Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Timo Saarinen

Container-based video processing

Master's Thesis
Helsinki, May 6, 2015

Supervisor: Professor Eljas Soisalon-Soininen, Aalto University
Instructor: Lasse Pajunen M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Timo Saarinen

Title:
Container-based video processing

Date: May 6, 2015 **Pages:** vii + 53

Major: Software Technology **Code:** T-106

Supervisor: Professor Eljas Soisalon-Soininen

Instructor: Lasse Pajunen M.Sc. (Tech.)

In recent years, the development and proliferation of mobile devices and the increasing speed of data communication have accelerated the rapid growth of video creation and consumption. Social media, for instance, has embraced video as its essential part.

However, different devices and platforms with various screen resolutions, video format capabilities and data communication speeds have created new challenges for video transcoding systems. Especially, system scalability is an important aspect to ensure a proper user experience for end-users by maintaining a high rate of overall transcoding speed despite usage peaks and fluctuating system load.

One way to build a scalable, rapidly deployable video transcoding service is to wrap transcoding instances into lightweight, portable containers, virtualized at the operating system level. Since containers share the kernel of the host operating system, new instances can be quickly launched when necessary.

First, this thesis discusses Linux container technology, its main derivatives and related tools. Furthermore, this thesis describes various utilities that facilitate the orchestration of Linux containers but also typical video processing and internet video technologies are introduced.

In order to investigate the advantages of using containers, we implemented a video transcoding service that uses application containers virtualized in CoreOS operating systems. The transcoding service is run on Amazon EC2 (Elastic Compute Cloud) instances. In addition to evaluating the service in terms of functionality, the thesis also discusses the strengths and weaknesses of the development process and use of container technologies within the scope of this project.

Keywords: Operating-system-level virtualization, Docker container, video processing, distributed system, container orchestration, cloud

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Timo Saarinen		
Työn nimi:	Kontteihin perustuva videoprosessointi		
Päiväys:	6. toukokuuta 2015	**Sivumäärä:**	vii + 53
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Eljas Soisalon-Soininen		
Ohjaaja:	Diplomi-insinööri Lasse Pajunen		

Viime vuosina mobiililaitteiden kehittyminen ja nopea leviäminen sekä nopeutuvat tietoliikenneyhteydet ovat kiihdyttäneet videoiden luonnin ja kulutuksen ripeää kasvua. Videosta on tullut olennainen osa sosiaalista mediaa.

Erilaiset laitteet ja alustat vaihtelevilla näyttöresoluutioilla, videoformaattituilla sekä tietoliikenneyhteyksien nopeuksilla ovat kuitenkin luoneet uusia haasteita videoiden prosessointiin. Erityisesti skaalautuvuus on olennainen aspekti yritettäessä varmistaa loppukäyttäjille asianmukainen käyttökokemus ylläpitämällä korkeaa prosessointinopeutta huolimatta käyttöpiikeistä ja vaihtelevasta systeemin kuormituksesta.

Eräs tapa rakentaa skaalautuva, ripeästi käyttöönotettava videoiden prosessointipalvelu on paketoida prosessointi-instanssit kevytrakenteisiin, helposti liikuteltaviin kontteihin, jotka virtualisoidaan käyttöjärjestelmätasolla. Koska kontit käyttävät samaa käyttöjärjestelmän ydintä, uusia instansseja voidaan luoda tarpeen vaatiessa hyvin nopeasti.

Tässä työssä esitellään Linux-kontteja ja joitakin sen johdannaisia sekä aiheeseen liittyviä työkaluja. Lisäksi erilaisia konttien orkestrointia helpottavia apuohjelmia käydään läpi, kuten myös videoprosessoinnin peruskäsitteitä ja Internetissä käytettyjä videoteknologioita.

Tutkiaksemme konttien käytöstä saatavia hyötyjä toteutettiin videoiden prosessointipalvelu, joka käyttää CoreOS-käyttöjärjestelmän päälle virtualisoituja sovelluskontteja. Se rakennetaan Amazonin EC2-instanssien päälle. Palvelua ei arvioida ainoastaan toiminnallisuuden kannalta, vaan myös kehittämisvaiheen sekä konttien käytön hyviä ja huonoja puolia käsitellään.

Asiasanat:	Käyttöjärjestelmätason virtualisointi, Docker-kontti, videoprosessointi, hajautettu järjestelmä, konttien orkestrointi, pilvi
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor, Professor Eljas Soisalon-Soininen, for the valuable comments and instructions. His warm and encouraging way of giving advice has kept me motivated throughout this project.

I would also like to thank my instructor, M.Sc. Lasse Pajunen, for introducing me this interesting topic and sharing his knowledge by giving useful tips. He really knows what he talks about.

In addition, I would like to thank my employer, Dream Broker Ltd, for providing a good environment with friendly atmosphere. The company, as well as this project, has helped me to develop as a professional.

Finally, I would like to thank my lovely wife Annika for her support. It is wonderful to start every day with you.

Helsinki, May 6, 2015

Timo Saarinen

Abbreviations and Acronyms

AuFS	Advanced Multi-Layered Unification Filesystem
CAP	Consistency, Availability, Partition-tolerance
DASH	Dynamic Adaptive Streaming over HTTP
DNS	Domain Name System
EC2	Elastic Compute Cloud
HAS	HTTP-based Adaptive streaming
HDS	HTTP Dynamic Streaming
HLS	HTTP Live Streaming
HSS	HTTP Smooth Streaming
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
LXC	LinuX Container
NAT	Network Address Translation
OS	Operating System
PID	Process Identifier
RTMP	Real-Time Messaging Protocol
RTSP	Real-Time Streaming Protocol
SCP	Secure Copy
SSH	Secure Shell
URL	Uniform Resource Locator
VM	Virtual Machine
VMM	Virtual Machine Monitor
VOD	Video-On-Demand
XML	Extensible Markup Language

Contents

Chapter 1

Introduction

As Internet has expanded around the world, becoming an essential tool for countless people, the requirements for web services have grown significantly. In many cases, it is no longer enough to have a single server for providing the required capabilities. Scaling up is often too expensive to provide enough computation power. Moreover, since complexity of web services has grown, there is a need for better hardware independence, availability, isolation and security.

To fulfill these needs, virtualization technologies started to improve significantly around year 2000 [47]. In 1998, VMware invented a technology to virtualize the x86 platform. Since then, this hypervisor-based virtualization has steadily grown its popularity, bringing forth software solutions also other than VMware, such as Xen and KVM. Virtual machines can be considered one of the foundations of cloud computing, which, in turn, has also become a significant part of web services' infrastructures.

However, as past studies have shown [38, 47], starting new virtual machines is relatively slow, and applications running in them suffer from overhead and impaired performance caused by the hypervisor layer between application and hardware. This has caused increased interest in finding a more lightweight solution.

In 2006, Paul Menage and Rogit Seth started to implement a feature to Linux kernel that would limit and isolate the resource usage of a collection of processes. In the following year, it was merged in Linux kernel with name Cgroups. This new feature, along with kernel namespaces, formed a basis for an operating system-level virtualization method called LXC that allows running multiple isolated Linux systems on a single host. These virtualized systems are called Linux containers, and since all of them use the same kernel in the host operating system, they can be started much faster than virtual machines. Figure 1.1 shows the main difference between hypervisor-based

and container-based virtualization: whereas the first one provides abstraction for full guest operating systems, the latter provides abstraction for the guest processes. Although there have been Linux container implementations also before, LXC was the first one to be merged into the Linux kernel, making containers a step simpler to be brought to use.

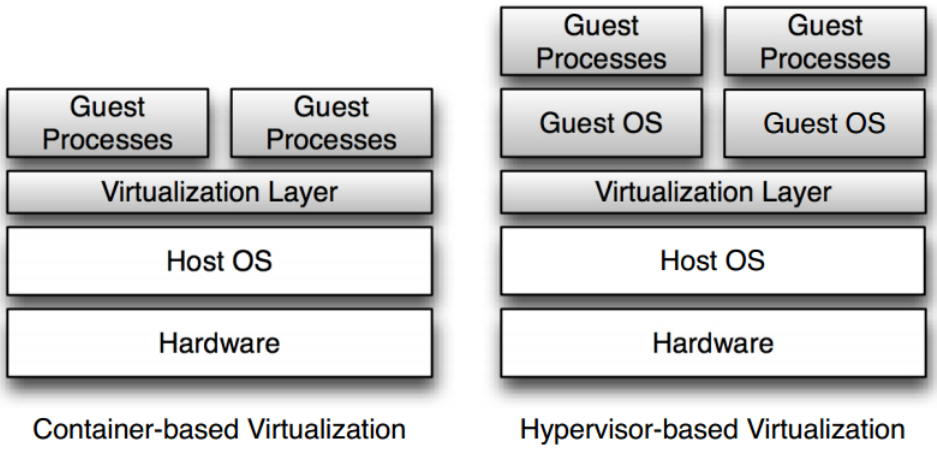


Figure 1.1: Comparison of container-based and hypervisor-based virtualization [47].

Container-based virtualization is a plausible alternative for virtual machines to fulfill the two crucial requirements when co-locating different workloads: isolation and resource control. However, only recently has it been adopted and standardized in mainstream operating systems.

Within the last couple of years, Docker, a virtualization engine based on Linux Containers (LXC) technology, has caused increased attention to Linux containers. Docker allows user to wrap single applications to containers that can be deployed on any system with Docker installed. Docker has made it relatively easy to embrace and gain benefits from LXC technology [15].

In many web services, it is important to keep the quality of the service in a good level despite variance in system load. By dividing service to multiple small, self-containing micro-services, and isolating them in a way that they affect each other minimally, the functionality and responsiveness of critical components can be kept better. This type of architecture, called micro-services architecture, also allows more efficient usage of available hardware capabilities. Furthermore, micro-services are easier to distribute across machines since they are not so tightly coupled. This is an important aspect because scaling out is typically cheaper than scaling up.

However, multiple micro-services on the same machine running without proper isolation may cause the host environment and other micro-services to affect too much their functioning and performance. On the other hand, virtual machines for each micro-service may be too heavyweight a solution.

By wrapping services to application containers, they get isolated and are minimally affected by changes in the host environment or other services. In addition, they can be copied and started relatively fast. To manage containers in a clustered environment, a range of tools have been developed, including CoreOS and Kubernetes. Quite recently, Amazon added a new service for running containers easier on Elastic Compute Cloud (EC2) instances.

In this work, a video transcoding service is implemented. It receives video files, transcodes them with different quality settings and outputs new video files. The system will consist of several micro-services that are wrapped to Docker containers. Not only the result itself but also the development phase will be evaluated and interesting findings shared.

This document is divided in the following chapters: Chapter 2: *Video Processing* introduces the main internet video technologies, key concepts of video transcoding and some open source video processing tools. Chapter 3: *Linux Containers* discusses the technologies used in Linux containers, differences of container-based and hypervisor-based virtualization and introduces some application container engines. Chapter 4: *Container orchestration* describes the characteristics of distributed systems, micro-services architecture and management of application containers in a cluster environment. Chapter 5: *Implementation* describes the requirements and the implementation phase of the video transcoding service explained briefly in the previous paragraph. Chapter 6: *Evaluation* focuses on the evaluation of the implemented transcoding service and gives some insights about the strengths and weaknesses of using container technologies.

Chapter 2

Video streaming and processing

The mobile Internet and use of Internet Protocol (IP) based videos are growing rapidly. According to Cisco's researches [1, 2], the mobile data traffic in 2014 was nearly 30 times the size of the entire global Internet in 2000. Furthermore, mobile video traffic exceeded half of the total mobile data traffic for the first time in 2012. The online video is not growing only in the mobile Internet. In 2013, IP video traffic was 66 percent of all consumer Internet traffic, and according to Cisco's forecast, the percentage will grow to 79 in 2018. Moreover, it is quite common today to watch IP-based videos on TV.

As there are web browsers and devices with different resolutions, playback capabilities and data communication speeds, a smooth user experience in watching online videos is not taken for granted. However, various technologies have been developed over time to improve user experience and cost-efficiency. These will be discussed in section 2.1.

Video transcoding is one of the core concepts when processing video. When converting a video from a format into another, or its bit rate has to be reduced to adapt better to a channel bandwidth, transcoding is needed. The principle and the reason for transcoding are discussed in more detail in section 2.2.

Handling a large amount of video content requires video processing tools—especially tools that provide a programmable interface or tools that can be integrated relatively easily to be a part of other programs. Section 2.3 introduces some open source tools that provide a command line interface.

2.1 Internet video technologies

Video has become quite common on web pages. As a research suggests [35], the user experience of watching video plays an important role. For example,

if the video start-up time is long, if there is plenty of buffering or if the video player does not adapt properly to the network connection bit rate, the user may stop watching and move on.

As Internet has evolved, different video streaming technologies have been developed, both for live streaming and video on demand (VOD). Basically they can be divided to three classes: stateful real-time streaming, progressive download and adaptive streaming. These are discussed in the following subsections.

2.1.1 Stateful real-time streaming

Traditionally, real-time video streaming has been implemented using stateful proprietary stream protocols, such as Real-Time Streaming Protocol (RTSP) and Real-Time Messaging Protocol (RTMP) [25, 49]. *Stateful real-time streaming* refers to real-time streaming in which the server keeps track of the state of its clients until they disconnect. During the session, the client communicates with the server by issuing various commands, such as *PLAY*, *PAUSE* and *TEARDOWN* in RTSP.

There are some properties that complicate the usage of these protocols [25, 49]. Firstly, they need a specialized streaming server. These servers may be costly to set up and maintain. Secondly, since these protocols are primarily based on User Datagram Protocol (UDP) and they do not use Hypertext Transfer Protocol (HTTP), there may be issues with firewalls, Network Address Translation (NAT) and mobile devices. Thirdly, maintaining sessions of numerous clients can be costly.

However, adaptive streaming technology provides an alternative to stateful protocols, allowing live streaming via HTTP. The technology is discussed in subsection 2.1.3.

2.1.2 Progressive download

Progressive download is a technology in which the video can be started to play after transferring some amount of the video data to a local buffer [50]. While playing, more video data is being downloaded to the buffer in the background. Thus, the media player does not have to download the video file completely to play parts of it. The video file is served by a standard HTTP server, thus additional issues with NAT are not caused. Progressive download does not support live streaming.

Progressive download has some advantages over downloading the video as a whole [40]. Firstly, it is cost-efficient to some extend. If an end-user wants to watch only a few seconds from the beginning and then stops watching,

the media player does not necessarily use network bandwidth to fetch the whole video file. Furthermore, if the media player and the HTTP server are capable of handling special offset parameters (such as byte range HTTP requests), the end-user can skip to a certain point of the video, and the media player starts buffering from that point onwards, saving bandwidth. Secondly, progressive download provides end-user a faster user experience. Watching can be started almost right away because only a small buffer of video data has to be fetched.

However, as a disadvantage, progressive download is not bitrate adaptive [40]. Thus, fluctuations in network connection speed will not cause the player to switch to another video profile. In addition, progressive download does not support live streaming.

2.1.3 Adaptive streaming

HTTP-based Adaptive streaming (HAS), originally proposed by Move Networks company in 2006, is a technology to provide a high-quality user experience with uninterrupted video streaming under changing network conditions and heterogenous devices [41, 43, 49]. In a typical HAS method, the raw video is encoded into different bitrates and profiles (quality levels). Every video profile is then split in short segments that are 2-10 seconds long. When a user starts playing a video, a manifest file is downloaded. It contains information about different profiles available, and URLs to the corresponding video segments (see Figure 2.1). The client then chooses a profile that suits best for the connection, and changes it on-the-fly if needed.

As progressive download, adaptive streaming avoids additional NAT issues by serving the segment files from a standard HTTP server. Another advantage of adaptive streaming is that it can be used for both live streaming and VOD.

Since 2006, several HAS solutions have been widely used. They include Microsoft's HTTP Smooth Streaming (HSS), Adobe's HTTP Dynamic Streaming (HDS), Apple's HTTP Live Streaming (HLS) and, as an open source approach, Dynamic Adaptive Streaming over HTTP (DASH). Numerous media content providers, such as Microsoft, Apple, Netflix have adopted the technology [34, 42].

2.2 Fundamentals of video transcoding

The operation in which a video is converted from one format into another is called *video transcoding* [48]. A *format*, in turn, is defined by many charac-

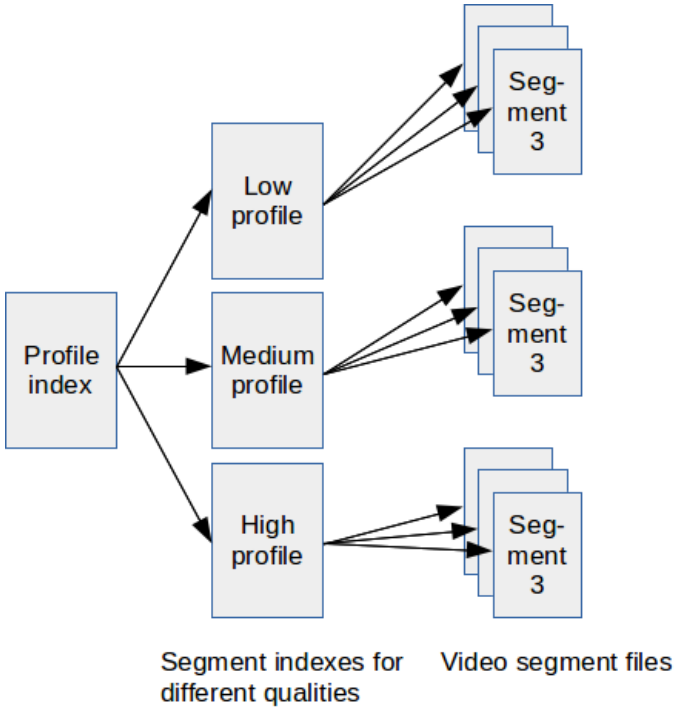


Figure 2.1: Example of video segment indexing used in adaptive streaming.

teristics of the video, such as bit rate, frame rate, spatial resolution, coding syntax and content.

The principle of a transcoder is to take in an encoded (compressed) video, decode (uncompress) it into raw format, and then encode it into another format. To speed up this computationally expensive process, transcoder may reuse the decoded video information in the encoding process. This information may include motion vectors and quantization parameters. Encoding and decoding, being crucial parts of the transcoding process, are discussed in the following subsection. In addition, reasons and objectives for transcoding are also discussed.

2.2.1 Video encoding

Video encoding, or compression, is the process of converting digital video data into a format that is suitable for transmission or storage [36]. Typically the size of the video is reduced in the process. Thus, a raw/uncompressed digital video often requires a much larger bit rate than an encoded one.

To decode/uncompress an encoded video, a complementary part is needed. It is called decoder. A *codec* is a pair consisting of an encoder and a decoder.

There are various ways to achieve data compression. First, videos typically contain statistical redundancy. *Statistical redundancy* means that the data contains repetitive information in a form or another. Those redundant parts can be expressed with less number of bits. With this method, the compression can be kept lossless. In *lossless* compression, the video is compressed in a way that it can be decoded back to the exact original data. However, lossless compression reduces the size only moderately. Second, by removing subjective redundancy, a higher ratio of compression can be achieved. *Subjective redundancy* means that there are elements in the video that can be lost in the compression without significantly affecting the viewer's perception of visual quality. This is called *lossy* compression, in which some data essential for the ability to decode video back to the original data is lost. Most video encoding methods achieve compression by exploiting temporal and spatial redundancy. *Temporal redundancy* refers to the fact that there is typically a high correlation between frames of video that are temporally near each other. *Spatial redundancy*, in turn, refers to the high correlation between pixels that are near each other.

Currently, one of the most commonly used video compression format is H.264 [36]. The standard itself being hundreds of pages long, it gives better performance than any of the preceding ones. H.264 is a lossy format.

2.2.2 Objectives of transcoding

There are various reasons for converting a video into another format [45, 48], and some of them are explained here.

First, there is bit rate reduction. Bit rate is the number of bits that are processed per unit of time. To smoothly stream a video, its bit rate at maximum should be as high as is the bandwidth. For example, the bit rate of a video stream may have to be lowered to adapt the stream to a channel bandwidth. This is one of the earliest and most important applications of transcoding. The idea of bit rate reduction is to reduce the bit rate while keeping the highest quality possible and maintaining low computation complexity.

Second, transcoding is used to improve error-resilience. Given the existence of transmission errors on channels that corrupt video quality, there is the need to make the bit stream more resilient to those errors. Variety of strategies exist to add error-resilience [46]. The bit stream structure can be affected at different levels: it can be added redundancy, or data segments can be localized to reduce error propagation, for example.

Finally, transcoding can also be used to insert new information into the video stream. It can be company logos and watermarks, for example.

2.3 Open source video processing tools

Audio and video content forms an important and expanding part of the digital collections world-wide [21]. The handling of the content requires additional tools—when dealing with large amounts of audio and video content, a robust and comprehensive tool that provides a programmable interface is indispensable.

There are various open source tools that are capable of numerous video processing operations. In this section, some the tools will be discussed: Mediainfo, a tool to display information about audio and video files, FFmpeg, a toolset for video and audio converting and editing, and MEncoder, an alternative for FFmpeg.

2.3.1 Mediainfo

Mediainfo is a relatively simple program that allows displaying information about audio and video files [9]. The information reveals many things, such as video and audio streams and their formats, codecs and durations. Mediainfo allows the information to be output in several formats, such as text, Extensible Markup Language (XML) and Hypertext Markup Language (HTML). The program can be used a complementary utility for programmatic video processing, for example. Listing 2.1 shows a portion of a sample Mediainfo output.

Listing 2.1: A portion of a sample Mediainfo output

```
General
Complete name                            : /home/tiksa/video.mp4
Format                                   : MPEG-4
Format profile                           : Base Media
Codec ID                                 : isom
File size                                : 140 KiB
Duration                                 : 7s 381ms
Overall bit rate                         : 156 Kbps
Writing application                      : Lavf56.15.102

Video
ID                                       : 1
Format                                   : AVC
Format/Info                              : Advanced Video Codec
Format profile                           : Baseline@L1.3
```

```
Format settings, CABAC                   : No
Format settings, ReFrames                : 1 frame
Codec ID                                 : avc1
Codec ID/Info                            : Advanced Video Coding
Duration                                 : 7s 367ms
Bit rate                                 : 54.9 Kbps
Width                                    : 320 pixels
Height                                   : 240 pixels
...
```

2.3.2 FFmpeg

FFmpeg project was originally started by Fabrice Bellard in 2000 [6]. Today, FFmpeg has numerous contributors and it consists of various libraries and programs, including libavcodec, libavformat and ffmpeg [16]. Library, for example, libavcodec contains encoders and decoders for different audio and video codecs. Library libavformat, in turn, contains muxers and demuxers for different multimedia formats. ffmpeg is a command line tool for conversing and manipulating multimedia files.

The usage of ffmpeg is that it is given an arbitrary number of files (or pipes, network streams or grabbing devices, for example) as input, and then it outputs an arbitrary number of files with desired conversion [16]. Each input file is demuxed to encoded data packets, and then decoded to raw frames. If there are multiple input streams, ffmpeg tries to synchronize them according to their timestamps. After decoding input streams, various filters can be applied. They include overlay images or videos, fading, padding, concatenation and trimming, for example. In addition to filters, streams can be modified also in many other ways, such as changing bit rate or aspect ratio. After processing raw frames, ffmpeg encodes them, and then uses a muxer to wrap the output streams to a video format. Figure 2.2 shows the phases in ffmpeg processing, simplified. ffmpeg provides a typical command line interface to process videos. As examples, see the following commands.

```
ffmpeg -i in.mp4 -r 24 out.avi

ffmpeg -i in.mp4 -i overlay.mp4 -filter_complex
    '[1]setpts=PTS-STARTPTS,scale=100x100[scaled];
    [0:v][scaled]overlay=x=10:y=10' out.mp4
```

The first command takes an MP4-formatted file as an input and outputs an AVI-formatted file with frame rate of 24. The second one takes two video files as input, scales the overlay video and adjusts its time stamp, and outputs the first input file overlaid by the second one.

Figure 2.2: Phases in processing videos with ffmpeg, simplified.

2.3.3 MEncoder

MEncoder is command line tool for encoding and transcoding audio and video files [5, 8]. Its key concepts are quite same than ffmpeg's. Actually MEncoder uses some parts of ffmpeg, such as libavcodec. MEncoder and ffmpeg could be considered as alternatives to each other.

As an example command, the following one takes an MP4-formatted file as an input and outputs an AVI formatted file using mp3lame as audio codec and H.264 as video codec.

```
mencoder in.mp4 -o out.avi -oac mp3lame -ovc x264
```

MEncoder is included in the same distribution with MPlayer, a media player. They are built in a way that MEncoder can process any video format that MPlayer understands. In addition, MEncoder and MPlayer can be used interactively to simulate filters in real-time. Various key bindings to can be defined to modify filters on the fly. For example, the following configuration binds arrow keys to actions that change x and y parameters by -5 or 5 of rectangle filter.

```
RIGHT change_rectangle 2  5
LEFT  change_rectangle 2 -5
UP    change_rectangle 3 -5
DOWN  change_rectangle 3  5
```

The actual command to start simulation can be the following:

```
mplayer -vf rectangle -input conf=crop in.mp4
```

Now the MPlayer starts and the crop filter can be adjusted to the desired place with arrow keys. The corresponding parameters can be seen in the standard output. Then they can be used in the actual MEncoder command. Many, but not all, filters are supported by this feature. Filters that are computationally expensive cannot be simulated in real-time.

2.4 Conclusion

In this chapter, many things related to video processing and streaming in the internet were discussed. It was noticed that the trend is towards HTTP-

based, adaptive streaming technologies from complicated streaming server solutions. Numerous large organizations, including Netflix, have introduced adaptive streaming technologies in their services. Morever, because videos are watched in various environments with different devices, networks and bandwidths, it is important to serve them in correct format and with appropriate bit rate. This is where transcoding is a crucial concept. Balancing between things such as video quality, compression complexity and bit rate is necessary. Finally, some useful open-source tools were introduced that provide an environment to process videos without having in-depth knowledge about video processing.

Chapter 3

Linux containers

Various virtualization technologies have existed for many decades. Around the turn of the millennium, hypervisor-based virtualization, in which the whole machine is virtualized, started to grow its popularity. Despite the performance penalty caused by the virtualization layer, it has been a useful technology for software engineers. Another virtualization technology, operating-system-level virtualization, started to gain attention some years ago. Being a more lightweight solution by virtualizing only the operating system, and after getting supported by the mainline Linux kernel, it became an alternative to the hypervisor-based virtualization in some cases. Docker, being a derivative of the operating-system-level virtualization technologies, provides an engine for running lightweight, isolated application containers that open up new opportunities software engineering.

In this chapter, the history and main properties of both virtualization technologies are discussed. Furthermore, they are compared from different perspectives. Finally, Docker and its key concepts are introduced.

3.1 Operating-system-level virtualization

Traditionally, an operating system (OS) allows one user space instance. However, *operating-system-level virtualization* is a technique that makes it possible for the kernel to allow multiple isolated user space instances [39, 47]. The isolated instances are called *containers*. In addition to isolation, operating-system-level virtualization methods often provide resource management features.

Various implementations for OS-level virtualization exist, the first ones being developed around 2000. The implementations include Virtuozzo (2001), Linux-VServer (2001), OpenVZ (2005) and LXC (2008). They use different

mechanisms for virtualizing, and they have mainly required a patch to kernel. However, in 2007, Cgroups functionality was merged into kernel, which has made it possible to implement OS-level virtualization mechanisms without patching kernel.

3.1.1 Control Groups

Control Groups (Cgroups) is a mechanism in Linux kernel to aggregate or partition sets of tasks into hierarchical groups with specialized behaviour [33]. Basically, with Cgroups one can limit CPU, memory and disk and network I/O throughput of groups of processes and measure their usage of resources (See Figure 3.1).

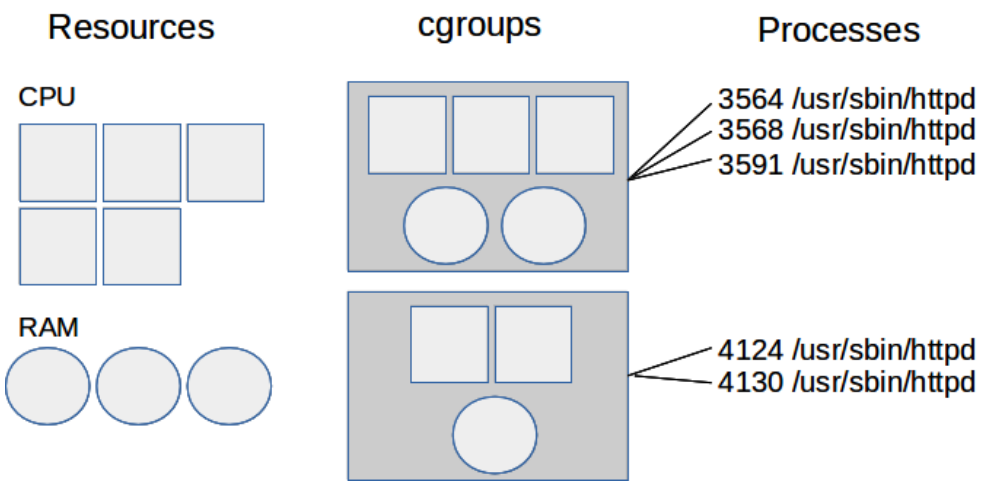


Figure 3.1: Resources shared with Cgroups

The development of Cgroups was started in 2006, mainly by Google engineers Rohit Seth and Paul Menage in 2006. It was merged into kernel in the following year, adding a good foundation for building container implementations [23]. A few years ago, however, some Linux developers felt that Cgroups is flawed in many ways and it should be rewrited [24]. In 2014, a revised version of Cgroups was merged into Linux kernel. About the same time, Linux kernel was merged a feature called namespaces. It provides a way to isolate containers so that their processes cannot see or affect other containers. Namespaces will be discussed in the following chapter.

Cgroups form a good basis for resource control when creating containers. As mentioned, Cgroups do not only enable limiting resources for containers

but also allow measuring usage of resources of each container. This feature can be used for billing purposes, for example.

3.1.2 Kernel namespaces

Kernel namespaces is a technique [31, 37] to wrap system resources in a way that they appear to the processes of certain namespace as their own, isolated instances. Currently, namescapes have support for six types of resources: process IDs, users, networks, filesystem mount points, IPC (Interprocess Communication), and UTS (UNIX Time-sharing System).

Although namespaces were not developed to strictly support only containers, it was one of the overall goals, and that is why they bring essential benefits when creating containers [47]. While Cgroups provides a basis for resource control of containers, namespaces can be used to isolate them. Thus, containers are separate instances that do not know about each other. In the context of containers, certain namespaces bring significant benefits, as shown in the following paragraphs.

PID namespaces give a unique ID number space for each container, meaning that different containers can use same IDs. This allows containers to be migrated between hosts while keeping the same process ID.

Network namespaces allow each container to have its own virtual netowork device. This is useful because it is possible to have multiple containerized web servers on the same host, all of them bound to port 80 in their network namespaces.

Mount namespaces isolate a set of filesystem mount points. In other words, containers with different mount namespaces can have a different view of the filesystem hierarchy. The `mount()` and `umount()` system calls no longer operate automatically on global mount points but only on the points associated with the mount namespace of the calling process.

User namespaces allow unique user and group ID number inside containers. Thus, the user and group ID of a process can be different inside and outside a container. Processes can be set to have root privileges inside a container, but no privileges outside.

UTS namespaces allow each container to have its own hostname and NIS (Network Information System) domain name.

IPC namespaces isolate System V IPC objects and POSIX message queues. Thus, each container has its own set of POSIX message queue filesystem and System V IPC identifiers.

3.1.3 Linux container implementations

Various OS-level virtualization systems have been developed over the years, and in this subsection three of them will be discussed: Linux-VServer, OpenVZ and LXC.

Linux-VServer was released in 2001, being one of the oldest implementations [39, 47]. To guarantee isolation, Linux-VServer built its own capabilities which could be installed through a kernel patch. In addition, it uses *chroot* mechanism to jail a file system inside a container. Because Linux-VServer isolates processes through a global PID space, processes with the same PID cannot be re-instatiated. As a drawback, the system is not able implement live migration, checkpointing and resuming. Another drawback in Linux-VServer is that network subsystems are not virtualized, which results in the lack of autonomous networking management.

OpenVZ was released in 2005 [47]. Unlike Linux-VServer, it uses kernel namespaces (see subsection 3.1.2) to provide resource isolation between containers. This enables it to implement live migration, checkpoint and resuming. Network namespaces allow OpenVZ containers to have their own network stacks. For resource control, OpenVZ has its own components, which requires an OpenVZ-patched kernel.

LXC (Linux Container) was released in 2008 [47]. As OpenVZ, LXC uses kernel namespaces to provide resource isolation between containers. The resource control is only allowed via Cgroups, which is not the case with Linux-VServer and OpenVZ. Unlike other container-based virtualization systems, LXC has an advantage of having its mainline implementation in the official kernel source code. This is possible because LXC is built on kernel namespaces and Cgroups, which, in turn, are parts of the official kernel.

3.2 Comparison to hypervisor-based virtualization

Hypervisor-based virtualization[1] is a technique to provide a virtual machine environment by simulating the underlying hardware. It provides hardware independence, availability, isolation and security [47].

The first hypervisor-based virtualization was demonstrated in 1967 with IBM's CP-40 system. However, the modern generation of virtual machines started to evolve around year 2000. In 1998, VMware, a company providing cloud and virtualization software and services, invented a technology to

[1] Often called also full virtualization.

virtualize the x86 platform.

Over the years, machine virtualization has steadily grown its popularity, bringing forth also other virtualization solutions, such as Xen and KVM. Virtual machines can be considered one of the foundations of cloud computing, which, in turn, has also become a significant part of web services' infrastructures [47].

3.2.1 Fundamentals of hypervisor-based virtualization

Hypervisor-based virtualization [14] is based on a component called Virtual Machine Monitor[2] (VMM). *Virtual machine monitor* is software which partitions physical servers in multiple complete virtual machines (VM). *Virtual machine*, in the context of hypervisor-based virtualization, is an emulation of a computer system containing the capability required for running an OS. VMM, in turn, is responsible for managing system resources such as CPU, RAM and disk and allocating them to different VMs.[3]

There are two types of VMMs [14, 47]. The first type, being the more common one, is used in the autonomous architecture: VMM is installed directly above the hardware. Therefore, there is not an actual host OS between hardware and VMs. The second type of VMM is used in the hosted architecture, where the VMM runs as an application on the host OS and manages VMs on it. The first type is naturally more efficient since the VMM has direct access to hardware.

VMs use a technique known as binary translation to run software [14]. As software contains privileged and non-privileged instructions, the VMM has to ensure that VMs do not run critical instructions that would change the state of the host machine. Non-privileged instructions can be run with native performance, but privileged instructions must be translated to different instructions that can be run safely. This naturally reduces performance.

3.2.2 Main differences

Hypervisor-based virtualization and OS-level virtualization differ from each other in many ways. The main difference is that in hypervisor-based virtualization, a machine capable of running an OS is virtualized, and in OS-level virtualization, an OS is virtualized. From guest OS user's perspective they may seem similar at first glance. Figure 3.2 gives an overview of the structures of some different virtualization techniques.

[2]Often called also as hypervisor.

[3]There is also a technique called paravirtualization that modifies the kernel of VMs for better performance in the virtual environment.

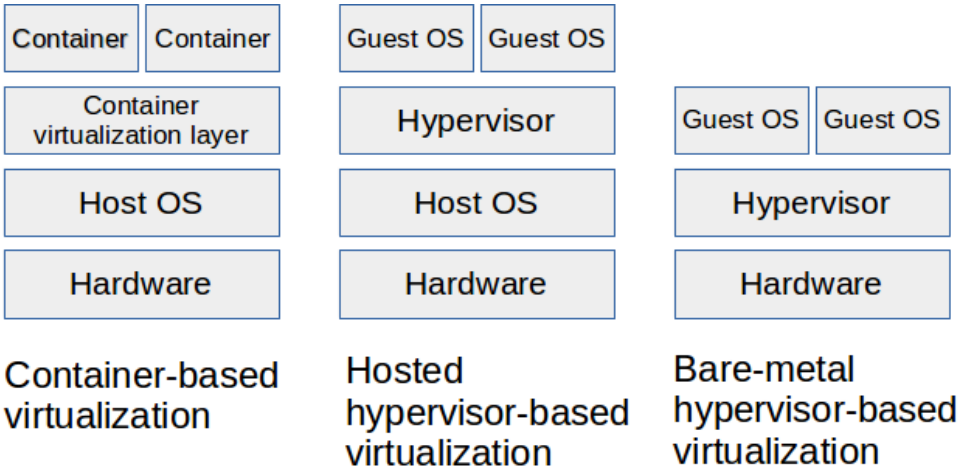


Figure 3.2: Comparison of virtualization techniques

However, the fact that the kernel of the host OS is shared between containers causes a constraint: containers must be the same type of OS as the host OS is. For example, an Ubuntu Linux can run only containers using Linux, such as Ubuntu, Gentoo, or Arch, but not Windows.

In OS-level virtualization, the virtualizing layer is quite light-weight. In LXC, for example, it basically consists of kernel features known as Cgroups and namespaces (see subsections 3.1.1 and 3.1.2). These features do not critically reduce the performance of running software since there is no need for techniques such as binary translation. In other words, the virtualization capability is a part of the OS, thus no VMM is needed.

3.2.3 Performance

According to the results of a performance research [47] comparing container-based and hypervisor-based virtualizations, the hypervisor-based one (Xen, hosted architecture) was suffering of an overhead of 4.3% in a CPU-intensive process. Moreover, the overhead in memory performance was approximately 31%, while container-based and native systems presented similar performance. As the study suggests, the overhead is caused by the virtualization layer that performs memory access translation. The disk throughput was also measured to be significantly poorer on Xen compared to LXC, of which throughput was near-native. The performance of the write and read speeds of Xen were approximately 65% and 50%, respectively. Finally, the network performance of Xen was also clearly worse. While LXC had quite small dif-

ference compared to native systems, Xen's average bandwidth was even 41% smaller than native.

Another study [37] comparing performance of Xen and LXC in processing HTTP requests and SQL SELECT queries also suggests that Xen has a significant performance overhead.

A study [15] comparing KVM (a hypervisor-based, hosted architecture virtualization) and Docker (a derivative of LXC, see chapter 3.3) performance states that "In general, Docker equals or exceeds KVM performance in every case we tested". However, according to the same study, both KVM and Docker have only negligible overhead for CPU and memory performance, and that both forms of virtualization should be used carefully for I/O-intensive workloads.

3.2.4 Operational capabilities

Due to the fact that Linux containers share the kernel of the host OS, isolated kernel updates are not possible. However, live migration between physical servers is possible in both hypervisor-based and container-based virtualization [27].

According to some studies [27, 38], many operations, such as creating or booting a Linux container and cloning a disk image takes significantly shorter time compared to similar ones of a hypervisor-based VM.

3.3 Docker

Docker is an LXC-based tool for creating application containers [4, 37]. It provides a command line toolset to wrap single applications to containers that can be run on any system having Docker installed. By utilizing LXC technology, Docker isolates the application with its environment and dependencies as if it was run in its own OS. These *dockerized* applications can then be run simultaneously on the same host OS. The goal of Docker is to make building, shipping and running of applications relatively simple.

Each Docker container is based on a Docker image. Docker image, in turn, is a stack of read-only filesystem layers. Docker relies on Advanced Multi-Layered Unification Filesystem (AuFS) to enable creation of hierarchical filesystems. AuFS is a filesystem consisting of multiple overlaid layers of filesystems. Every layer has a reference to its parent layer. Thus, Docker has to store only the differences of layers, which helps keeping the size of images relatively small. Read-only layers allows an image to be used as a basis for multiple containers.

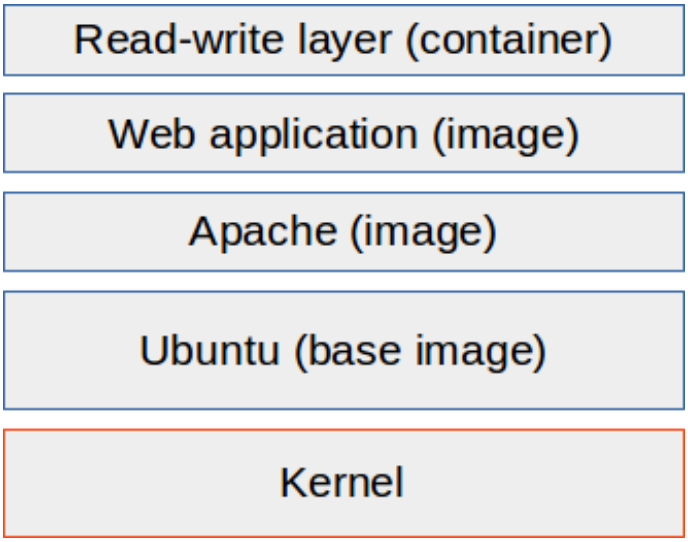


Figure 3.3: Layered filesystem of Docker container

When an image is used to create a container, an extra read-write layer is created on top of read-only layers. If a process inside the container creates a file, it is created to the read-write container. Moreover, if a file located in one of the lower layers has to be modified, it is copied to the read-write layer and changes go into the copy. As in example visualized in Figure 3.3, a plain Ubuntu can be used as a base image. Then, other images can be built on that image, resulting in a Docker image consisting of a web application and Apache installed on Ubuntu, for example.

A Docker image is typically described in a Dockerfile. *Dockerfile* is a file that contains steps how to create a specific Docker image; which base image to use, what dependencies to download, what commands to run.[4] Docker images can be created by oneself, but there is also a web site called Docker Hub[5] which provides thousands of public images for different purposes.

3.4 Conclusion

In this chapter, both hypervisor-based and operating-system-level virtualization were discussed. Although the latter one has been existed since around 2000, it has become a more plausible technology after being supported by the

[4]Another way to create a Docker image is to start a Docker container, do modifications inside it (install software, for example), and commit it with `docker commit`.

[5]https://hub.docker.com

mainline Linux kernel. It was noticed that although the virtualization technologies may seem quite similar from user's point of view, they are based on different technologies. Furthermore, despite operating-system-level virtualization being more lightweight and having better performance, it has also some limits. In addition, Docker was introduced. Docker is an example of software that enables relatively powerful things by utilizing Linux container technologies.

Chapter 4

Container-based distributed systems

In the previous chapter, a container was viewed mainly as a single unit. However, they are typically used when building complex systems, which can consist of a few or more than a thousand of containers. That is why it is important to view a container also as a small component in a bigger system.

When there is a system consisting of multiple containers communicating with each other, a distributed system is formed. This means that the system has to deal with the challenges and difficulties distinctive to distributed systems. Fortunately, there are tools and operating systems developed that aim to reduce the impediments in building a clustered, container-based system.

Although a distributed system comes with its challenges, it also comes often with some benefits. Application containers bring a new opportunity for building a modular, scalable distributed system out of small, easily manageable components. This is called micro-services architecture.

In this chapter, distributed systems and some of their typical properties are discussed. After that, micro-services architecture is discussed and compared with its opposite, monolithic architecture. Finally, container orchestration—the need for it along with some tools—are discussed.

4.1 Distributed systems

According to a definition [28], a *distributed system* "consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages". Distributed systems are typically more complicated than centralized ones. Although they provide various ben-

efits, such as better scalability, computing speed and avoidance of single point of failure, building a robust distributed system requires understanding of challenges related to it. The network in which the components of the distributed system are connected may suffer from high latencies, jitter and overload. In addition, some components may fail or crash for various reasons. This often brings difficulties with system availability and consistency of data between components. In the following subsections, some typical properties causing challenges in building distributed systems are discussed.

4.1.1 CAP theorem

In 2000, professor Eric Brewer introduced *CAP theorem*, an idea of the trade-off between three properties in a shared-data system [13]: consistency, availability and partition-tolerance. It states that at most, only two of the properties can be chosen for a system. *Consistency* means that there must exist a total order between data operations. Thus, any read operation must return the value of the latest write operation. *Availability* requires that "every request by a non-failing node in the system must result in a response" [18]. That is, the service must eventually terminate, regardless of the algorithm being used. *Partition-tolerance* means that the system continues to operate despite the network being allowed to lose arbitrarily many messages sent from a node to another.

As Brewer points out in his article written in 2012 [10, 12], CAP theorem is often misunderstood. Firstly, the "2 of 3" formulation is misleading because the choice between consistency and availability can occur multiple times within the same system. This can happen at very fine granularity - the choice can be different in quite similar operations. Moreover, because of optimizations and variety of systems, the CAP properties are more continuous than binary. Secondly, if the service cannot be reached at all, there is no choice between consistency an availability. However, there is an exception to this called offline mode [26] when part of the service runs on the client. Using the persistent on-client storage in HTML5 forms an example of this.

CAP theorem yields some insights into how a distributed system should be designed in fault-prone networks [19]. Software architects have to choose a strongly consistent system with best-effort availability, weakly consistent system with high availability or something between them.

4.1.2 Consensus problem

The *consensus problem* refers to the situation in which information has to be shared among a group of processes, preferably in a fault-tolerant manner [11].

That is, the correct (fault-free) processes should be able to consistently agree on correct results despite actions, possibly malicious ones, made by the faulty processes. The importance of the consensus problem stems from it being at the core of protocols related to synchronization, reliable communication, resource allocation and task scheduling, among others.

There are two kinds of process faults: crashes and Byzantine failures [17]. In a *crash*, the process stops all its activity. In a *Byzantine failure*, in turn, no assumptions are made about the behaviour of the process. It can do things that it is not supposed to do: send messages at wrong times, act dead for some time, or make conflicting claims about other processes.

Various protocols have been developed to provide ways to handle the consensus problem. A *t-crash resilient* protocol that can tolerate up to t crashed processes, and a *t-Byzantine resilient* protocol that can tolerate up to t processes exhibiting Byzantine failures.

One of the best-known consensus algorithm is the Paxos algorithm [29]. There are numerous variations of the algorithm with different resilience capabilities. The main principle of Paxos is that processes are given different roles, and they communicate with each other until the majority of the processes agree upon the value of piece of data.

4.2 Micro-services architecture

A traditional approach to choosing a software architecture is so-called monolithic architecture [32]. In *monolithic architecture* the software is deployed as a united component. This approach, however, has some potential problems. Firstly, as the code base grows large, productivity slows down. The quality of code will decline, and the original modularity will erode. Secondly, continuous development will get difficult - to create frequent updates to the software the whole system has to be deployed. Thirdly, although monolithic software can be copied and run simultaneously on multiple instances, it cannot be scaled in multiple dimensions. Scaling only a certain component of the software independently is quite impossible. For relatively simple systems using monolithic architecture can result in a optimal, manageable solution. However, there is also an alternative architecture called micro-services architecture.

In *micro-services architecture*, the software is divided into multiple, separate components that can be deployed independently from each other (see comparison of the architectures in Figure 4.1) [32, 44]. The services then communicate with some lightweight mechanism such as HTTP. Although micro-services architecture typically causes some overhead, it has many ad-

vantages. Firstly, small, decoupled components are easy to handle and develop. They can be written in a programming language that is best suitable for each one. Secondly, the system can be updated relatively easily because small services can be quickly services. Thirdly, scaling in multiple dimension is possible. For example, if a system consists of multiple services such as web applications, database, and worker instances, and the bottleneck is worker performance, new worker instances can be deployed. All the services do not have to be scaled by the same factor.

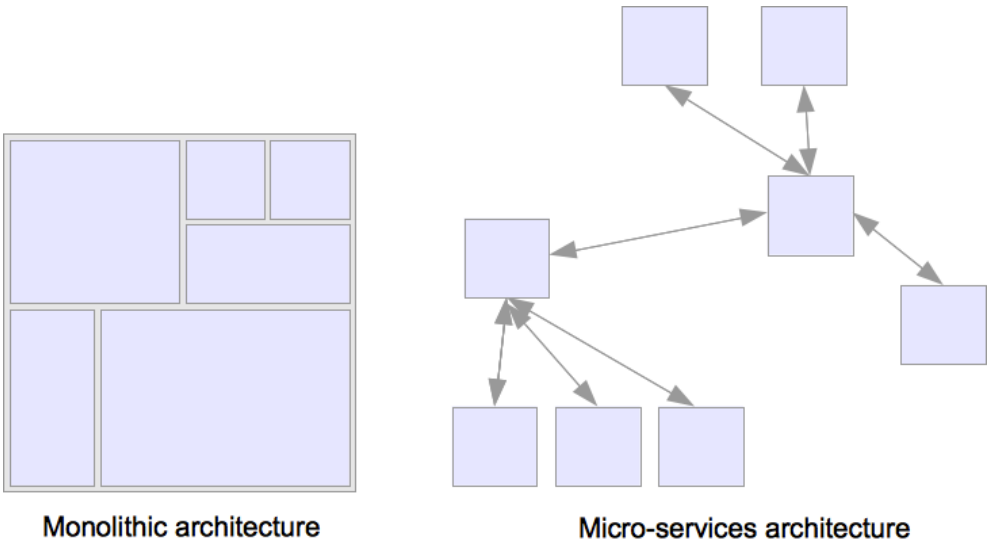


Figure 4.1: Monolithic and micro-services architectures

In general, however, micro-services architecture increases the complexity of the system [32]. First of all, the system becomes a distributed system. As mentioned, there must be a communication mechanism between services. In addition, testing becomes more difficult. Moreover, there must be a good coordination between the development teams to manage services in a way that they are compatible with each other.

Application containers bring a new opportunity for building micro-services. Services can be wrapped into containers with relatively small overhead, and, being virtually independent from each other, they can be managed as separate components. By creating container images from services, new instances can be created and shut down in seconds [15]. Cluster-level container management tools developed in recent years, such as *fleet* and *Kubernetes*, help in controlling micro-services to work together.

4.3 Container orchestration

Virtualization technology offers various useful mechanisms for isolation, resource management, live migration and checkpointing [20, 47]. However, if there is a large number of virtualized units, managing them in a controlled manner and using the provided mechanisms in the right way may become a real challenge. This raises a need for an autonomic, self-managing system that could drive the mechanisms without relying on human operators. That is, there would be a policy at cluster level to handle the behaviour of individual units and thus orchestrate the whole system. In a clustered environment, application containers are relatively small components and the ability to orchestrate them coherently is important.

At the moment, there are various operating systems designed for running application containers. In addition, several systems have been developed to orchestrate containers at cluster level. The operating systems include CoreOS[1], Ubuntu Core[2], Project Atomic[3] and RancherOS[4]. The orchestrating systems include Kubernetes[5], Shipyard[6] and Mesosphere[7]. Fleet, packaged in CoreOS, is also a type of orchestration tool, albeit a lower-level one. CoreOS and Kubernetes are introduced in the following subsections.

4.3.1 CoreOS

CoreOS is a minimal Linux distribution, designed for running services in Docker containers [3]. In addition to being minimal and being for designed for Docker, it has some features that make it exceptional from mainstream Linux distributions. First, CoreOS does not ship a package manager. Any software to be used should run in a Docker container which, in turn, has its own ways to handle dependencies. Second, CoreOS is designed to be clustered—it ships with various tools, such as fleet and etcd to easen running application containers in across multiple machines.

In a complex, highly dynamic clustered environment, it is essential to be notified when something changes: for example, new services are attached to the cluster, or locations or IP addresses of services change. To handle service discovery, CoreOS utilizes *etcd*, a distributed key-value store.

[1] https://coreos.com/
[2] http://developer.ubuntu.com/en/snappy/
[3] http://www.projectatomic.io/
[4] http://rancher.com/rancher-os/
[5] http://kubernetes.io/
[6] http://shipyard-project.com/
[7] http://mesosphere.com/

etcd uses a consensus algorithm called Raft (see subsection 4.1.2) which is meant to be more understandable than Paxos while keeping the same level of fault-tolerance and performance [22]. According to [3], etcd tolerates machine failure, including the master, and handles master elections during network partitions. Container applications can read and write data into etcd. For example, connection details of a database service can be stored into etcd and watched by other services to reconfigure themselves if something changes. To make individual machines aware of being a member of a cluster, they are connected via so-called discovery token. It is created at `http://discovery.etcd.io/new`, and given as a paramater when starting the etcd service in a machine. The token points to a URL[8] which discloses information about the cluster nodes.

Since one of the principles of CoreOS is to reduce impediments arisen in a clustered environment, it tries to aggregate individual machines into a pool of resources. *fleet* is a program that aims to eliminate the need of dealing with individual containers or machines when submitting new services [3, 7]. fleet functions as a cluster manager that decides where services should run, handles machine failures and helps in efficient resource utilization. It also simplifies the process of updating CoreOS across the cluster. fleet can be considered as a distributed init system that ties systemd[9] and etcd together.

Figure 4.2 visualizes etcd and fleet in use: etcd forms a communication channel between containers and fleet manages the allocation of resources for application containers at the cluster level.

4.3.2 Kubernetes

Kubernetes is a container orchestration system released by Google [30]. It aims to provide an environment to schedule, deploy and manage groups of containers in a clustered environment.

Network is an important aspect since service discovery, communication and synchronization are done through it. To architect the network of their systems, Kubernetes provides some abstractions, of whom two will be introduced.

A *pod* is a group of coupled containers on the same machine, sharing the same resources. The containers are placed inside the same network namespace, and thus they can communicate with each other via localhost. Moreover, pods can be given an own routable IP address. However, since pods

[8]Example token: https://discovery.etcd.io/7ab76cebba7d0bc39fd4cdd46a6d0b37

[9]*systemd* is an init system used in a wide range of Linux distributions. Init (short for initialization) system is the first process a Unix computer starts, being the ancestor of all other processes.

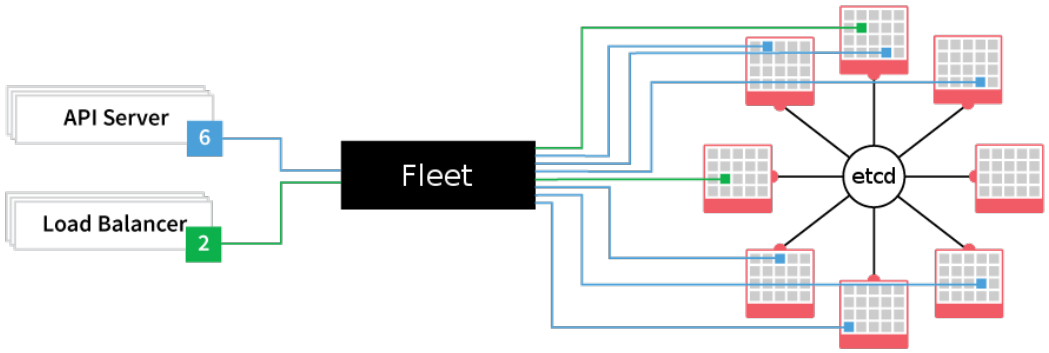


Figure 4.2: etcd and fleet in a CoreOS-based application container cluster [3].

are considered ephemeral, it is not recommended to address a pod by its IP address. Machine maintenance, for example, can replace the instances with new ones.

A *service*, in turn, is a group of pods with a stable addressing. All requests to the given IP address are load-balanced to active pods in the service. Pods can come and go, and the service corrects the routes. Services handle routing using both iptables and a service proxy. The service proxy keeps a list of all its pods capable of responding to requests targeted to the service. Most services only talk to other services, and typically they are not exposed to the outside world.

Service discovery in a Kubernetes cluster can be done in two ways. First, by exposing environment variables (regarding IP addresses and such) in the pods. Second, by having an internal DNS service in the cluster and letting each service to have a DNS address.

4.4 Conclusion

In this chapter, many things related to distributed systems, micro-services and container orchestration were discussed. As noticed, distributed systems have their advantages but also disadvantages. They might improve things such as isolation, modularity and scalability, but at the same time they challenge developers with consensus problem, latencies and complexity. Moreover, CAP theorem forces developers to make trade-offs between consistency availability.

Furthermore, the micro-services architecture has received attention. It is

justified to state that lightweight application containers bring new opportunities for building micro-services.

Finally, some technologies facilitating container orchestration—managing numerous containers in a controlled manner—was discussed.

Chapter 5

Project implementation

This chapter discusses the implementation of the transcoding service prototype, described in the previous chapter. After going through the target and environment of the project, the architecture and the main responsibilities of its components are introduced. The components themselves, their internal design and patterns, are not discussed in depth because they are not so relevant from the perspective of this paper. A significant portion of implementation time was spent on programming those services though. However, what is more relevant is how the individual components are transformed into Docker images, how they are run inside containers, communicating with each other in a cluster.

Second, it is shown how a Dockerfile is written for one of the components. A practical example about creating a Docker image is given.

Third, it is discussed how containers are started from the Docker images and run in a cluster. The process of setting up a cluster capable of running Docker containers is gone through, as well as creating services eligible for fleet and handling persistent data across containers. Finally, the discovery between each components of the transcoding service—and interaction them—is discussed.

5.1 Target and environment

The target of this project is to develop a video transcoding service prototype that is capable of receiving video files and output different quality versions of the original video. It must use Docker container technologies and comply with the micro-services architecture.

5.1.1 Motivation

Video transcoding is computationally heavy. When building a transcoding service, the load for the system may vary quite a lot. This, in turn, may affect the overall quality of the service: transcoding may become very slow, or the whole service ceases to respond, for example. One option to try to avoid these problems is to make the service comply with the micro-service architecture. By separating functionalities to micro-services and making them to not share the same computational resources, the load on a micro-service does not affect another. In addition, the service can be scaled in many dimensions. If video transcoding capability needs to be increased, new transcoding micro-services can be added without affecting the service providing a user interface, for example. By having lightweight, self-contained micro-services, the available hardware capability can be shared more efficiently. Furthermore, micro-services are ofter easier to move from place to place since they are not so tightly coupled with their host machines. Because scaling out is typically cheaper than scaling up, this is an important aspect.

However, if there are multiple micro-services on the same machine running without proper isolation, the host environment and other micro-services may affect too much their functioning and performance. On the other hand, having an own virtual machine for every micro-service may be too heavy.

To try to solve these problems, the transcoding service will be built with container technologies, using a microservices-architecture. After the implementation, the result itself but also the development phase will be evaluated, and interesting findings shared. The goal is to gives some valuable insights about the strengths and weaknesses of using container technologies when developing a small distributed system.

5.1.2 Environment and requirements

The transcoding service has two types of requirements: functional requirements that are visible to user, and internal requirements regarding architectural and technological choices.

The functional requirements for the service are that it must provide a website with an user interface allowing end-users to upload MP4 videos. These videos must be transcoded into multiple videos with different qualities, and the end-user must be able to watch them after the transcoding process.

The internal requirements are that the service must comply with the micro-services architecture, containing at least a few unique services. Services must be distributed among multiple nodes. Moreover, at least one of the services must have multiple instances running at the same time (video

transcoding processes).

In addition, the service instances must be run as Docker containers. The Docker containers must be run on a cluster of virtual machines, in a cloud.

5.2 Architecture and components

The video transcoding service follows micro-services architecture. It consists of four types of components: Web front-end server, Database, Storage server and Transcoder. The system has only one instance representing each component, except from Transcoder with one to several instances.

The main flow of the system is as follows (Figure 5.1 also demonstrates the flow):

1. User requests an HTML page from Web front-end server containing a video file uploading form.
2. User uploads a video.
3. Web front-end server sends the video to Storage server and inspects the local copy with mediainfo, generates transcoding jobs and inserts them to the Database.
4. Transcoder instances poll Web front-end server for unprocessed jobs. Web front-end server queries Database, and returns if there are.
5. After getting a job, a Transcoder instance fetches the original video file from Storage server, transcodes it to the quality given in the job object, and sends the output file to Storage server.
6. The Transcoder instance notifies Web front-end server for a processed job. Web front-end server updates the job status into the database.
7. The HTML page lists all jobs, their statuses and links to videos with different quality versions.
8. User watches video files with desired quality.

5.2.1 Web front-end server

Web front-end server functions as the core of the system. Its responsibilities include

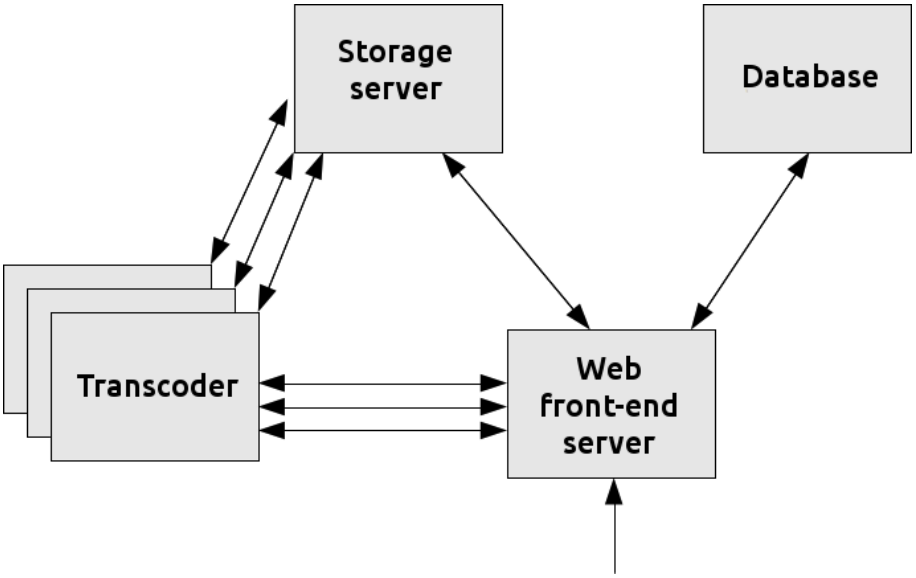


Figure 5.1: System architecture

- providing an user interface,
- communicating with the database,
- inspecting uploaded video files,
- providing an interface for Transcoder instances to process jobs and
- intermediating between user and Storage server.

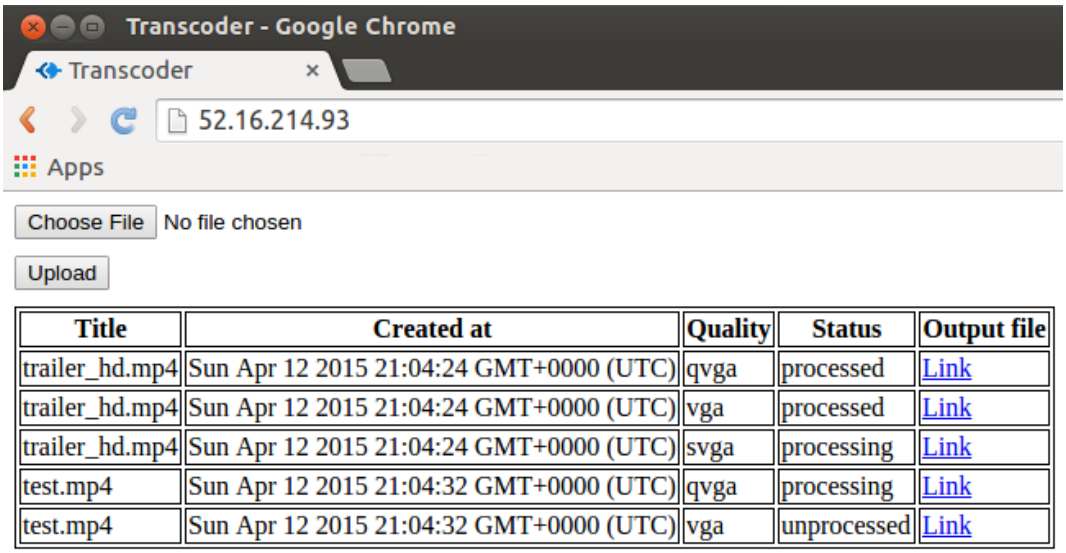
The user interface is a stripped-down one containing only a video uploading form and a list of jobs. No additional styling is added. Figure 5.2 shows the appearance of the user interface.

The video file inspection is started by executing mediainfo with the video file as its parameter. The width, height and aspect ratio of the video are obtained from the mediainfo output. Depending on the characteristics of the video, several lower quality videos will be transcoced. For each of them, a job describing the quality is generated.

The output videos are capped to the following resolutions[1]: QVGA (320x240), VGA (640x480) and SVGA (800x600).

The Transcoder instances are given the following HTTP interface:

[1]http://en.wikipedia.org/wiki/List_of_common_resolutions



Title	Created at	Quality	Status	Output file
trailer_hd.mp4	Sun Apr 12 2015 21:04:24 GMT+0000 (UTC)	qvga	processed	Link
trailer_hd.mp4	Sun Apr 12 2015 21:04:24 GMT+0000 (UTC)	vga	processed	Link
trailer_hd.mp4	Sun Apr 12 2015 21:04:24 GMT+0000 (UTC)	svga	processing	Link
test.mp4	Sun Apr 12 2015 21:04:32 GMT+0000 (UTC)	qvga	processing	Link
test.mp4	Sun Apr 12 2015 21:04:32 GMT+0000 (UTC)	vga	unprocessed	Link

Figure 5.2: Appearance of the Web front-end user interface

Method	Path	Description
GET	/choose-job	Returns an *unprocessed* job and marks it as *processing*
POST	/job-processed/<job-id>	Marks the job with id <job-id> *processed*

To prevent multiple Transcoder instances from getting the same job when requesting for a job at the same time, the query that returns an unprocessed job and the query marks it as processing are put inside a transaction.

When a user uploads a video file, Web front-end server sends it to Storage server. As the transcoded video files are also kept in Storage server, user has to be given a way to download them to watch them. Storage server, however, is not exposed directly to the user. Thus, Web front-end server provides path `/files/*` for requesting files from Storage server. All GET requests to the path are proxied to the Storage server.

Web front-end server is written in Node.js[2] enhanced with Express[3] web application framework at server side. To add dynamic content (list of jobs) to the HTML page at the client side, Embedded JavaScript[4] (EJS) is used. Video files are inspected with Mediainfo (see subsection 2.3.1).

[2]https://nodejs.org
[3]http://expressjs.com
[4]http://www.embeddedjs.com/

5.2.2 Database

Database component is responsible for storing information about the transcoding jobs. The transcoding service uses a relational database management system called MySQL, which contains a database called *transcoder*, with one table in it: *job*. A job instance has the following properties:

- id: unique identifier
- title: original name of the video file
- file id: public identifier connecting all jobs related to specific file upload
- created: creation time stamp of the job
- aspect ratio: a decimal representing the aspect ratio of the video
- quality: desired transcoding quality of this job (QVGA, VGA, SVGA)
- status: processing status (unprocessed, processing, or processed)

5.2.3 Storage server

Storage server component functions as a static web server that receives and serves video files. It listens for HTTP POST requests to receive files and GET requests to serve files. The files are stored at */files/* folder (more about handling persistent data with containers in subsection 5.4.3).

As Web front-end server, Storage server is written in Node.js and Express framework.

5.2.4 Transcoder

Transcoder instances poll Web front-end server for unprocessed jobs by sending a HTTP GET request with intervals of 500 ms. If they get a job as a response, an ffmpeg command to transcode is formed and executed. The ffmpeg command is given different parameters depending on the given quality requirement. For example, to output a QVGA quality video, a command similar to the following one would be executed:

```
ffmpeg -i http://<storageserver hostname>/oR9aS21d.mp4 -vsync 1
    -vcodec libx264 -profile:v baseline -crf 20 -preset veryfast
    -r 25 -s 320x240 ./transcoded/oR9aS21d-qvga.mp4
```

The following table explains the parameters in the command.

Parameter	Explanation
-i <url>	Give input file as a remote stream from Storage server
-vsync 1	Duplicate and drop frames to achieve exactly the requested constant frame rate.
-vcodec libx264	Use libx264 to encode as H.264
-profile:v baseline	H.264 profile ("main" for VGA quality, "high" for SVGA quality)
-crf 20	Constant rate factor to target certain video quality
-preset veryfast	Very fast encoding speed but less compression
-r 25	Frame rate of the output video
-s 320x240	Resolution of the output video

As Web front-end server and Storage server, Transcoder is written in Node.js.

5.3 Creating Docker images

To run the components of the transcoding service as Docker containers, there must be Docker images which the containers can be created from. As discussed in subsection 3.3, a typical way to create Docker images is to write a Dockerfile.

Each of the four components has a Dockerfile. The following listing is from the Dockerfile of Web front-end server.

```
FROM node:0.10

WORKDIR /src
COPY . .

RUN apt-get update
RUN apt-get install -y mysql-client mediainfo
RUN npm install

EXPOSE 3000
CMD ["node", "app.js"]
```

The first line tells Docker to use an image named node:0.10 as a base image (which will be automatically downloaded from Docker Hub, if not already). `WORKDIR` points the working directory inside image during the building phase. The line starting with `COPY` tells Docker to copy all files from the directory where Dockerfile is located to the working directory inside

the image. After that, some commands are run to install dependencies. In fact, every `RUN` instruction creates a new layer on top of the current image. The `EXPOSE` instruction tells containers created from the image to expose port 3000 to the host machine. The last instruction tells the command to be run after a container has started.

To build Dockerfiles and push them to the Docker hub, a Bash script was created. It calls two commands for each component (in this case, Web front-end server):

1. `docker build -t tiksa/thesis-webapp thesis-webapp/` to build a Docker image and tag it with name tiksa/thesis-webapp
2. `docker push tiksa/thesis-webapp` to send the image to Docker hub

5.4 Running containers in a cluster

5.4.1 Setting up a cluster

The cluster used for the transcoding service, being a small one, consists of three micro EC2 machines running in Amazon AWS cloud. They are named *coreos-1*, *coreos-2* and *coreos-3*. The first one is intended to run Web front-end server, Database and Storage server. coreos-2 and coreos-3 will run only Transcoder containers. Figure 5.3 visualizes the cluster setup.

To make individual CoreOS machines to be aware of being in a cluster and communicate with each other via etcd, they have to be connected via an etcd discovery token. This is accomplished during start-up of the CoreOS instances. They are given a snippet of information in the AWS Management Console about services to be started and their parameters. The following snippet was used for all the instances:

```
#cloud-config

coreos:
    etcd:
        discovery: https://discovery.etcd.io/504
            d625362b0477a06da6c12444e84e0
        addr: $private_ipv4:4001
        peer-addr: $private_ipv4:7001
    units:
        - name: etcd.service
          command: start
        - name: fleet.service
          command: start
```

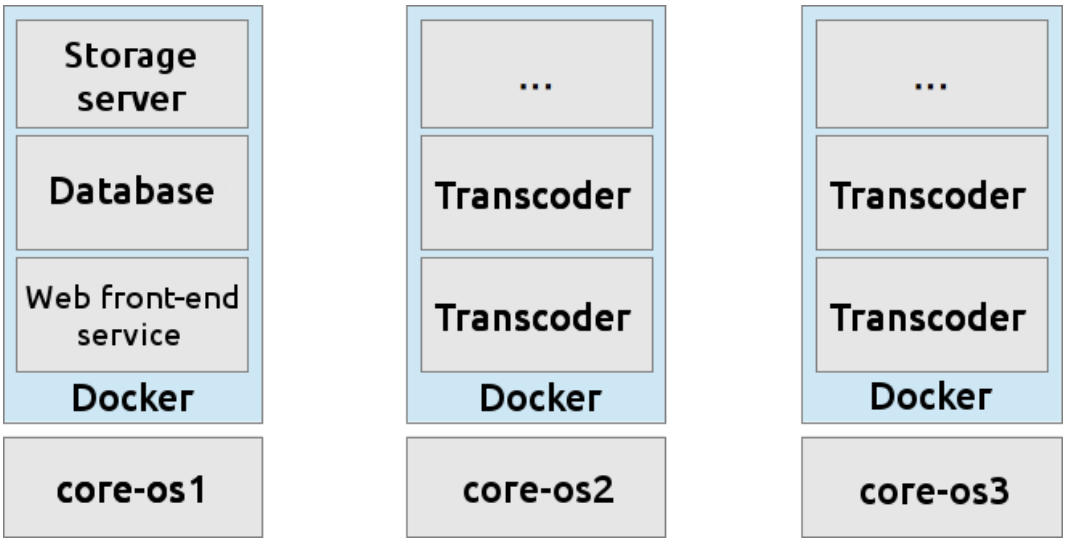


Figure 5.3: Containers distributed in a CoreOS cluster.

Basically, it tells the instance to start services etcd and fleet, and to use a specific discovery token for etcd. It also tells etcd to advertise certain IP addresses and ports for client-to-server and server-to-server communication.

5.4.2 Wrapping container creation to services

Now that there is a CoreOS cluster running and Docker images are built, the next step is to start the actual containers. It is possible to connect to each CoreOS instance via SSH, pull the needed Docker images and start containers manually. However, this approach is quite error-prone and laborious. Instead, it is recommended to use fleet to launch services at cluster-level, not on separate machines. To make the components eligible for being run as a fleet service, a descriptive service file has to be written. The following snippet describes Web front-end server as a service:

```
[Unit]
Description=Thesis webapp
Requires=docker.service

[Service]
EnvironmentFile=/etc/environment
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill webapp
ExecStartPre=-/usr/bin/docker rm webapp
ExecStartPre=/usr/bin/docker pull tiksa/thesis-webapp
```

```
ExecStart=/usr/bin/docker run --rm=true --name=webapp -p 80:3000
    -e COREOS_PRIVATE_IPV4=${COREOS_PRIVATE_IPV4} -e COREOS_PORT
   =80 --link mysql:mysql --link storageserver:storageserver
   tiksa/thesis-webapp
ExecStop=/usr/bin/docker stop webapp

[X-Fleet]
MachineID=63f57bcb9c014d94aebdb4f6e8d55c20
```

What is interesting in this description are the `docker` commands and the X-Fleet section. By executing `docker kill` and `docker rm`, it is ensured that the previous container is stopped and removed. The `docker pull` command fetches the newest Docker image, if not already fetched. Only the layers up from the modified one have to be fetched. The `docker run` command starts the an actual container. It has various parameters that are explained in the following table.

Parameter	Explanation
–rm=true	Remove the container when it stops.
–name=webapp	Set name of the container.
-p 80:3000	Forward port 80 of the host to port 3000 of the container.
-e KEY=VALUE	Pass environment variable to the container. CoreOS private IP address and port are needed to advertise the container between cluster nodes via etcd.
–link mysql:mysql	Link other container (mysql) to the starting container with alias mysql. This adds environment variables that contain information about the linked container, such as IP address and exposed port.
–link storage-server:storageserver	See the previous one.

In the X-Fleet section, different constraints regarding the decision about in which node the container is going to be started, can be listed. Because all components except Transcoder are desired to be run on the same machine (coreos-1), they are given its machine ID. This forces fleet to always start those containers on that machine. On the contrary, Transcoder containers are instructed to run on any container except the one that is running Web front-end server container. In the service file, it is formulated as a statement `Conflicts=thesis-webapp.service`. In practice, this results in a

policy where the Transcoder containers are distributed evenly on machines coreos-2 and coreos-3.

5.4.3 Handling persistent data

Normally, when containers are removed, all their data is deleted. However, both Database and Storage server are intended to persist their data storages despite deletion of the containers running the components. This is achieved by using volumes. In `docker run` command, a `-v` parameter is given which makes the container mount a specific directory in the file system of the host machine into a directory in the container.

In case of Database, the following parameter will be set:

```
-v /home/core/thesis-mysql:/var/lib/mysql
```

Thus, all the data will be stored at `/home/core/thesis-mysql`. If the container is stopped and removed, and a new container with an updated Database image is started, it will continue using the same data.

Similarly, Storage server containers are started with parameter:

```
-v /var/thesis-storageserver/files/:/files/
```

5.4.4 Service discovery

Now that containers can be started in the cluster, they have to make communicate with each other. The transcoding service has the following relations between the containers of its components:

Service	Services to be discovered
Database	None
Storage server	None
Web front-end server	Database (same host), Storage server (same host)
Transcoder	Web front-end server (other host), Storage server (other host)

As a relatively simple solution, containers that run on the same host can use `--link` parameter in the `docker run` command to obtain address information about each other. For example, Web front-end server uses parameter `--link mysql:mysql` to link the Database container, and in the application code, an environment variable called `MYSQL_PORT_3306_TCP_ADDR` is used to form a database connection.

For containers that run on different hosts, etcd is used to relay address information. Moreover, a Node.js module called *node-etcd* is used to interact easily with the etcd daemon. Both Web front-end server and Storage server advertise their IP address and port every five seconds, with a time-to-live (TTL) value of ten seconds, whereas Transcoder containers, in turn, watch for those values and use them to communicate with those components.

Figure 5.4 gives an overview of the relevant ports that are exposed internally and externally, and forwardings related to them.

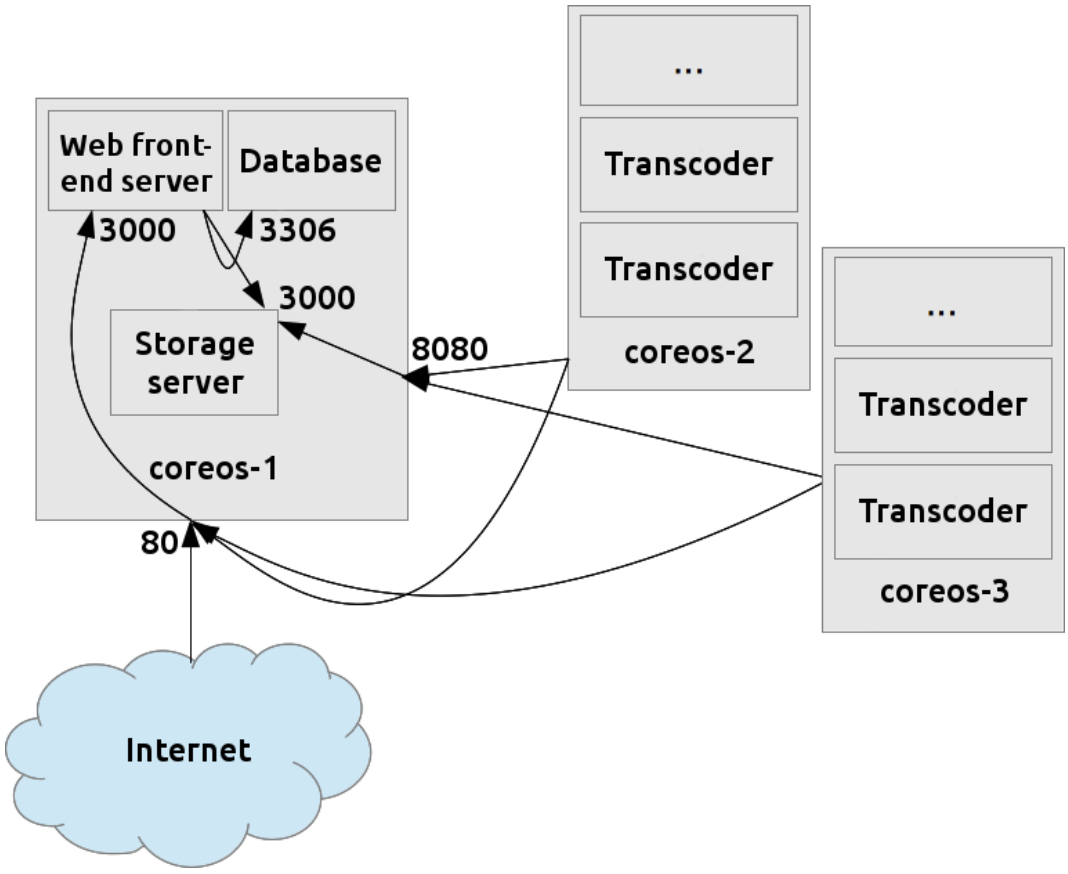


Figure 5.4: Exposed ports and port forwardings

5.4.5 Deployment

To automate the process of making the cluster to replace one or all of its currently running services (related to the transcoding service) with updated ones, a Bash script was made for each component. The script builds a new

Docker image with the latest application code in it and relaunches the service by pulling that image from the Docker Hub. In addition, a Bash script that runs all these scripts in a row was made to make it possible to launch all the services with one command.

Chapter 6

Evaluation

In this chapter, the implemented prototype will be evaluated. The focus will be more on the development and deployment process, scalability, maintainability, and other operational characteristics—the functionality of the actual service visible to the end-user will be evaluated quite superficially.

6.1 Functionality

The transcoding service implemented in this project meets the requirements set for it. The service provides an online user interface that allows end-users to upload MP4 videos (also other formats are allowed, there is no guarantee of success though). The videos are transcoded to multiple output videos with different qualities, depending on the quality of the input video. The service also provides links for watching the output videos.

6.2 Architecture

The architectural requirements for the service were met. It consists of four types of micro-services which are distributed among three nodes. In addition, one of the micro-services, Transcoder service, has multiple instances running. All the micro-service instances are run as Docker containers on a cluster of virtual machines, in a cloud.

6.3 Performance

The video processing performance depends largely on the underlying hardware capacity. However, some measurements were done. A 52 seconds long

video with size of 13.9 megabytes and overall bit rate of 2.24 Mbps was uploaded four times to the service. Each of them generated three transcoding jobs (QVGA, VGA and SVGA qualities), thus 12 jobs in total. Then, a varying number of Transcoder instances were started at once (divided evenly to two CoreOS hosts), and the total time elapsed for the transcoding was measured. With each configuration, the total processing time was measured three times. The following table shows average processing times.

Number of Transcoder instances	Average processing time (s)
2x1	144
2x3	154
2x6	150

Because the Transcoder instances poll for available jobs with intervals of 500 ms, they might have been idling a few seconds in total during the test despite available jobs. Anyway, although the the number of samples was quite small, seems that one Transcoder instance for each CoreOS node has the best total performance.

However, the optimal number of Transcoder instances for a CoreOS node does not depend only on things such as number of computing cores. In addition, there is a clear trade-off: with a high number of Transcoder instances, a high number of jobs are being processed in parallel but the processing time for each video is much longer, and vice versa. This could be also considered as a matter of user experience: is it preferred that end-users get their videos quickly to start being processed, which takes longer, or is it acceptable to let them wait for some time and then get the video processed fast?

6.4 Scalability

The scalability of the transcoding services is evaluated mainly in an operational sense.

Because the micro-services in the transcoding service were wrapped into containers, they were quite easy to start on different host. This, in turn, makes it easier to scale out. For example, after having a Transcoding instance functioning on a specific host, it was quite a small step to make it distributable to any available host. However, sometimes there were some quirks in fleet. For example, sometimes when issuing a *destroy* command for all Transcoding instances, some of them did not seem to obey. In addition, it was important to have the same CoreOS versions on each node to ensure that etcd, for example, functions as expected.

Furthermore, the hardware capabilities of the available hosts were able to be shared more efficiently—small, self-contained, decoupled containers can be distributed with less effort than large, tightly coupled services. For example, coreos-1 host was running three separate services. Despite this, they were isolated from each other because they were run as containers. Although their dependencies were changed many times, only the modified service had to be restarted. There was no need to change the host environment in any way, for example by installing dependencies there.

6.5 Development flow

In the beginning, each service, excluding Database, was developed locally as an independent application to some extent. As they became more and more dependent of each other, they were made to communicate each other locally. After having somewhat working components implemented, the CoreOS cluster was set up. After this, the services were created Docker images. The deployment flow, as explained in chapter 5, was, however, too slow for iterative, yet active development phase. The time from changing code to see it live took even minutes.

At this point, the flow for development was changed. The goal was to have an effective flow but also a somewhat realistic development environment. To address this problem, each service was created a separate, development-purpose fleet service file. The main difference was that the created containers would mount a directory from the CoreOS host and use it as the source directory. Furthermore, after changing code, the source files would be sent to the remote source directory on the CoreOS host via SCP. This enabled quite efficient development flow. In most cases, the iteration time were several seconds.

However, when new Node.js modules were to be downloaded, they had to be installed manually on CoreOS, or sent via SCP from the development machine, which might take even minutes. To address this problem, and to see code changes live even faster, it would be probably worth setting up a local CoreOS cluster. Vagrant[1], for example, would perhaps have been quite a suitable tool for this. This would enable having a common source directory for containers and local development tools, or at least the file transfer time would be dramatically shorter.

[1] https://www.vagrantup.com/

6.6 Testability

Because the transcoding service is run with Docker container which, in turn, are built from Docker images, it is relatively easy to setup a similar cluster and test the service there. There is no need to spend time to make the operating system, numerous dependencies and software versions identical—Docker specifically aims to take care about that.

In this respect, testability is good. In general, however, distributed systems are sometimes quite difficult to test reliably. Because no thorough testing was not carried out, it is challenging to evaluate testability in its entirety.

6.7 Reliability

According to ISO 9126 standard, reliability means maturity, fault tolerance, and recoveribility. Since the transcoding system has not been thoroughly tested, it has had only few users and it is prototypal by nature, it cannot be considered mature. Fault tolerance and recoverability is more or less mediocre; if a Transcoder instance fails to complete a transcoding job, the instance itself does not crash but continues by taking the next available job. The other parts of the transcoding service also have somewhat same level of fault tolerance. If a current task fails, then it fails, but the container instance itself is able to continue.

Chapter 7

Conclusion

As Internet and its derivatives, such as social media, have expanded dramatically, the requirements for web services have grown significantly. Many of them have to serve massive numbers of users simultaneously. This often complicates the software and infrastructure. To manage a complex distributed system and numerous components, containers may be a solution worth considering.

In this work, different topics from video processing to container orchestration were discussed. Different video streaming technologies, key concepts of video transcoding and some related open source software were gone through. Then, operating-system-level virtualization and hypervisor-based virtualization were introduced and compared. In addition, Docker, one of the popular applications of container technologies, was introduced. Because containers often communicate with other containers, they form a distributed system. Some charasteristics of distributed systems and micro-service architecture were discussed, as well as container orchestration.

In addition, a prototypal video transcoding service was implemented. Although the components of the service might have been interesting to be discussed as independent components, the focus was in the system as a whole; how the components communicate with each other and work together so that a coherent system is formed. Although the user interface visible to the end-users is stripped-down, the interior of the system is relatively complex.

In respect of project requirements, the implemented transcoding service was successful. However, there are plenty of things that could be improved. First, to prevent Transcoder instances from idling because of polling, they could be set to listen for available jobs at some port and Web front-end server could send them job descriptions. There could be a load balancer container that divides jobs for idling Transcoder instances. Second, the service could always be built more reliable in terms of service discovery. If a container fails

fatally or disappears for other reasons, the other containers do not handle it always correctly. Database and Storage server, for example, are linked to Web front-end server with Docker's `--link` parameter, which is not very flexible compared to a proper etcd-based discovery. Finally, failure recoverability could be improved. Currently, if a transcoding job fails, it stays in status *processing*, and it is not retried or given to another Transcoder instances.

Although numerous topics related to video processing, application and Linux containers, and distributed systems were discussed, this paper just scratched the surface. As future work, it would be interesting to study more the technologies developed for container orchestration. The container technology itself is relatively young, and so are many container orchestration technologies. A small project consisting of several containers is somewhat straightforward to be made manageable. However, how about a system that contains hundreds of containers that come and go? Are the existing orchestrating tools reliable enough to make the system robust?

It will be exciting to see how the field of container technology develops in the future. Many of the related technologies are not very mature, and there are not any proper alternative for Docker at the moment. Fortunately, it seems that there are some challengers being developed. Moreover, the future will show how cloud providers will react to containers and operating-system-level virtualization in general. Amazon AWS, for instance, has taken a step by releasing a service for running containers in cloud.

Bibliography

[1] Cisco Visual Networking Index: Forecast and Methodology, 2013-2018, 2014. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf. Accessed 21. Feb 2015.

[2] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014-2019, 2015. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.pdf. Accessed 21. Feb 2015.

[3] CoreOS Documentation, 2015. https://coreos.com/docs/. Accessed 4. Mar 2015.

[4] Docker documentation, 2015. https://docs.docker.com. Accessed 2. Feb 2015.

[5] Documentation page for MPlayer, 2015. https://www.mplayerhq.hu/DOCS/HTML/en/index.html. Accessed 16. Apr 2015.

[6] FFmpeg, 2015. Wikipedia page about FFmpeg. http://en.wikipedia.org/wiki/FFmpeg. Accessed 28. Feb 2015.

[7] Fleet - a distributed init system. Github page., 2015. https://github.com/coreos/fleet. Accessed 4. Mar 2015.

[8] Manual page of MPlayer, 2015. http://www.mplayerhq.hu/DOCS/man/en/mplayer.1.html. Accessed 16. Apr 2015.

[9] Mediainfo manual page, 2015. http://manpages.ubuntu.com/manpages/precise/man1/mediainfo.1.html. Accessed 28. Feb 2015.

[10] Abadi, D. J. Consistency tradeoffs in modern distributed database system design. *Computer-IEEE Computer Magazine 45*, 2 (2012), 37.

[11] Barborak, M., Dahbura, A., and Malek, M. The consensus problem in fault-tolerant computing. *aCM Computing Surveys (CSur) 25*, 2 (1993), 171–220.

[12] Brewer, E. Pushing the cap: Strategies for consistency and availability. *Computer 45*, 2 (2012), 23–29.

[13] Brewer, E. A. Towards robust distributed systems. In *PODC* (2000), vol. 7.

[14] Deka, G. C., and Das, P. K. An overview on the virtualization technology. *Handbook of Research on Cloud Infrastructures for Big Data Analytics* (2014), 289.

[15] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. *technology 28* (2014), 32.

[16] FFmpeg project. FFmpeg documentation, 2015. `https://www.ffmpeg.org/documentation.html`. Accessed 21. Jan 2015.

[17] Fischer, M. J. The consensus problem in unreliable distributed systems (a brief survey). In *Foundations of Computation Theory* (1983), Springer, pp. 127–140.

[18] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2 (2002), 51–59.

[19] Gilbert, S., and Lynch, N. A. Perspectives on the cap theorem. Institute of Electrical and Electronics Engineers.

[20] Grit, L., Irwin, D., Yumerefendi, A., and Chase, J. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing* (2006), IEEE Computer Society, p. 7.

[21] Hock, K. S., and Lingxia, L. Automated processing of massive audio/video content using ffmpeg. *Code4Lib Journal*, 23 (2014).

[22] Howard, H. Arc: analysis of raft consensus. *Technical Report UCAM-CL-TR-857* (2014).

[23] Jonathan Corbet. Cgroups, 2007. `http://lwn.net/Articles/236038/`. Accessed 22. Jan 2015.

[24] Jonathan Corbet. Fixing control groups, 2012. `http://lwn.net/Articles/484251/`. Accessed 22. Jan 2015.

[25] Kim, D., Baek, J., and Fisher, P. S. Adaptive video streaming over http. In *Proceedings of the 2014 ACM Southeast Regional Conference* (2014), ACM, p. 26.

[26] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 3–25.

[27] Kumar, A. S. *Virtualizing Intelligent River R: A Comparative Study of Alternative Virtualization Technologies*. PhD thesis, Clemson University, 2013.

[28] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[29] Lamport, L. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[30] Marmol, V., Jnagal, R., and Hockin, T. Networking in containers and container clusters.

[31] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview, 2013. `http://lwn.net/Articles/531114/`. Accessed 23. Jan 2015.

[32] Namiot, D., and Sneps-Sneppe, M. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014), 24–27.

[33] Paul Menage, Paul Jackson, Christoph Lameter. Cgroups, 2014. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`. Accessed 22. Jan 2015.

[34] Pires, K., and Simon, G. Dash in twitch: Adaptive bitrate streaming in live game streaming platforms. In *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming* (2014), ACM, pp. 13–18.

[35] Qiu, F., and Cui, Y. An analysis of user behavior in online video streaming. In *Proceedings of the international workshop on Very-large-scale multimedia corpus, mining and retrieval* (2010), ACM, pp. 49–54.

[36] Richardson, I. E. *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

[37] Scheepers, M. J. Virtualization and containerization of application infrastructure: A comparison.

[38] Seo, K.-T., Hwang, H.-S., Moon, I.-Y., Kwon, O.-Y., and Kim, B.-J. Performance comparison analysis of linux container and virtual machine for building cloud.

[39] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 275–287.

[40] Stockhammer, T. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems* (2011), ACM, pp. 133–144.

[41] Thang, T. C., Le, H. T., Pham, A. T., and Ro, Y. M. An evaluation of bitrate adaptation methods for http live streaming. *Selected Areas in Communications, IEEE Journal on 32*, 4 (2014), 693–705.

[42] Tian, G., and Liu, Y. Towards agile and smooth video adaptation in dynamic http streaming. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 109–120.

[43] Timmerer, C., Griwodz, C., Begen, A. C., Stockhammer, T., and Girod, B. Guest editorial adaptive media streaming. *IEEE Journal on Selected Areas in Communications 32*, 4 (2014), 681–683.

[44] Van Garderen, P. Archivematica: Using micro-services and open-source software to deliver a comprehensive digital curation solution. In *Proceedings of the 7th International Conference on Preservation of Digital Objects, Vienna, Austria* (2010), Citeseer, pp. 145–149.

[45] Vetro, A., Christopoulos, C., and Sun, H. Video transcoding architectures and techniques: an overview. *Signal Processing Magazine, IEEE 20*, 2 (2003), 18–29.

[46] Vetro, A., Xin, J., and Sun, H. Error resilience video transcoding for wireless communications. *Wireless Communications, IEEE 12*, 4 (2005), 14–21.

[47] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* (2013), IEEE, pp. 233–240.

[48] Xin, J., Lin, C.-W., and Sun, M.-T. Digital video transcoding. *Proceedings of the IEEE 93*, 1 (2005), 84–97.

[49] Yang, H., Chen, X., Yang, Z., Zhu, X., and Chen, Y. Opportunities and challenges of http adaptive streaming. *International Journal of Future Generation Communication & Networking 7*, 6 (2014).

[50] Yetgin, Z., and Seckin, G. Progressive download for multimedia broadcast multicast service. *IEEE MultiMedia*, 2 (2009), 76–85.