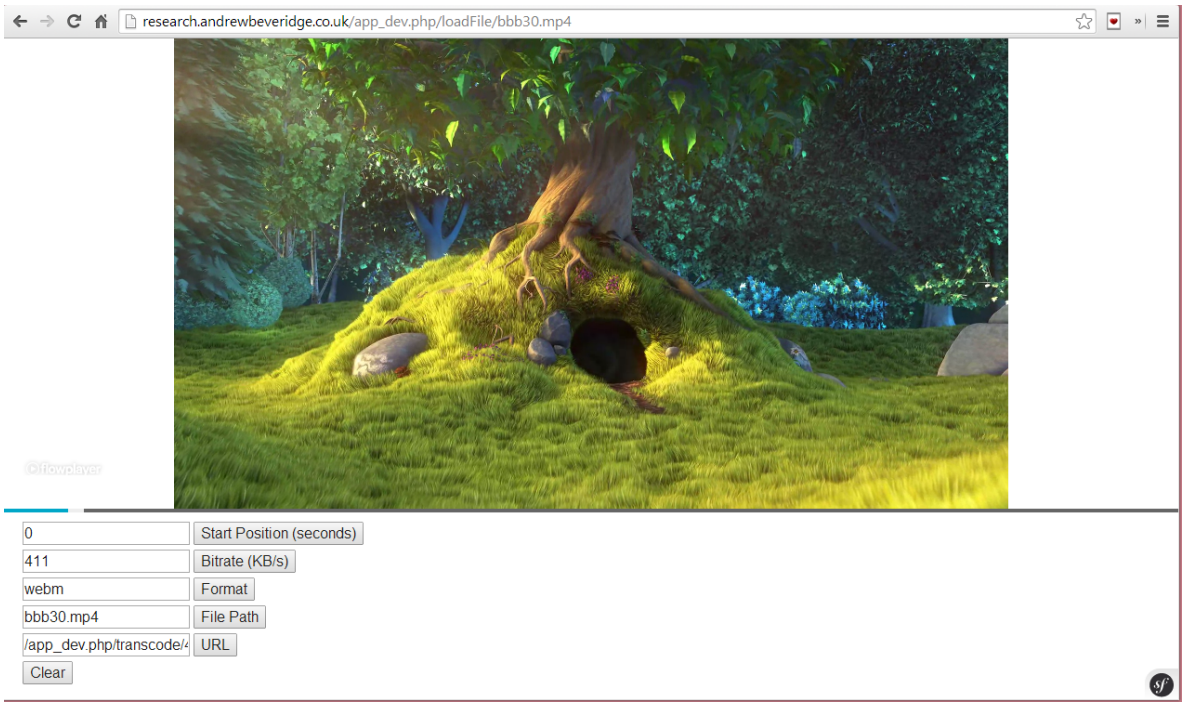

Figure 9: Playback page of testing platform showing transcoded video stream playing.

Seeking is achieved by some JavaScript code which replaces the Flowplayer seek event handler with custom functionality, which generates a new transcode URL based on the seek time clicked and loads the new URL into Flowplayer. This causes the server to kill the transcode process which was previously running and immediately start a new one, but reading from the source file starting from the selected second. Remarkably, this whole process takes less than half a second, so the user experience of seeking within a file which doesn't actually exist yet is not detrimentally affected. Crucially, this functionality means JavaScript is a requirement for this technique to be used. However since Flowplayer (and in fact, almost all HTML5 video players)

rely on JavaScript this point is considered moot.

As a testing platform for online video, this technique is extremely valuable as it decreases the latency of testing a source video in a particular format, at a particular bitrate, to less than 1 second. A researcher or developer can simply upload the high quality source file to the test server, select the test file to play and view it. Then, they can simulate viewing that video from clients with alternative specifications to their own machine by changing the parameters in the control boxes - the most important control being bitrate. Being able to quickly see how a video looks when transcoded to meet a specific bitrate requirement is very useful in many video production situations.

6 Testing and Evaluation

It is evident that the core functionality of transcoding and streaming a file does indeed work, from the simple fact that it is possible to click a test file link and have it start playing, despite the original being in a format not supported by the client platform.

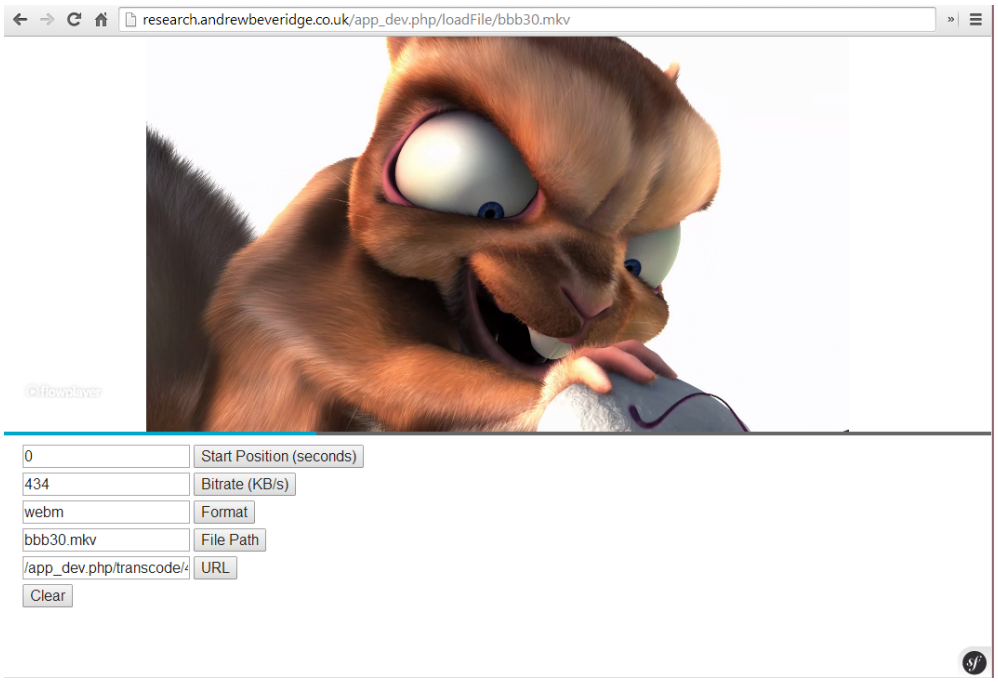

Figure 10: MKV source video transcoded to WEBM to enable playback in Google Chrome.

To test the system in more detail, it must be broken down into more distinct sections. The following sections will provide testing of specific areas, then finally offer a more general evaluation of the system functionality and performance.

6.1 Bandwidth Measurement

A large portion of functionality of the transcoding system relies upon reliable estimation of the bandwidth available to the client. As such, this was the first of several functions which were tested discretely. To test this, one needs to have an accurate known available bandwidth to compare with. As effective available bandwidth can be affected by a myriad of networking and software related causes, a fair measurement was sought. The chosen method was a virtual machine with a clean installation of Debian Linux to act as a client, running on the same host server as the virtual machine running the implementation. This client had a network interface throttling tool called "wondershaper" installed, and this tool was used to limit the upstream

and downstream bandwidth of the virtual network interface card to a specified value as such (in this case, 100 kilobytes per second):

```
# wondershaper eth0 100 100
```

Then the test platform was loaded, and the resulting average bandwidth calculated by the 3 second test performed by the measureBandwidth function was recorded. This test was then repeated for 3 different bandwidth values, and performed 3 times for each, with the following results:

Bandwidth Limit (KB/s)	**Measurement 1**	**Measurement 2**	**Measurement 3**
1000	972	944	988
100	92	89	96
10	3	9	4

Table 3: Test results for bandwidth measurement function

The conclusion drawn from these results is that despite a carefully controlled system there is some fluctuation in the results from the function. Also, it does not perform well when measuring a connection with a bandwidth which is too low to realistically support modern video streaming. However, given that it reliably returns a value which is less than 10% below the correct available bandwidth in less than 3 seconds, it is considered a success and is suitable to rely on for the intended purpose; estimating the bitrate at which the client connection can support streaming video.

6.2 Transcoded Video Quality

As it has been proven that video quality impacts viewer engagement (Krishnan and Sitaraman, 2012), we must ensure the output of the transcode processes are not producing files which would be perceived as poor quality; or at least, we must quantify the quality somehow to enable comparison of quality between bitrates and transcode techniques. However, visual quality of any given video file is an extremely subjective measurement which is very difficult to quantify objectively. Several techniques exist to quantify the similarity of one still image with another, such as Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity (SSIM), but neither are ideal for video streams as in order to calculate a measurement, one must extract every frame of the source video and every frame of the output video and compare each frame one by one. This method does not take into account visible artefacts of low bandwidth lossy video compression such as *tearing*, which cannot be represented in still image form.

However despite these drawbacks these indexes are widely used for objective quality comparison, as there is no widely known better metric yet. SSIM is the metric chosen for the following comparisons, as it tends to correlate more with subjective quality test results (Wang et al., 2003).

The following results show the average SSIM index of some transcoded files at specified bitrates. In very simplified terms, the average SSIM value provides a value between 0 and 1 which represents an objective measurement of the quality of the transcoded video, where 1 is identical to the source file and 0 is the exact opposite at every point. As such, it is an easy to comprehend objective value which can be used to estimate the quality of a transcode.

Figure 11: Output format: MP4, 200KB/s. **SSIM: 0.913**

Figure 12: Output format: WEBM, 200KB/s. **SSIM: 0.850**

As these results show, the transcoding systems works to an easily acceptable standard for both average (200KB/s) and low (50KB/s) bitrate restraints, and clearly the testing platform serves the intended purpose precisely, else the results obtained above would not have been

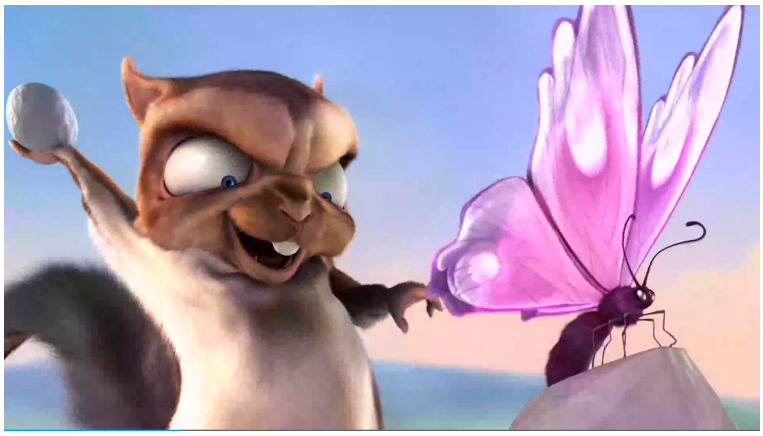

Figure 13: Output format: MP4, 50KB/s. **SSIM: 0.771**

Figure 14: Output format: WEBM, 50KB/s. **SSIM: 0.782**

possible. If SSIM is accepted as a fair metric to evaluate the output quality, at average bitrates both compression formats produce video streams which are around 90% of the source file quality, which is an impressive feat considering the configuration options which are required to enforce real-time transcoding to stream on demand.

7 Discussion

7.1 Limitations

While this technique successfully offers a solution to the problem of transcoding and streaming a video file on demand to multiple platforms from a web application, and presents this technique as a method which could potentially replace the standard model of transcoding all files on upload, no research has been performed into the scalability of this technique on live systems. Several possible challenges with this technique are perceived:

- The transcoding processes might prove to require too much processing power to be realistic on a live system with millions, thousands or even hundreds of users.
- A large scale web application which serves video to multiple global locations is likely to use a CDN of some sort, or at least have a far more complex infrastructure than the extremely simple test server this research was tested on. As such, it is likely that implementing this technique on a larger scale would require significant infrastructure reconfiguration or redistribution, which may outweigh any advantages.
- Storage space is getting cheaper all the time, and video compression formats are getting more computationally expensive to encode all the time. While it may be cost effective to trade CPU power for storage space right now, as codecs evolve and disk space gets cheaper, that balance may shift back towards being cheaper to store all transcoded formats permanently, in which case this technique becomes redundant.

7.2 Conclusion

This research project set about to provide valuable information about the effectiveness and scalability of various video transcoding technologies. Certainly it has provided valuable information about the effectiveness of at least one technique for transcoding video on demand; however the scalability of this technique has not been formally assessed. A system to test and evaluate methods of streaming high quality video was developed, and it is very functional and useful for any activity in which there is a need to test video quality when transcoded to a specific set of requirements. All requirements of the test system were successfully satisfied. An investigation into the possibility of streaming transcoded video on demand was undertaken, culminating in the design and implementation of a successful technique for solving this problem, which has not

been previously documented in academic research. However the scalability of it has not been measured using any specific process. In theory the technique implemented should be scalable if a caching system for transcoded files is implemented, but this is a field which requires further research. Calculating the economic cost of on demand transcoding and offering a comparison with existing methods of serving video (e.g. storing transcoded files on disk permanently) was originally intended to be researched as part of this project, but on further research it was found that the existing research by Kathpal et al. (2012) goes into sufficient detail for the purpose of this research, and in fact leads on to this research quite well, since the identified issue with latency identified as *Further Work* in that paper is directly solved by the transcoding system offered in this research. Overall, a new technique for streaming transcoded video was presented which solves a technical challenge which had not previously been documented, a testing platform was developed to enable testing of that and other transcoding systems, and general background research was performed to document the current state of video container and compression formats and their support on various platforms.

7.3 Future Work

Further research in this field should focus on taking the simple technique implemented in this research and finding out if it can be viable in real world, enterprise level use cases where scalability is crucial. Specific topics which could be investigated are: caching; for example storing the output from each complete transcode process for a short period of time, perhaps influenced by some estimated popularity factor of the video, to dramatically reduce wasted computation power by transcoding the same file to the same format multiple times for different clients; speed increases using multiple machines to perform transcodes; new compression formats which may be developed with greater platform support than the formats investigated in this research.

8 References

Deneke, T. (2011). *Scalable Distributed Video Transcoding Architecture*. PhD thesis, Åbo Akademi University.

Flowplayer, Ltd (2014). Flowplayer license: GPL license v3. `https://flowplayer.org/license/`.

Goh, K. H., Tham, J. Y., Zhang, T., and Laakko, T. (2011). Context-Aware scalable multimedia content delivery platform for heterogeneous mobile devices. In *MMEDIA 2011, The Third International Conferences on Advances in Multimedia*, page 1–6.

Huang, Z., Mei, C., Li, L. E., and Woo, T. (2011). CloudStream: delivering high-quality streaming videos through a cloud-based SVC proxy. In *INFOCOM, 2011 Proceedings IEEE*, page 201–205.

jQuery Foundation (2014). License | jQuery foundation. `https://jquery.org/license/`.

Kathpal, A., Kulkarni, M., and Bakre, A. (2012). Analyzing compute vs. storage tradeoff for video-aware storage efficiency. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, page 13–13.

Krishnan, S. S. and Sitaraman, R. K. (2012). Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, page 211–224, New York, NY, USA. ACM.

Li, Z., Huang, Y., Liu, G., Wang, F., Zhang, Z., and Dai, Y. (2012). Cloud transcoder: Bridging the format and resolution gap between internet videos and mobile devices. In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, page 33–38.

Liu, Y., Li, F., Guo, L., Shen, B., and Chen, S. (2012). A server's perspective of internet streaming delivery to mobile devices. In *INFOCOM, 2012 Proceedings IEEE*, page 1332–1340.

Potencier, F. (2014). Symfony2 license (current) - symfony. `http://symfony.com/doc/current/contributing/code/license.html`.

Q-Success (2014a). Usage statistics and market share of server-side programming languages for websites, april 2014. http://w3techs.com/technologies/overview/programming_language/all.

Q-Success (2014b). Usage statistics and market share of web servers for websites, april 2014. http://w3techs.com/technologies/overview/web_server/all.

Sanson, H., Loyola, L., and Pereira, D. (2012). Scalable distributed architecture for media transcoding. In *Algorithms and Architectures for Parallel Processing*, page 288–302. Springer.

Schwarz, H., Marpe, D., and Wiegand, T. (2007). Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120.

SkySQL Corporation (2014a). MariaDB license - MariaDB knowledgebase. https://mariadb.com/kb/en/mariadb-license/.

SkySQL Corporation (2014b). MariaDB vs. MySQL | MariaDB. https://mariadb.com/products/mariadb/mariadb-vs-mysql.

The FFmpeg Team (2014a). FFmpeg documentation : :. http://ffmpeg.org/general.html.

The FFmpeg Team (2014b). FFmpeg license and legal considerations. https://www.ffmpeg.org/legal.html.

The PHP Group (2014). PHP: history of PHP and related projects - manual. http://uk1.php.net/history.

Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. (2003). The SSIM index for image quality assessment. http://www.cns.nyu.edu/lcv/ssim/.

A Client Bandwidth Measurement

A.1 Server Side

Listing 2: Server-Side Bandwidth Measurement Helper

```
    // Send the browser a file with specified length to test connection speed
    public function servePayloadAction($lengthKB, $startTime)
    {
        $response = new Response();
        $filename = $this->get('kernel')->getRootDir() . "/../src/AB/Bundle/
            TranscodingExperimentsBundle/Resources/payload/5MB.bin";
        $response->headers->set('Cache-Control', 'private');
        $response->headers->set('Content-type', 'application/octet-stream');
        $response->headers->set('Content-length', $lengthKB * 1000);

        $d = $response->headers->makeDisposition(ResponseHeaderBag::
            DISPOSITION_ATTACHMENT, 'payload' . $lengthKB * 1000 . '.bin');
        $response->headers->set('Content-Disposition', $d);

        $response->sendHeaders();
        return $response->setContent(file_get_contents($filename, false, null,
            -1, $lengthKB * 1000));
    }
```

A.2 Client Side

Listing 3: Client-Side Bandwidth Measurement

```
function measureBandwidth(relativePath, runTimeMs, callback) {
  var speeds = [];
  var totalData = 0;
  var runStart = new Date().getTime();
  var testStart = new Date().getTime();
  var runEnd = runStart + runTimeMs;
  var average = 0;

  function logTestResults(length, start, end) {
    var KBps = Math.round(100 * (length / ((end - start) / 1000)) / 1024) / 100;
    speeds.push(KBps);
    totalData += length;
    average = Math.round(100 * (totalData / ((end - runStart) / 1000)) / 1024) /
        100;

    if (end > runEnd) return true;
    return false;
  }

  testStart = new Date().getTime();
  $.ajax(relativePath + "servePayload/50/" + testStart).done(function (msg) {
    if (logTestResults(msg.length, testStart, new Date().getTime())) return
        callback(average);

    testStart = new Date().getTime();
    $.ajax(relativePath + "servePayload/100/" + testStart).done(function (msg) {
      if (logTestResults(msg.length, testStart, new Date().getTime())) return
          callback(average);
```

```

      testStart = new Date().getTime();
      $.ajax(relativePath + "servePayload/250/" + testStart).done(function (msg) {
        if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);

        testStart = new Date().getTime();
        $.ajax(relativePath + "servePayload/500/" + testStart).done(function (msg) {
          if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);

          testStart = new Date().getTime();
          $.ajax(relativePath + "servePayload/1000/" + testStart).done(function (msg) {
            if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);

            testStart = new Date().getTime();
            $.ajax(relativePath + "servePayload/2000/" + testStart).done(function (msg) {
              if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);

              testStart = new Date().getTime();
              $.ajax(relativePath + "servePayload/3000/" + testStart).done(function (msg) {
                if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);

                testStart = new Date().getTime();
                $.ajax(relativePath + "servePayload/4000/" + testStart).done(function (msg) {
                  if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);

                  testStart = new Date().getTime();
                  $.ajax(relativePath + "servePayload/5000/" + testStart).done(function (msg) {
                    if (logTestResults(msg.length, testStart, new Date().getTime())) return callback(average);
                  });
                });
              });
            });
          });
        });
      });
    });
  });
}
```

B Main Controller

B.1 Transcode Action

Listing 4: Transcode Action

```
// Load in client IP, requested video and bitrate data, generate tailor made stream and start streaming it
public function transcodeAction($clientBitrate, $inputFilepath, $startSecond, $containerFormat)
{
  // Set up Symfony debugging
  Debug::enable();
  ErrorHandler::register();
  ExceptionHandler::register();
  // Debug errors by showing them in HTML despite supposedly being a video stream
  //$containerFormat = "html";

  // Set up Monolog logging to channel "transcode"
  $transcodeLog = $this->get('monolog.logger.transcode');
  $transcodeRangeRequestLog = $this->get('monolog.logger.transcode_range_request');

  // Stop PHP from timing out on long reads
  set_time_limit(0);

  // Get entity manager and Transcode entity repository
  $em = $this->getDoctrine()->getManager();
  $transcodeProcessRepository = $em->getRepository('ABTranscodingExperimentsBundle:TranscodeProcess');

  // Start looking at / parsing request variables and headers
  $inputFilepath = rawurldecode($inputFilepath);
  $clientIP = $this->container->get('request')->getClientIp();
  $clientBitrate = intval($clientBitrate);
  $transcodeLog->info("\n\n\n\n\ntranscodeAction(): Client GET request from IP: $clientIP for filepath $inputFilepath transcoded to $containerFormat with $clientBitrate KB/s bandwidth available");

  // Check if byte range request was made and parse to get rangeStart and rangeEnd values
  $rangeRequested = isset($_SERVER['HTTP_RANGE']) ? true : false;
  preg_match('/^bytes=(\d*)-(\d*)$/', @$_SERVER['HTTP_RANGE'], $rangeMatches);
  $rangeStart = intval(@$rangeMatches[1]);
  $rangeEnd = intval(@$rangeMatches[2]);
  $transcodeLog->info("transcodeAction(): Client byte range requested: $rangeStart to $rangeEnd");
  $transcodeRangeRequestLog->info("transcodeAction(): Client byte range requested: $rangeStart to $rangeEnd");

  // Get fingerprint/cookie/IP address from the client, pass it to getClient()
  // getClient will attempt to find a matching client entity or create a new one
  $client = $this->getClient(array(
    'ip' => $clientIP
  ));
  $clientID = $client->getId();
  $transcodeLog->info("transcodeAction(): Client linked up with ID: $clientID");

```

```
44      // If any running transcodes belong to this client, kill these transcode
            processes
45      // Also update the database if any processes owned by this client have
            naturally ended
46      $transcodeLog->info("transcodeAction(): Killing any running transcodes owned
             by this client");
47
48      // Check if any running transcodes are owned by this client.
49      $query = $transcodeProcessRepository->createQueryBuilder('tp')
50        ->where('tp.clientID = :clientID')
51        ->andWhere('tp.status = :status')
52        ->setParameter('clientID', $clientID)
53        ->setParameter('status', 'running')
54        ->getQuery();
55      $clientRunningTranscodes = $query->getResult();
56      $transcodeLog->info("transcodeAction(): Client has " . count(
            $clientRunningTranscodes) . " running transcode(s) in the database");
57
58      // Loop through any running transcodes and kill/update them
59      foreach ($clientRunningTranscodes as $runningTranscode) {
60        $runningTranscodePID = $runningTranscode->getProcessPID();
61        if ($this->killTranscodeProcess($runningTranscodePID)) {
62          $runningTranscode->setStatus('killed');
63          $transcodeLog->info("transcodeAction(): Killed running transcode with
                PID $runningTranscodePID");
64        } else {
65          $runningTranscode->setStatus('ended');
66          $transcodeLog->info("transcodeAction(): Updated transcode table since
                transcode {$runningTranscode->getId()} with PID $runningTranscodePID
                 has ended");
67        }
68      }
69      $em->flush();
70
71      // Build absolute path to input video file, from Symfony application root
            directory, hard-coded video folder and path requested by client
72      $rootDirectory = $this->get('kernel')->getRootDir() . '/../';
73      $videoDirectory = 'web/bundles/abtranscodingexperiments/videos/';
74      $absoluteInputFilepath = $rootDirectory . $videoDirectory . $inputFilepath;
75      $transcodeLog->info("transcodeAction(): Path to input video: {
            $videoDirectory}{$inputFilepath}");
76
77      // Get path to statically built FFmpeg binary in resources directory. This
            should be re-built with build.sh for target platform.
78      $ffprobe = FFMpeg\FFProbe::create(array(
79        'ffmpeg.binaries' => $this->get('kernel')->getRootDir() .
80          "/../src/AB/Bundle/TranscodingExperimentsBundle/Resources/public/bin/
                ffmpeg-static/target/bin/ffmpeg",
81        'ffprobe.binaries' => $this->get('kernel')->getRootDir() .
82          "/../src/AB/Bundle/TranscodingExperimentsBundle/Resources/public/bin/
                ffmpeg-static/target/bin/ffprobe"
83      ));
84
85      // Extract input file duration in seconds, using FFprobe
86      $duration = round($ffprobe->format($absoluteInputFilepath)->get('duration'),
             2);
87      $transcodeLog->info("transcodeAction(): Extracted duration from input file:
            $duration seconds");
88
89      // Estimate the file size of the completed transcode using client bandwidth
            and input duration. (KB/s * seconds) * 1024 = bytes
90      $estimatedOutputSize = round($clientBitrate * $duration * 1024);
```

```
91      $transcodeLog->info("transcodeAction(): Calculated estimate size for output
            file: $estimatedOutputSize bytes");
92
93      // Now we've parsed all our input variables and got a running transcode, let
            's start streaming
94      $response = new Response();
95      $transcodeLog->info("transcodeAction(): Cleaned up old transcodes and
            matched client; Building response");
96
97      // Set HTTP version to 1.1
98      $response->setProtocolVersion('1.1');
99      $transcodeLog->info("transcodeAction(): Setting HTTP version to 1.1");
100
101     // Check byte range for sanity
102     if ($rangeRequested) {
103       if ($rangeStart > $estimatedOutputSize) {
104         $transcodeRangeRequestLog->info("transcodeAction(): Client requested
                range start of $rangeStart which is > file size, sending HTTP 416");
105
106         // Tell the client that range was stupid and we don't like it
107         $response->setStatusCode(Response::HTTP_REQUESTED_RANGE_NOT_SATISFIABLE)
                ;
108         $transcodeLog->info("transcodeAction(): Setting response code to 416
                Requested Range Not Satisfiable");
109
110         // Tell client the theoretical range of bytes it can request
111         $response->headers->set("Content-Range", "bytes bytes */
                $estimatedOutputSize");
112         $transcodeLog->info("transcodeAction(): Adding header: Content-Range:
                bytes */$estimatedOutputSize and exiting");
113
114         return $response;
115       }
116
117       // Since we're technically responding with part of the file, it's an HTTP
              206
118       if($containerFormat !== "flv") {
119        $response->setStatusCode(Response::HTTP_PARTIAL_CONTENT);
120        $transcodeLog->info("transcodeAction(): Setting response code to 206
              Partial Content");
121       }
122     }
123
124     // This header should go along with all responses to reassure client they
            can seek, unless flv
125     if ($containerFormat == "flv") {
126       $response->headers->set("Accept-Ranges", "none");
127       $transcodeLog->info("transcodeAction(): Adding Accept-Ranges: none header
              to stop seeking");
128     } else {
129       $response->headers->set("Accept-Ranges", "bytes");
130       $transcodeLog->info("transcodeAction(): Adding header: Accept-Ranges to
              allow seeking");
131     }
132     // The end byte is unlikely to have been requested so it will probably be
            set to 0, set it to the last byte.
133     $rangeEnd = $rangeEnd == 0 ? $estimatedOutputSize - 1 : $rangeEnd;
134
135     // In theory the amount of data we intend to sent is the end minus start
            byte, but rangeEnd is 0-indexed
136     $length = $rangeEnd - $rangeStart;
137
138     $response->headers->set("Content-Length", "$length");
```

```
139     $transcodeLog->info("transcodeAction(): Adding header: Content-Length:
            $length");
140
141     // This tells the browser the theoretical range of bytes it can request
142     if($containerFormat != "flv") {
143      $response->headers->set("Content-Range", "bytes $rangeStart-$rangeEnd/
             $estimatedOutputSize");
144      $transcodeLog->info("transcodeAction(): Adding header: Content-Range: bytes
              $rangeStart-$rangeEnd/$estimatedOutputSize");
145     }
146
147     // Follow apache as closely as possible
148     $response->headers->set('Connection', 'Keep-Alive');
149     $transcodeLog->info("transcodeAction(): Adding header: Connection: Keep-
            Alive");
150
151     // Follow apache as closely as possible
152     $etag = md5($absoluteInputFilepath.$clientBitrate);
153     $response->headers->set('ETag', $etag);
154     $transcodeLog->info("transcodeAction(): Adding header: ETag: $etag");
155
156     // Output appropriate content type header
157     if ($containerFormat == "webm") {
158       $response->headers->set('Content-Type', 'video/webm');
159     } elseif ($containerFormat == "mp4") {
160       $response->headers->set('Content-Type', 'video/mp4');
161     } elseif ($containerFormat == "flv") {
162       $response->headers->set('Content-Type', 'video/x-flv');
163     } else {
164       // This is just for debugging, so we can more easily see symfony errors
165       $response->headers->set('Content-Type', 'text/html');
166     }
167     $transcodeLog->info("transcodeAction(): Adding header: Content-Type for
            $containerFormat video stream");
168
169     // Send client the headers before we start sending binary data
170     $response->sendHeaders();
171     $transcodeLog->info("transcodeAction(): Sending headers to client, ready to
            start stream");
172
173     // Calculate start point for transcode based on range request
174     //$startSecond = $this->getTranscodeStartSecond($rangeStart,
            $estimatedOutputSize, $duration);
175     //$transcodeLog->info("transcodeAction(): Calculated startSecond:
            $startSecond from rangeStart: $rangeStart ");
176
177     // Now we've cleaned up and killed old processes, let's create a new
            transcode process
178     $transcodeProcess = $this->createTranscodeProcess($clientID, $clientBitrate,
             $absoluteInputFilepath, $containerFormat, $startSecond);
179     // Get output file path
180     $outputFilepath = $transcodeProcess->getOutputFilepath();
181     // Process PID, for keeping track of transcode
182     $pid = $transcodeProcess->getProcessPID();
183
184     // Try opening file until file is created - this is just in case the
            transcode process takes a while to start up
185     // Log how long we've been waiting, error if more than 10 seconds
186     $sleepTime = 0;
187     while (1) {
188       $fp = @fopen($outputFilepath, 'rb');
189       if ($fp === FALSE) {
```

```
190         $transcodeLog->debug("transcodeAction(): Waiting for output file to be
                created. Time so far: $sleepTime seconds");
191         if ($sleepTime > 10) {
192           $transcodeLog->error("transcodeAction(): Timed out waiting for more
                  than 10 seconds for output file to be created; check transcode log
                   for errors");
193           die();
194         }
195         // Sleep for 0.01 seconds at a time between polling fopen for output
                file
196         usleep(10000);
197         $sleepTime += 0.01;
198       } else {
199         $transcodeLog->info("transcodeAction(): Output file exists and is
                readable");
200         break;
201       }
202     }
203
204     // Ok, now we're finally at the meat of this function. We've got a file
            pointer to an open video file,
205     // in theory we just need to read data from the file and send it to the
            client. Since PHP doesn't echo anything immediately, we do this
206     // in Large chunks and flush them to the client in every loop iteration.
207     // However, we also need to check whether we actually have Large of data to
            send to the client first, and delay until we do.
208
209     // Stream data of exact length
210     if($rangeRequested) {
211      $transcodeLog->info("transcodeAction(): Processing client requested range:
            start $rangeStart, end $rangeEnd, length $length");
212      $transcodeRangeRequestLog->info("transcodeAction(): Processing client
            requested range: start $rangeStart, end $rangeEnd, length $length");
213     } else {
214      $transcodeLog->info("transcodeAction(): No range requested by client, range
            : start $rangeStart, end $rangeEnd, length $length");
215      $transcodeRangeRequestLog->info("transcodeAction(): No range requested by
            client, range: start $rangeStart, end $rangeEnd, length $length");
216     }
217
218
219     // Cheat headers block - if client requests last few KB from end of file, or
            first few from start, read them from static webm file
220     // Handle tiny file requests
221     if($length < 512000) {
222         $transcodeRangeRequestLog->info("transcodeAction(): Cheating by reading
                $length bytes from start of static file!");
223         $staticWebMFilepath = "/home/andrew/Other/transcoding-experiments/web/
                bundles/abtranscodingexperiments/videos/bbb30.webm";
224         $fpstatic = @fopen($staticWebMFilepath, 'rb');
225         $this->readBufferedData($fpstatic, 0, $length);
226         $totalBytesSent += $length;
227     }
228     // If range starts in the last 500KB of the fake file, read from static file
229     if( ($estimatedOutputSize-$rangeStart) < 512000) {
230         $transcodeRangeRequestLog->info("transcodeAction(): Cheating by reading
                $length bytes till end of static file!");
231         $staticWebMFilepath = "/home/andrew/Other/transcoding-experiments/web/
                bundles/abtranscodingexperiments/videos/bbb30.webm";
232         $fpstatic = @fopen($staticWebMFilepath, 'rb');
233
234         $filesizestatic = filesize($staticWebMFilepath);
235         $staticstart = $filesizestatic - $length;
```

```
236
237         $this->readBufferedData($fpstatic, $staticstart, $length);
238         $totalBytesSent += $length;
239     }
240     // Cheat the first 500KB even if they make a normal request
241     if($rangeStart == 0) {
242         $transcodeRangeRequestLog->info("transcodeAction(): Cheating by reading
                512000 bytes from start of static file!");
243         $staticWebMFilepath = "/home/andrew/Other/transcoding-experiments/web/
                bundles/abtranscodingexperiments/videos/bbb30.webm";
244         $fpstatic = @fopen($staticWebMFilepath, 'rb');
245         $this->readBufferedData($fpstatic, 0, 512000);
246         $totalBytesSent += 512000;
247     }
248
249     // Transcode Buffer Loop: Send the requested amount of data to the client
            whilst waiting for transcode to actually write it
250     // In a loop which ends when we've met the requested amount of data, wait
            until transcode has written a chunk of new data, then stream it
251     $totalBytesSent = 0;
252     while (1) {
253       // Stop if we've sent more data than we were meant to
254       if($totalBytesSent >= $length) {
255        $transcodeLog->info("transcodeAction(): TotalBytesSent == length so we've
                finished sending the requested data. Killing PID $pid");
256        $this->killTranscodeProcess($pid);
257        break;
258       }
259
260       $stat = fstat($fp);
261       $outputSize = $stat['size'];
262       $transcodeLog->debug("transcodeAction(): Inside transcode buffer loop,
            output filesize: $outputSize");
263
264       // Wait for output file to have enough data to stream another chunk
265       $sleepTime = 0;
266       $minimumSize = $totalBytesSent + 40960;
267       if ($outputSize < $minimumSize) {
268         // If the output file hasn't met our minimum size for buffering yet,
                continue waiting till it has
269         $pidCheck = $this->checkProcess('ffmpeg', $pid);
270         if ($pidCheck === false) {
271           $transcodeLog->info("transcodeAction(): Transcode process with PID:
                $pid seems to have finished. Successfully streamed $totalBytesSent
                bytes.");
272           break;
273         }
274
275         $transcodeLog->debug("transcodeAction(): Inside transcode buffer loop,
                waiting for output size $outputSize to be > $minimumSize");
276         if ($sleepTime > 60) {
277           $transcodeLog->error("transcodeAction(): Inside transcode buffer loop,
                timed out waiting for more than 60 seconds for transcode to write
                enough data");
278           break;
279         }
280         // Sleep for 1 second at a time between checking output filesize
281         usleep(1000000);
282         $sleepTime += 1;
283         continue;
284       }
285
286       // We essentially want to read exactly $length bytes from the output file
```

```
      // But since we have to wait for the transcode to write data first, we can't just read it all at once
      // Instead we wait till a chunk of new data exists, read as much as we can and add the number of bytes read to $totalBytesSent
      // Then we iterate again, wait for more new data, read the new chunk starting where we left off and repeat until $totalBytesSent = $length


      // Calculate the bytes we will actually read from the output file
      if( $outputSize < $length ) {
      // Output file is bigger than $bufferMin (otherwise we'd still be waiting), but less than the total desired length.
      // Read as much data as we already do have in the output file as fast as possible, our client is hungry!
        $endByte = $outputSize - 1;
      } elseif( $outputSize >= $length ) {
      // Output file is larger than the length requested by the client. We should be able to read the remaining amount in the range.
        $endByte = $length;
      }

      // Start at whatever position in the output file we're already at
      $startByte = $totalBytesSent;
      $endByte = $outputSize;

      // Amount of data we are reading this iteration is $endByte - $startByte
      // On first run this will likely be $outputSize - 0
      // We need to keep track of exactly how much data we have already sent
      $dataToRead = $endByte - $startByte + 1;
      $totalBytesSent += $dataToRead;

      $transcodeLog->debug("transcodeAction(): Inside transcode buffer loop, reading $dataToRead bytes of new data from $startByte to $endByte");
      $this->readBufferedData($fp, $startByte, $endByte);
    }

    return new Response();
  }
```