Reliable Distributed Video Transcoding System

ŽYGIAMANTAS BRUZGYS

Master’s Degree Project
Stockholm, Sweden June 2013

TRITA-ICT-EX-2013:183
Reliable Distributed Video Transcoding System

ŽYGIMANTAS BRUZGYS

Master’s Thesis
Supervisor: Björn Sundman
Examiner: Johan Montelius

TRITA-ICT-EX-2013:183
Abstract

The video content is becoming increasingly popular in the Internet. With an increasing popularity, increases the variety of different devices that are used to play video. Video content providers perform transcoding on video content, thus enabling it to be replayed on any consumer’s device. Since video transcoding is a computationally heavy operation, video content providers search a way to speed-up the process. In this study we analyse techniques that can be used to distribute this process across multiple machines. We propose a distributed video transcoding system design that is scalable, efficient and fault-tolerant. We show that our system configured with 16 worker machines performs the transcoding up to 15 times faster compared to the transcoding time on a single machine that does not use our system.
# Contents

**Contents** iv  
**List of Figures** vi  
**List of Acronyms and Abbreviations** vii  
**List of Definitions** ix  

1 Introduction 1  
1.1 About Screen9 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
1.2 Limitations of the Research . . . . . . . . . . . . . . . . . . . . . . 3  
1.3 Structure of the Report . . . . . . . . . . . . . . . . . . . . . . . . . 3  

2 Background 5  
2.1 Understanding Video Compression . . . . . . . . . . . . . . . . . . . 5  
2.2 Ordinary Video Structure . . . . . . . . . . . . . . . . . . . . . . . . 6  
2.3 Transcoding Video . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9  

3 Distributed Video Transcoding 11  
3.1 Segmenting Video Streams . . . . . . . . . . . . . . . . . . . . . . . . 11  
3.2 Effective Segmentation . . . . . . . . . . . . . . . . . . . . . . . . . . 13  
3.3 Balancing the Workload . . . . . . . . . . . . . . . . . . . . . . . . . . 14  

4 System Design 17  
4.1 Main Components . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
4.2 Technology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
4.2.1 GlusterFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
4.2.2 ZooKeeper . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
4.2.3 FFmpeg . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20  
4.3 Queue Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21  
4.3.1 Worker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  

5 Evaluation 29  

6 Discussion 37
CONTENTS

6.1 Scalability and workload balancing ........................................... 37
6.2 Overhead ................................................................. 38
6.3 Fault-tolerance .......................................................... 38

7 Conclusions ............................................................................. 41

Bibliography ............................................................................. 43
## List of Figures

1.1 Classification of video transcoding operations ........................................... 2

2.1 Temporal redundancy between subsequent frames [20] .......................... 6

2.2 MPEG hierarchy of layers ........................................................................ 7

2.3 An example of closed and open group of pictures (GOPs) ........................ 8

2.4 General transcoder architecture .............................................................. 9

3.1 General distributed stream processing scheme ........................................... 12

3.2 Relationship between playback order and decoding order ....................... 12

3.3 Graphs showing transcoding time dependency on a video file size .......... 13

3.4 General distributed stream processing scheme ........................................... 18

4.1 ffmpeg transcoding process ...................................................................... 20

4.2 Activity diagrams showing behaviour of get method ............................... 22

4.3 Throughput and latency of different queue implementations .................. 24

4.4 Latency spreads for different fault-tolerant queue implementations ........ 25

4.5 Class diagram of the worker ................................................................... 26

5.1 Transcoding times with different number of workers ............................... 30

5.2 Speed-up of each video transcoding operation when using different transcoding profiles ................................................................. 31

5.3 Network usage when during the transcoding session with 8 workers ........ 32

5.4 CPU usage of one worker during the transcoding session with 8 workers 33

5.5 CPU usage of all machines during the transcoding session with 8 workers 33

5.6 Time-line of transcoding session with 8 workers ...................................... 34

5.7 Transcoding speed-up of a larger (21 min) video ...................................... 34

5.8 CPU usage of all machines during the larger video transcoding session with 8 workers ................................................................. 35

5.9 Segmentation and concatenation times .................................................... 36
List of Acronyms and Abbreviations

API  Application Programming Interface  
CBR  Constant Bit-Rate  
Codec  Coder-Decoder  
CPU  Central Processing Unit  
DCT  Discrete Cosine Transform  
DTS  Decoding Timestamp  
DVD  Digital Video Disc  
FUSE  File-system in User-space  
Gb/s  Gigabits per Second  
GHz  Gigahertz  
GOP  Group of Pictures  
HD  High-Definition  
MB  Megabyte(s)  
P2P  Peer-to-peer  
PTS  Playback Timestamp  
RAM  Random Access Memory  
VBR  Variable Bit-Rate
List of Definitions

1080p A video parameter showing that a video is progressive and its spatial resolution is $1920 \times 1080$.

B-frame A compressed video frame that needs some preceding and subsequent frames to be decoded first in order to decode this frame.

Bit-rate The number of bits that are processed per unit of time.

I-frame A video frame that does not have any references to other frames.

Interlaced video A technique of increasing the perceived frame rate without consuming extra bandwidth.

P-frame A compressed video frame that needs some preceding frames to be decoded first in order to decode this frame.

Progressive video A video where each frame contains all lines and not only even or odd lines as in interlaced video.

Spatial resolution A video parameter stating how many pixels there are in one frame.

Temporal resolution A video parameter stating how many frames there are shown in one second.

Video container A video file format that describes how different streams are stored in a single file.
Chapter 1

Introduction

Video content is currently becoming increasingly popular in the Internet. According to YouTube statistics [6], 72 hours of video are uploaded to YouTube every minute and over 4 billion hours of video are watched each month on YouTube. Video consumers watch videos from different devices that have different screen sizes and computation power. Thus, one of the main challenges for video content providers is to adapt video content to a screen size, computation power and network conditions of each consumer’s device. For such adaptation a video transcoding is used. During the video transcoding a video signal representation is converted to other representation, i.e. video bit-rate, spatial resolution (also referred as video image resolution) or temporal resolution (also referred as frame-rate) are adjusted for specific needs [28]. Video content providers transcode a single video to many different formats in order to later serve the same video content to different consumer devices.

Video transcoding operations can be classified into two groups: heterogeneous and homogeneous [7]. Figure 1.1 visualises such classification. Heterogeneous transcoding is a process when the format of the video is changed, e.g. video container, coding-decoding algorithm (also referred as codec) or an interlaced video is changed to progressive video and vice-versa. Homogeneous transcoding does not change the format of the video, but changes its quality attributes such as bit-rate, spatial or temporal resolution, or converts variable bit-rate (VBR) to constant bit-rate (CBR) and vice versa. During a single video transcoding session multiple operations can be applied, e.g. a video container, a codec and a couple of quality attributes can be changed.

A video container is a file format that specifies how different video, audio, subtitle streams coexist in a single file. Different video containers support differently encoded streams, e.g. WebM container supports only VP8 video codec and Vorbis audio codec, while MP4 supports many different popular codecs, and Matroska container may contain streams encoded with almost any codec. Codecs are used to compress and decompress video files. Today there is a significant amount of codecs. To name a few of them, H.261, H.262 and H.263+ are used for low bit-rate video applications such as a video conference. MPEG-2 [7] targets high quality applica-
Figure 1.1: Classification of video transcoding operations

Figure 1.1: Classification of video transcoding operations

and is widely used in digital television signals and DVDs. MPEG-4 or H.264 are codecs that are becoming increasingly popular for video streaming applications and on-line video services. The transcoding system that we develop is expected to transcode user uploaded videos, this means that it should support many different video formats.

The bottleneck of video transcoding is a central processing processor (CPU). Several research articles propose methods that increase transcoding efficiency by using motion or coding mode information in order to reduce the amount of data needed to be decoded and encoded [16, 25, 12]. Other research studies propose methods that split video into segments, transcode these segments on a couple of processors and possibly on a couple of machines, and later join transcoded segments back into a single video piece. Such methods use cluster [21], peer-to-peer (P2P) [9] or volunteer cloud [15] infrastructures. However, little research that has been done considers fault-tolerance during the video transcoding process.

The goal of the thesis is to design, implement and test a distributed video transcoding system that is:

- **general purpose**, i.e. supports many different video codecs and containers,
1.1 About Screen9

Screen9 is a Swedish company that provides on-line video services. The company develops an Online Video Platform that allows customers to provide a high quality video experience to their users across different devices such as computers, smartphones or tablets. The company is responsible for storing, transcoding and streaming videos across the Internet, as well as providing video players on some platforms, if necessary. Online Video Platform then later provides detailed statistics for customers that gives them an insight on how their video content is consumed.

1.2 Limitations of the Research

This research does not include any transcoding process optimisations, such as possibilities of reusing motion vectors of an original codec. It would be very difficult to provide a general video transcoding system that supports many codecs and is optimized in such way. Instead of optimising a transcoding process, we will simply distribute the computation along many computers. For this purpose we use a general purpose transcoder that is already developed.

1.3 Structure of the Report

In Chapter 2 we introduce the definitions that are needed in order to understand the domain. We briefly explain how video compression works, what is the structure of the video, and what is video transcoding. We review existing solutions in speeding-up video transcoding operation. In Chapter 3 we introduce what is distributed processing and how it is possible to transcode the video in a distributed fashion. Chapter 4 is for explaining our proposed system and its design. In Chapter 5 we show and explain the results of experiments that we have performed. And finally we draw the conclusions in Chapter 7.
Chapter 2

Background

In this chapter we introduce definitions that are needed to understand the complexity of video transcoding. This includes a short introduction to a video stream structure. Finally, we present current video transcoding research issues.

2.1 Understanding Video Compression

Digital video compression, developed in the 1980s, made it possible to maintain various telecommunication applications, such as teleconferencing, audio conference, digital video streaming, file transfers, broadcast video, HDTV, etc. Compression is a process intended to yield a compact digital representation of a signal. Therefore, video compression is a reduction of a video digital representation bit rate, together with motion estimation, compensation, entropy coding, etc.

Video consist of a sequence of images called frames. In order to ensure that a human eye perceives a smooth motion as opposed to separate images, a number of frames per second has to be showed sequentially. Today films are usually shot with temporal resolution of 24 frames per second. Uncompressed video of such films contains a large amount of data, e.g. a 1080p HD video (with 24 frames per second) would normally produce a data rate of:

\[
1920 \times 1080 \times 3 \times 8 \times 24 \approx 1139.06 \text{ Mbits/s}
\]

It is impossible to provide a smooth playback of such video stream when it is transferred over a gigabit Ethernet line without pre-buffering more than the half of the video. This is the reason why before distribution video streams are compressed using codecs. Video compression employs redundancy and irrelevancy reductions:

- Redundancy reduction exploits the fact that neighbouring video stream frames and regions of the same frame contain many similarities. These similarities are called temporal and spatial respectively. H.264/AVC codec takes an advantage of temporal similarities by forming a prediction from data in one, or
possibly more, preceding or following frames, and then codes only differences between the prediction and the actual frame. Figure 2.1 illustrates temporal similarities between two subsequent frames. The codec exploits spatial similarities similarly as temporal similarities except that prediction is formed not from different frames, but from regions in the same frame.

- Irrelevancy reduction exploits how the human brain perceives visual and aural information. While watching video, human observers usually focus on a small region, such as a face or motion activity, rather than the whole scene. This provides an opportunity to greatly compress the peripheral region where the observer may not notice the quality degradation.

Overall, digital video compression technologies have influenced the way visual information is created, exchanged and consumed. A variety of different compression schemas exist [26], thus standardisation and unification of these techniques are crucial for the niche.

2.2 Ordinary Video Structure

As mentioned in the Section 2.1, frames in a compressed video stream usually contain differences between predictions and actual frames. In order to store such information different codecs impose different stream structures. Figure 2.2 visualises a general structure of MPEG layers.

We are interested in how to split a video stream, so that we could distribute the workload across multiple machines. For this purpose the right layer has to be chosen. Choosing different layers imposes different performance results in terms of total transcoding time and implementation complexity, e.g. splitting at macro-block layer gives finer granularity, but it also requires larger amount of communication between multiple machines because of dependencies between subsequent macro-blocks. An algorithm performing such task would have higher complexity. Moreover, different codecs encode macro-blocks slightly differently, e.g. Theora codec also introduces
another term *super block*: a block that stores a group of blocks [10]. Therefore, it is not a good choice to split at the macro-block level.

All codecs operate with frames. Different types of frames store different amount of information. Depending on how much information a frame stores, it can be one of these types [17, 29]:

- **I-frame** (also known as *intra, reference or key frame*) contains all the necessary data in order to recreate an image. This type of frames does not require data from other frames. When video seek is requested most applications looks for the closest I-frame and start building a picture from this frame.

- **P-frame** (also known as *predicted frame*) encodes the difference between prediction based on the closest preceding I or P-frame and the actual frame (see Section 2.1). P-frames may also be called *reference frames*, because neighbouring B and P-frames can refer to them. This type of frames usually takes less amount of space then I-frames.

- **B-frame** (also known as *Bi-directional frame*) takes the least amount of space. This type of frames use information from both preceding and following P or I-frames.

In conclusion, I-frames contain full picture information and do not depend on other frames, whereas P and B frames are sets of instructions to convert the pre-
vious and/or following frames into the current picture. Such frame dependency is visualised in Figure 2.3. In this figure, arrows show what frames are needed in order to decode the current frame. If the frame, which the current P or B frame depends on, was lost it would be impossible to correctly decode the frame. It is very important to address this issue when splitting video stream into multiple segments.

![Diagram of Closed GOP](image.png)

(a) Closed GOP

![Diagram of Open GOP](image.png)

(b) Open GOP

Figure 2.3: An example of closed and open group of pictures (GOPs)

A group of pictures (GOP) setting defines the pattern how I, P and B frames are used. This layer is the most promising for video splitting because it groups frames that have strong dependencies between each other. Depending on the frame pattern used in GOP, GOPs can be either Closed or Open:

- **Closed GOP** does not contain any frames that refer to a frame in the previous and/or subsequent GOPs. Such GOP always begins with an I-frame.

- **Open GOP** contains frames that refer to frames in the previous and/or subsequent GOPs. Such GOP provides slightly better compression because it is not required to contain any I-frames. However, in order to display an Opened GOP, other frames have to be decoded and this increases the seek response time.

Open and Closed GOP structures are shown in Figure 2.3. Arrows point to other frames that are needed to be decoded before the actual frame can be decoded.
2.3 Transcoding Video

A video transcoding is a process when a video signal representation is converted to other representation, i.e. one or several video attributes, such as bit-rate, spatial resolution or temporal resolution, are adjusted for specific needs. As mentioned in Chapter 1, video transcoders can perform heterogeneous and/or homogeneous operations. To perform such operations, video transcoders generally cascade a decoder and an encoder: the decoder decodes the video bit-stream, then operations such as changing a spatial or temporal resolution are performed, and later the encoder re-encodes the resulting bit-stream into a target format. Figure 2.4 visualises the architecture of such transcoder.

![Figure 2.4: General transcoder architecture](image)

Such video transcoding scheme is computationally expensive. This scheme does not use many coding parameters and statistics such as motion or coding mode information, which can be obtained from the input compressed video stream. Consequently, there is much research done that focuses on reducing the complexity of video transcoding process and thus reducing the total time it takes to transcode the video stream. T. Shanableh et al. state [23] that around 60% of total encoding time encoder spends on calculating motion vectors. A significant reduction of total transcoding time was achieved with the reuse of macro-block information. This work was later extended to discrete cosine transform (DCT) domain [24]. In [11] authors proposed a frame-rate control scheme that has a lower computational complexity. This scheme can adjust the number of skipped frames according to information of incoming motion vectors.

It is more difficult to optimise heterogeneous transcoding operations because different codecs encode information, such as motion vectors, differently. Therefore, heterogeneous transcoder architecture is more complex and may need some assumptions, e.g. in [23] a proposed algorithm makes an assumption that the motion between frames is uniform. The task gets more difficult when codecs use different techniques for encoding such information. One of the examples of more complex codecs is H.264. This codec is increasingly popular and offers a higher quality at all
bit rates. However, syntax is very different compared to other popular codecs, such as DivX, e.g. H.264 employs $4 \times 4$ integer transformation instead of $8 \times 8$ DCT as it is used in many other codecs, and H.264 uses different motion vector prediction coding algorithms. Therefore, transcoder cannot use motion vectors extracted from a different video source directly. Clearly, in order to build a general transcoder that can perform both heterogeneous and homogeneous operations and convert from any codec to any other codec, it is reasonable to use a general cascaded decoder and encoder architecture. As Figure 2.4 suggests, another advantage of such architecture is the ability to transcode to multiple codecs simultaneously.

The other approach to speeding up transcoding is to distribute the workload across multiple machines. Some research [21, 27] describes a distributed video transcoding system that consist of a single source, multiple transcoding, and a single merging machines. The source machine is responsible for segmenting the video and passing these segments to transcoding machines, whereas the merging machine is responsible for merging the transcoded segments back to a single video. Other research [9] employs peer-to-peer networks in order to perform transcoding tasks. Similarly, this system has media source, receiver, and transcoder roles. Another research [15] employs idle computing resources of home users. This work proposes a middle-ware architecture called Ginger and demonstrates that it is possible to perform video transcoding using such system. However, there is no research that concentrates on making the system fault-tolerant, and no research analyse the impact of a performance when failures occur.
Chapter 3

Distributed Video Transcoding

A distributed system can be described as a collection of autonomous processors that communicates with each other over a communication network. Such system has a potential to provide more reliable service because of a possibility of replicating computer resources. This more reliable service has a higher availability, i.e. it has a higher chance to be accessible at any time, and is more fault-tolerant, i.e. it has an ability to recover from system failures. A distributed system has also a potential to be scalable, i.e. such system performance has a potential to improve after adding more machines to the system.

These characteristics of a distributed system are desired to be in a video transcoding service. This is because video transcoding is a CPU bound process and there is a need to increase the speed of video transcoding as demand of such service increases. The most useful characteristic of a distributed system is an ability to add more computers in order to decrease total processing time (this is often referred as horizontal scaling).

Figure 3.1 shows a general distributed stream processing scheme. This scheme consists of two dedicated nodes: one for distributing segments of the stream, the other for concatenating stream segments back into a single stream, and number of worker nodes that process the stream segments. Such scheme allows us to utilise several computer nodes and improve the total throughput of the system. Adding more workers to such scheme should improve total throughput provided that enough bandwidth is available and computation to communication ratio is high.

3.1 Segmenting Video Streams

Distributed video transcoding system relies on a fact that a given video can be segmented to a number of segments which can be later transcoded on several machines. As it was described in Section 2.2 a video stream is made from several layers. GOP (Group of Pictures) is the most promising layout. However, it is not guaranteed that a GOP will contain an I-frame. If a segment starts with a GOP that does not begin with an I-frame, the transcoder will not be able to recreate an original frame.
sequence, which means that the quality of a film clip will degrade.

We suggest cutting before an I-frame. This will ensure that every segment starts with a frame, containing all the information needed for decoding the frame. Some reordering of other frames may be needed before cutting, as well as converting some frames into different frame types, i.e. if there is a B-frame before an I-frame (which uses information from an I-frame and is just before the end of a segment), this B-frame should be converted to a P-frame. P-frame, as opposed to a B-frame, does not require any information from a following I-frame and this allows us to cut segments so that they can be later decoded correctly.

Frame reordering near splitting points of the video may be needed because frames are stored not in the same order as they are displayed. Every frame has two timestamps: playback timestamp (PTS) and decoding timestamp (DTS). In Figure 2.3 frames are visualised in playback order (i.e. every frame from left to right has an increasing PTS). Every B-frame requires a subsequent I or P-frame to be decoded first. Video frames are usually stored in decoding order. If a frame sequence showed in Figure 3.2 will be cut before an I-frame, a subsequent segment will have two redundant B-frames that should be in the preceding segment. For
this reason it is needed to reorder frames, i.e. put these B-frames to the preceding segment and convert the last B-frame to a P-frame. This way it won’t depend from the I-frame which will be in a subsequent segment.

3.2 Effective Segmentation

![Graph showing transcoding time dependency on a video file size](image)

(a) Transcoding times of 1 minute video files

(b) Transcoding times of 2 minutes video files

Figure 3.3: Graphs showing transcoding time dependency on a video file size

Video file have many different properties, e.g. bit-rate, spatial and temporal resolutions, codec, duration, file size. Effective segmentation should produce a number of segments from a single video file, so that it would take approximately the same amount of time to transcode each segment. In other words, it should take the ap-
approximately same amount of CPU resources to transcode any of the segments. Our hypothesis is that all the segments should share same features. Clearly, all segments will have the same bit-rate, spatial and temporal resolutions, and codec. We want to determine if video transcoding time depends on segment file size. For this reason we take a video file and divided it into 1 and 2 minute segments. We performed two transcoding operations for each of the segment. During one transcoding operation we simply changed the codec; and during the other transcoding operation we firstly reduced spatial resolution (downscaled) and the later changed the codec. Our results are plotted in Figure 3.3.

In Figure 3.3 we can clearly see that video transcoding time does not depend on a segment file size. It is more likely that a segment with a bigger file size will be transcoded slower, however this tendency is not significant and we chose not to rely on it while segmenting the videos. On the other hand, it is clear that it takes approximately the same amount of time to transcode a segments with the same duration and same (both source and target) spatial resolution. For this reason we have chosen to rely on a video duration as a main feature for comparing video transcoding time of the segments.

### 3.3 Balancing the Workload

When trying to split the work and distribute over several machines, a question arises how to balance the workload, e.g. if one of the machine becomes slow, it should not receive more work since it may become a bottleneck and increase total processing time. There are number of ways to approach this issue and we list couple of them:

- **Static.** Static load balancing techniques know in advance processing capabilities of each computer. It then later divides the workload according to this knowledge, i.e. computers with weaker processing capabilities get less work in order to finish the work at the same time as other faster computers. However, such load balancing does not manage failures or slow nodes, i.e. when for any reason a computer starts to process slower than usual, the work load will get stacked up in his local queue.

- **Acknowledgement based.** This is a simpler implementation of load balancing, where the whole workload is divided into small pieces. Whenever a computer finishes processing a single piece, it sends an acknowledgement and then later receives a new item. Such load balancing method reacts to failures and slow computers much better than static load balancing.

- **Machine learning based.** Compared to the previous load balancing methods, this one is more complex and requires additional computations. This approach takes a set of features (e.g. audio and video bit-rates, codecs, file-size, video duration) and using learning techniques forms a model [19]. This model can later be used to predict processing times and divide the workload by using such
3.3. BALANCING THE WORKLOAD

prediction. Such method can react to environmental changes (crashes, slow responses), but the reaction time is greater than using the acknowledgement based load balancing.

- **Priority queue based.** This load balancing method is very similar to acknowledgement based one. The difference is that this method does not require a central manager that pushes the workloads to participants, instead there is a priority queue (possibly distributed) that prioritize work items and participants fetch those items from the queue.

We are going to use priority queue based load balancing. This method provides all the benefits of acknowledgement based method and its algorithm is less complex, thus easier to code and maintain in the future.
Chapter 4

System Design

In this chapter we will provide an overview of a distributed video transcoding system design and reasoning that led to such proposed design.

4.1 Main Components

The distributed video transcoding system consists of three main components: reliable distributed storage (GlusterFS), reliable queue manager (ZooKeeper) and workers (see Figure 4.1). The queue manager is the essential part of the system. It stores work items that are passed to workers. There are three types of work items:

- **Split video into segments.** When a worker receives such task, it receives additional information such as where the video is located, where to put temporary files such as video segments, and where to put resulting outputs. Along with this information worker also receives transcoding profiles. These transcoding profiles describe the desired output result and provide information such as desired video or audio codec, bit-rates, resolution, frame-rate, and file format. The worker then takes the input file that is stored in a distributed storage, extracts audio, splits video into segments and puts resulting files back to the distributed storage so that other workers could reach them. The worker then later schedules the tasks of the following two types.

- **Transcode video segments.** When a worker receives this task, it receives an input video (or audio) file from the distributed storage, starts the ffmpeg tool with necessary arguments, performs transcoding of the video and transfers the resulting output back to the distributed storage.

- **Concatenate video segments.** This task tells the worker to concatenate video segments and multiplexes the resulting video with transcoded audio. Afterwards, the worker performs a clean-up.

Distributed storage is a single place where all workers can access input files, temporary files and store output files. All workers see the same snapshot of files.
stored in this storage. In other words, all workers can access files that were stored by any other worker in this storage. There is no communication between the distributed storage and the queue manager.

Workers are dedicated machines for transcoding video files. They do not store almost any files locally, but instead store them in a distributed storage. The only files stored in workers locally are intermediate transcoding results. Workers do not communicate directly with each other, but uses either the queue manager or the distributed storage for this purpose. This enables other workers to re-do the task if the worker that was executing such task failed. The queue manager is responsible for detecting worker failures and rescheduling failed tasks.

4.2 Technology

In this section we will introduce the technologies that are used in our proposed system design and we will present the reasoning why such technologies were chosen.

4.2.1 GlusterFS

GlusterFS [2] is a distributed file-system that empowers FUSE (Filesystem in Userspace) to provide an access to the data through a single mount point. As opposed to Hadoop file-system [22], GlusterFS does not have centralized meta-data server, but relies on P2P for this task. A GlusterFS storage unit is called a brick, which is a storage file system assigned to a volume. A volume is a final distributed storage unit that can be later mounted using FUSE.
GlusterFS bricks of a volume can be configured differently:

- **Distribute.** Bricks configured in this mode will distribute files along the bricks. In this mode, file-names are hashed and this hash is used to determine to which brick a file should be written.

- **Replicate.** In this mode a brick will simply replicate another brick. In other words, it provides redundancy to storage and helps to maintain availability.

- **Stripe.** Instead of distributing the whole files, bricks configured in this mode (in a similar fashion as Hadoop file-system) will split the file into pieces and distribute these pieces across the bricks. Such mode is best suited for large files.

GlusterFS provides all the necessary features for our distributed system. Since it is already used in Screen9, we therefore chose to use this file-system to satisfy our needs.

### 4.2.2 ZooKeeper

ZooKeeper [13] is a service designed for coordinating processes of distributed applications. Its goal is to provide a simple API that enables others to build more complex coordination primitives. Such coordination primitives include configuration, group membership, leader election and distributed locks.

ZooKeeper can be seen as a replicated *in-memory* database. This database is organised similarly as file systems in UNIX, i.e. elements of the database is organized in a tree. These elements are called *znodes*. Any znode can contain by default up to 1 MB of information. There are two types of znodes:

- **Regular.** Such node can be created and deleted by any client. This node can store information and have several children znodes.

- **Ephemeral.** Clients can create and delete these znodes in the same way as regular nodes. The difference is that the system will delete this znode if a session with a client that created this znode terminates (probably due to a failure).

ZooKeeper implements a leader-based atomic broadcast protocol, called Zab [15]. This protocol guarantees that all update operations are linearisable. However, this protocol requires the majority of processes to be alive. This is why it is recommended to run an odd number of servers with ZooKeeper. Along with such guarantee, ZooKeeper also guarantees FIFO client ordering, i.e. all client update operations are performed in the same order they were issued. In order to be able to recover after a node failure, all updates are forced to disk media before being applied to the in-memory database.
ZooKeeper provides a watch mechanism. Any ZooKeeper client may watch any znode. When an update on the znode is issued (and update is performed by the same or other client), ZooKeeper notifies a client by calling a function (or a method) in a client’s code. This fault-tolerant kernel that contains of watch mechanism, znodes (regular and ephemeral) is enough to build reliable distributed system coordination primitives.

We have chosen to use ZooKeeper because of two main reasons. First, it is a fault-tolerant service, i.e. if one of the servers running ZooKeeper instance crashes, the service will still work. Second, ZooKeeper provides mechanisms to track failures of its clients. Such feature is very important for building a reliable distributed queue.

4.2.3 FFmpeg

FFmpeg project \[1\] contains a set of tools and libraries for encoding, decoding, multiplexing, demultiplexing, cropping, resizing, watermarking, and performing other similar operations for audio and video. FFmpeg project contains a tool called ffmpeg that is a general purpose video transcoder. This transcoder supports a great amount of audio/video codecs and containers. It uses a cascaded decoder-encoder architecture (see Section 2.3). This means that ffmpeg firstly demultiplexes audio/video streams, then decodes demultiplexed streams, applies filters, encodes, and finally multiplexes encoded streams to a single file or stream. Such scheme is visualised in Figure 4.2. Filters are used for resizing, cropping, scaling, changing audio volume, applying equaliser effects, etc.

![Figure 4.2: ffmpeg transcoding process](image)

The ffmpeg tool can also be used for splitting and concatenating streams. There are several ways to achieve this with ffmpeg:

- Providing a start time offset and duration. This method, however, is not accurate, i.e. it seeks to a closest key-frame and there is no way to select accurately to which frame you want to seek. After concatenating segments that were cut this way, the resulting video playback is not smooth.
• using *segment* proxy format (also referred as video container). This proxy format detects the real format from the target file name. It accepts an extra parameter which is segment length (in seconds) and tries to cut near the desired point, but just before the I-frame. It reorders and converts frames in segments so that every segment would provide smooth playback and it starts filling the next segment as soon as it finishes writing to the previous one. After concatenating segments that were cut this way, the resulting playback is smooth.

FFmpeg is an open-source general purpose video transcoder that supports many different video, audio and subtitle codecs. To our knowledge no other transcoder is as stable and supports such a great number of different codecs. Therefore, we chose to use this one for our system.

### 4.3 Queue Manager

The queue manager manages the workers workload. We have chosen to implement queue manager with ZooKeeper [13]. As mentioned in Section 4.2.2, ZooKeeper offers a reliable storage and primitives with which it is possible to build complex distributed computing control structures such as reliable and fault-tolerant queue. Using a ZooKeeper and Kazoo library [3] we have built such queue [4] with the following API:

- **put** method puts an item into the queue. Optionally, one can pass a priority argument, which is an integer. Similarly to UNIX, lower number value means higher priority.

- **put_all** method takes a list of entries ant puts them to the queue with the same specified priority.

- **get** method takes an item from the queue, but instead of removing it, locks it by creating an ephemeral znode. This znode tells other participants that this queue item is currently being processed and other item should be taken instead. If a current participant calls get for the second time before calling consume, the method will return the same item as the previous get call. If there are no items available, the method will block. Optionally, an argument can be passed telling the maximum amount of seconds to wait for an item.

- **consume** method removes an item, which is currently being processed and which was retrieved with the get method, from the queue. This method returns false if the connection with ZooKeeper gets lost and a participant no longer holds the lock of this item.

- **holds_lock** method checks if a participant still holds a lock of this item.
The most complex method of this API is `get`. This method is responsible for filtering locked items, locking and fetching items. The behaviour of this method is visualised in Figure 4.3. When this method is called, it firstly checks if a caller has called this method before, and has not yet called the method `consume`, i.e. the caller is currently processing an item (holds a lock of the item). If the caller is processing the item then the call will simply return that item. Otherwise, the method will create a closure of an `event` object and a function `call for updates`. This function `call for updates` gets firstly called by the method `get`. The closure can be later called at any time by another thread managed by ZooKeeper client when watch is triggered. In order to avoid race conditions, methods `get` and `call for updates` are synchronised with a lock. After calling `call for updates` function, method `get` simply blocks by calling a method of the event object and then waits for events: item fetched or time-out. When a time-out event occurs, method simply returns `None`; when `check for updates` function retrieves an item then method `get` sets the cancel flag. Since there is no way to remove ZooKeeper watches, this cancel flag is needed to stop executing `check for updates` closure in the future.

As mentioned earlier, the function `check for updates` is called when watch is triggered either by the method `get` or by the ZooKeeper client. Once called it fetches all items and locked items from the ZooKeeper. Then it calculates the difference between these two lists so that the resulting list would contain all available items. It afterwards tries to lock the item by creating ephemeral node under the
special znode. If it succeeds it then fetches the content of the item and notifies the method `get` about it.

**Herd Effect**

This queue implementation is not prone to herd effect. If there are many clients waiting for a new available item, once such item appears, they all will be notified by ZooKeeper watch mechanism and they will all try to get that item even though only one client can obtain this item. One way to avoid such herd effect is to use locking mechanism that is provided with Kazoo library. Listing 4.1 provides an example on how to use this locking mechanism with a reliable queue. With such mechanism only a single client at a time is able to fetch items from the queue.

```
Listing 4.1: Queue implementation without herd effect
1 from kazoo.client import KazooClient
2 3 stopped = false
4 client = KazooClient(hosts="localhost:2181")
5 lock = client.Lock()
6 queue = client.LockingQueue()
7
8 while not stopped:
9     lock.acquire() # this method blocks
10    try:
11        # Try to fetch the item, but give up trying after 2
12          # seconds
13        item = queue.get(2)
14    finally:
15        lock.release()
16        if not item is None:
17            process(item)
18            queue.consume()
```

In order to determine if it is beneficial to use a lock with the queue we performed some tests. For this we used three servers hosted in UpCloud [5]. All servers had one 2 GHz CPU, 4096 MB of RAM and were connected to 1 Gb/s network. One server was dedicated for running ZooKeeper service, the rest were used for running workloads. The workload is divided into two roles: producer and consumer. Producer tries to produce items as fast as possible, whereas consumer tries to fetch all items as fast as possible. We ran such test with different three queue implementations:

- **fault-tolerant without lock**, our queue implementation that provides fault-tolerance but does not use locking mechanism, i.e. is prone to herd effect;

- **fault-tolerant with lock**, our queue implementation used with locking mechanism
simple, simple queue mechanism that comes with Kazoo library and that does not provide fault-tolerance.

The results of our tests are visualised in Figure 4.4. In this figure we show throughput and average latency as the number of concurrent request increases. When number of concurrent requests is low, queue with locking mechanisms has lower throughput and higher latency than the one without locking. However, as the number of concurrent requests increases, the throughput of the queue without locking degrades, whereas throughput of the queue with locking stays at about the same level. Also the latency of queue without locking becomes higher than the latency of the queue with locking. This happens because the locking mechanism reduces the amount of requests, i.e. ZooKeeper has to send less watch events and less consumers will request for an item, and this helps to maintain the throughput.
In Figure 4.5 a histogram of request latencies, when there are 24 producers and 48 consumers, is plotted. This figure shows that participants of a queue without lock may wait for an item for 10 seconds, whereas with lock no participant waited for an item longer than 3 seconds. This means that lock does not only help to stabilize throughput, but also distributes items in a more fair fashion, i.e. all nodes will receive approximately the same amount of items. On the other hand, when a queue without lock is used, some of the items are delivered faster as compared to a queue with lock. Lock definitely adds extra complexity, which increases the latency; however, video transcoding takes minutes, therefore it is acceptable to have such latency. Moreover, these are results from the stress testing, video transcoding systems do not receive such a high number of requests.

![Histogram of request latencies](image1)

![Histogram of request latencies](image2)

**Figure 4.5:** Latency spreads for different fault-tolerant queue implementations
4.3.1 Worker

A worker is a machine dedicated for transcoding video segments. It has a GlusterFS volume mounted in its system in order to access input files and store intermediate and final results. Once started, a worker connects to a ZooKeeper and registers by creating an ephemeral node. Then it checks if there is something in the queue, and if not, it will wait until the queue notifies him. When the item appears in the queue, the worker fetches the item, de-serialises it, and executes it. Once executed, the worker calls a consume method (see Section 4.3) and then try to fetch another item from the queue.

Figure 4.6: Class diagram of the worker

Figure 4.6 shows a class diagram of the worker. Class ZkManager is responsible for managing connection with ZooKeeper. It connects to ZooKeeper and fetches items from the queue. ZkManager consists of two objects, i.e. LockingQueue and Lock types. The former object type is described in Section 4.3. It ensures the fault-tolerance of the system. The latter object type is used in order to remove the herd effect (as described in Section 4.3). ZkManager can schedule and retrieve items from the queue. A worker usually only retrieves items, other services may use these objects to schedule new videos to be transcoded.

Each item of the queue is a serialised class that extends class Command. Class Command has the only method called execute. There are three subclasses of a class Command: SplitAndSchedule, Transcode, ConcatMuxClean. These subclasses correspond to work item types described in Section 4.3. Command
ConcatMuxClean always gets the highest priority, i.e. if there is such task it is always taken first, whereas SplitAndSchedule always gets the lowest priority, meaning that this task will not be executed if there are other tasks that were not taken.

An execute method of a class SplitAndSchedule fetches a file stored in a GlusterFS through a mount point in his local file tree. The tool ffmpeg is later called, which demultiplex video, audio and subtitle streams from the input file, then segments the video into smaller segments, stores everything in the same mount point, and the segments get uploaded to the distributed storage. Segmenting is performed by a segment ffmpeg format. This format requires additional parameter which is segment duration. When a segment reaches this duration an ffmpeg tool finds the next key-frame and cuts the video before that key-frame. This tool performs all the necessary adjustments so that the segments can be later fully decoded. All operations with an ffmpeg are performed using Video class. This class is an ffmpeg wrapper that provides all the necessary functionality to other classes.

A Transcode class contains a TranscodingContext field. This field stores a transcoding profile or the properties of a desired output video file. When an execute method of a class Transcode is called, it starts an ffmpeg (using Video class) and passes all the necessary parameters so that the output video file would have the properties as defined in a TranscodingContext.

ConcatMuxClean command simply concatenates the video into a single piece, multiplexes audio, video and subtitle streams and performs all necessary clean-up operations.
Chapter 5

Evaluation

In order to evaluate our system we have deployed all our services in a cloud service called Upcloud [5]. We have used the following configuration:

- We set-up three servers for ZooKeeper, each of them had one 2 GHz CPU, 4 GB RAM, and 10 GB HDD drive;

- GlusterFS distributed storage was set-up on two computers, each of them had one 2 GHz CPU, 1 GB of RAM, and 10 GB SSD drive. We have configured GlusterFS in a distribute mode (see Section 4.2.1);

- Each worker machine had one 2 GHz CPU, 1 GB of RAM, and 10 GB HDD drive. We ran tests with up to 16 worker machines.

All servers were connected to 1 Gb/s network line.

As our test subjects we have chosen two video clips and each of those clips were transcoded to two different target video files, resulting four video files. In Listing 5.1 and 5.2 we show the output of an `ffprobe` tool that lists detailed information about our two test videos.

```
<table>
<thead>
<tr>
<th>Listing 5.1: Detailed information about Video 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Duration: 00:01:20.23, start: 0.540000, bitrate: 5953 kb/s</td>
</tr>
<tr>
<td>2 Stream 0:0[0x1e0]: Video: mpeg2video (Main), yuv420p,</td>
</tr>
<tr>
<td>→ 720x576 [SAR 64:45 DAR 16:9], 25 fps, 25 tbr, 90k</td>
</tr>
<tr>
<td>→ tbn, 50 tbc</td>
</tr>
<tr>
<td>3 Stream 0:1[0x1c0]: Audio: mp2, 48000 Hz, stereo, s16p, 224</td>
</tr>
<tr>
<td>→ kb/s</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Listing 5.2: Detailed information about Video 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Duration: 00:05:00.00, start: 0.000000, bitrate: 2723 kb/s</td>
</tr>
<tr>
<td>2 Stream 0:0(und): Video: h264 (High) (avc1 / 0x31637661),</td>
</tr>
<tr>
<td>→ yuv420p, 1920x818 [SAR 1:1 DAR 960:409], 2587 kb/s,</td>
</tr>
<tr>
<td>→ 24 fps, 24 tbr, 16k tbn, 48 tbc</td>
</tr>
</tbody>
</table>
```
Each video file was transcoded into two different output video files. Both outputs were encoded with H.264 video codec using 2 passes and AAC audio codec, but the first video output had $854 \times 480$ spatial resolution and the second had $640 \times 360$. During a single test session we queued to transcode all two videos using all profiles and collected network data transfer ratios, CPU usage, and events, such as timestamps of a beginning and end of each command. We have repeated this test session with 1, 2, 4, 8, 12 and 16 computers. It is worth mentioning that during each session we have demultiplexed audio (in order to be transcoded separately from the video); and the number of segments we divided the video stream was always equal to the number of workers. In Figure 5.1 we show after how much time from the beginning of each test session outputs were available (or were fully transcoded).

![Figure 5.1: Transcoding times with different number of workers](image)

It is clear from Figure 5.1 that as the number of workers doubles, availability times reduces approximately two times. In order to determine what is the actual speed-up of each transcoding operation, we gathered how much time did it take to perform each transcoding operation by subtracting two timestamps: the end of a concatenate operation and the beginning of a split operation. We divided these time variables with the time that shows how much does it take to perform a transcoding operation on a single machine with simply calling `ffmpeg` command and not using our system (it takes 2467 seconds). We have plotted these results in Figure 5.2. We can see that our system does not add big overhead when there is a single worker, i.e. the ratio is almost one. It means that transcoding operation will take approximately the same amount of time when executing it using our system or just simply launching `ffmpeg`. Speed-up continuously grows, and our test results show that with 16 machines you can get videos transcoded from 10 up to 15 times faster than compared to transcoding times on a single machine.
We wanted to determine if network can be a bottleneck in our system. For this reason we have measured the network ratio all the time. In Figure [5.3] we visualise network usage of GlusterFS nodes and all worker nodes. In this figure we can see an increased bandwidth at the beginning in gfs1 node. It is clear that our two video files were stored in a single node. The segmented videos were split and later stored on both GlusterFS nodes. At around 45th second there is a much smaller bandwidth increase, at this time all the nodes downloads video pieces in order to transcode using a second profile. There are other two noticeable spikes at around 220th and 280th seconds. These spikes appeared during the concatenation of a larger video. We can see that the bandwidth limit (1 Gb/s or 131072 kB/s) was never reached.

We have been measuring CPU usage during our test sessions. In Figure [5.4] CPU usage of a single machine during video transcoding test with 8 workers is plotted. As we can see almost all the time CPU usage was close to 100%.

In Figure [5.5] shows how much time on average the CPUs of all workers were busy during the whole session. As we can see the workload was quite balanced. We were however concerned why CPU usage was around 70%, which is less than we were expecting. For this reason we have plotted a detailed time-line with all events that occurred. Such information is visualised in Figure [5.6] In this figure you can see how long did it take to segment, transcode and concatenate each video on every node. We can see that the videos were segmented not as great as expected, some workers had higher workload and other workers had to wait for some other workers to finish the transcoding.
Figure 5.3: Network usage when during the transcoding session with 8 workers
Figure 5.4: CPU usage of one worker during the transcoding session with 8 workers

Figure 5.5: CPU usage of all machines during the transcoding session with 8 workers
Figure 5.6: Time-line of transcoding session with 8 workers

Figure 5.7: Transcoding speed-up of a larger (21 min) video
In order to find out how our system works with larger files, we performed another scalability test. We used a video described in Listing 5.3 and transcoded to H.264 video codec using 2 passes, AAC audio codec and target resolution 854 × 364. Firstly, we transcoded it on a single machine by simply launching ffmpeg and measured time (it took 43 minutes to perform this operation). We used this time as a reference. We later performed the test on our system using multiple machines. We divided all times with the reference time and then plotted the result in Figure 5.8. The graph shows that with 16 workers our system transcoded the video 15 times faster as compared to a transcoding time on a single machine without using our system. Figure 5.8 shows an overall CPU usage during the transcoding session with 8 worker machines. It is clear that the workload was balanced pretty well and CPU was busy around 90% of a time, which is a better result than the one with a smaller video files.

Listing 5.3: Detailed information about Video 3

```
1 Duration: 00:21:38.00, start: 0.000000, bitrate: 1038 kb/s
2 Stream 0:0(und): Video: h264 (High) (avc1 / 0x31637661), ↪ yuv420p, 1280x720, 1032 kb/s, 29.97 fps, 29.97 tbr, ↪ 60k tbn, 59.94 tbc
3 Stream 0:1(und): Audio: aac (mp4a / 0x6134706D), 44100 Hz, ↪ stereo, fltp, 144 kb/s
```

During the final test we wanted to approximately determine what is an overhead for using our system compared to simply launching an ffmpeg tool. Our system along with transcoding also performs segmenting and concatenating. Using one worker machine we segmented the video (described in Listing 5.2) located in GlusterFS into different number of segments and later concatenated it back into a
single piece and measured the times. Figure 5.9 shows how much time it takes to segment a video to $n$ pieces and how much time it takes to concatenate $n$ segments into a single video.

Figure 5.9: Segmentation and concatenation times
Chapter 6

Discussion

From the previous chapter there is a number of topics worth attention: scalability and workload balancing, overhead, and fault-tolerance. In this section we are going to make a discussion about these topics.

6.1 Scalability and workload balancing

Our performance test results showed that our system is scalable. 16 workers can transcode the video approximately 15 times faster. However, as we can see, some videos did not show such good results. These testing videos were not very long (less than 5 minutes), thus it is not easy to cut videos into equal segments with ffmpeg because it firstly writes frames to a separate file until the desired duration of a file is reached, then it continues writing until it finds a I-frame and then cuts it at that position. It is clear that the lower number of I-frames are available in the video stream, the worse our algorithm will perform. Performance tests done with a longer video proves this statement. In this video the number of I-frames were higher than in smaller videos, therefore the average CPU usage was higher as well.

During our tests we separated audio file from video and transcoded audio separately without segmenting it. And we segmented the video into the same number of segments as the number of available workers (or workers that has not failed). Segmenting into higher number of pieces will add a little overhead and also improve workload balancing. It is faster to transcode smaller pieces with multiple machines as a larger piece with a single machine. However, sometimes it is not possible to efficiently segment video into smaller pieces because of an I-frame issue we discussed earlier.

This leads us to a conclusion that it may make sense to write our own segmenting tool that writes a number of frames without transcoding and when decodes a small number of frames near the cutting point in order to generate an I-frame for the next segment. We have not done any testing with this but it should to provide desirable results.
6.2 Overhead

Our system adds overhead to a transcoding process. This happens because along with transcoding our system has to segment, concatenate and transfer videos. In our performance test we have shown how much time it takes to transcode and segment videos. This test includes the transfer times from GlusterFS storage as everything was stored in that storage; segmenting and concatenating were performed on different machines. It is clear that this overhead is not big and does not do big impact on our scalability result.

6.3 Fault-tolerance

Our system is fault-tolerant and this is achieved by using fault-tolerant subsystems. In this section we are going discuss what kind of faults there may occur and how system will react to them.

There are three main parts of the system; naturally, fault can appear in any of these parts: distributed storage, distributed queue and workers. Also faults may occur in communication between these systems.

**Fault in a Distributed Storage**  If fault appears in distributed storage (GlusterFS), the way the system reacts depends on configuration of such distributed storage. If there are no nodes dedicated for replicating data and one of the servers crashes the data that was stored in this server will be lost. The executing tasks will probably fail, the subsequent tasks will however complete successfully if no other failures occur. If a server that crashed has a replica, the system will continue functioning normally.

**Fault in a Distributed Queue**  Distributed queue is based on ZooKeeper. This means that the same fault-tolerance properties apply to our subsystem. In order for ZooKeeper to function a majority of configured severs must be on-line, e.g. if we have 3 servers, up to 1 server can crash in order for system to continue functioning normally. If more servers crash, the system will stop functioning as writes to the queue will be prohibited.

**Worker fault**  Only one worker must be on-line in order for system to be functioning. Workers are tracked by ZooKeeper. When a ZooKeeper detects that a worker has crashed it simply puts back its task to the queue. All input files are stored in a distributed storage and all input parameters are stored in a queue. This means that only the intermediate state will be lost if a worker crashes and this intermediate state will be recreated in other worker once it takes the re-enqueued item.

**Connection faults**  If a connection between ZooKeeper and worker fails, it will be considered as a crash. Both ZooKeeper and worker will detect this state. ZooKeeper
will re-enqueue item and worker will stop performing commands until the connection gets restored.

If a connection between worker and GlusterFS fails, a worker will return an item to the queue and try to connect to other GlusterFS server. It will continue trying to reconnect to servers and will continue working afterwards by taking a next task from the queue.

There is no direct connection between workers, or between GlusterFS and ZooKeeper.
Chapter 7

Conclusions

In this project we have proposed a system design for distributed video transcoding. We have shown that this system design is scalable and fault-tolerant. In our design there are three different commands: segment, transcode, and concatenate the video. Each of this command is executed by any worker, i.e. all workers have equal responsibilities.

Decoded video frames do not depend on other frames. This makes it easy to segment the video. However, it takes a huge amount of time to transfer decoded video across the network. Therefore, we used an ffmpeg tool to perform segmenting on coded video. This tool segments the video in such a way so that it would be possible to correctly decode any segment.

During the project we have also developed a reliable distributed queue using ZooKeeper for our system. We performed tests and showed that it is more efficient to allow only one client to access a queue at a time. This way the herd effect is avoided resulting a better throughput and lower average latency.

Our performance test results show that with 16 worker machines it is possible to transcode video files up to 15 times faster. Results also show that our design can be improved and we have discussed what can be done in order to improve it. The most promising improvement is to have a better segmentation algorithm. With a more effective segmentation algorithm it would be possible to distribute the workload in a more even way and decrease the total transcoding time.

In conclusion, this report shows promising results to scaling video transcoding and thus reducing total transcoding time. With speeding-up video transcoding, video content in the Internet will have higher quality and will be adapted to bigger amount of devices.
Bibliography


