Cloud execution environment for real-time media applications

EDDIE KÄMPE
Cloud execution environment for real-time media applications

Author: Eddie Kämpe
Examiner: Johan Montelius, KTH
Industrial Supervisor: Stefan Hellkvist, Ericsson AB

Master’s Thesis
School of Information and Communication Technology
KTH Royal Institute of Technology

June 22, 2015
Abstract

Smartphones and other Internet of Things devices have become a rapidly growing topic. Along with the growth comes new technologies, like Web Real-Time Communication (WebRTC), that enables richer services to be built for the devices. These kind of services are typically consumed on-demand, in shorter periods at a time. Likewise have cloud computing exploded in popularity during the last years. Cloud computing offers compelling advantages, such as rapid elasticity and on-demand usage, that allow servers’ resource utilization to be more efficient. The flexibility of allocating and releasing resources swiftly as they are required, enables services that run in the cloud to adopt to ephemeral workloads.

The research in this thesis targets a real-time video streaming service that is based on WebRTC. Incoming streams are handled by Multipoint Control Units (MCUs) that have the responsibility to redistribute the incoming streams to the consumers. Scaling horizontally aligns well with the idea of cloud computing. The work in this thesis is based on the extreme case where each of the incoming streams are handled by a separate MCU.

The thesis presents the process of finding a flexible Cloud Execution Environment (CEE) for the streaming service. The process includes an analysis of the streaming service’s requirements, an evaluation of existing solutions, and an implementation. Moreover, the thesis includes a discussion about the capabilities of the implemented system. The result of the thesis is a CEE upon which the streaming service can be deployed and managed. The developed CEE allows any workload that is encapsulated within a Docker container to be orchestrated, not exclusively the streaming service, which makes the implementation viable to other cloud computing projects.

Keywords: Cloud Execution Environment, Web Real-Time Communication, Docker, Horizontal scaling.
Sammanfattning

Användandet av smartphones och andra ”Internet of Things”-enheter ökar snabbt. I takt med ökningen, så släpps nya tekniker som möjliggör utveckling av mer avancerade tjänster. Ett exempel är Web Real-Time Communication (WebRTC). Den här typen av tjänster konsumeras oftast sporadiskt under kortare tidsintervall. Även cloud computing har drastiskt ökat i popularitet under de senaste åren. Hög elasticitet samt möjligheten att allokerera datorresurser på begäran har medfört att utnyttjandegraden av datorers kapacitet kan höjas. Flexibiliteten att snabbt kunna alloker och frigöra resurser möjliggöra att tjänster kan utvecklas för att utnyttja upp- och ner- skalningsmöjligheterna bättre, även för kortvariga lastökningar.

Forskningen i rapporten riktar in sig på ett system för videoströmning mellan användare i realtid baserat på WebRTC. Inkommande strömmar hanteras av Multipoint Control Units (MCUs), som har som uppgift att vidaredistribuera strömmarna till andra användare som vill spela upp strömmen. Horisontell skalning och cloud computing har mycket gemensamt. Det underliggande arbetet till den här rapporten fokuserar på ett extremfall, där varje inkommande videoström hanteras av en enskild MCU.


Acknowledgements

This thesis was carried out at Ericsson Research in Kista, Sweden. I am grateful to Ericsson and in particular Stefan Hellkvist, that I got the opportunity to carry out this thesis work. Stefan, with his deep technical excellence, has been a great resource to turn to when I was in doubt. Throughout this thesis, he has shown true commitment and an open mind about the work, to let me steer and complete the project.

I would like to express my gratitude to my academic supervisor and examiner, Johan Montelius. He has put his trust into my work, allowing me to carry out this thesis freely. He has always been there to answer my questions.

Finally, I would like to thank all employees at Ericsson Research that I have come in contact with, for your support and willingness to help me.

Stockholm, June 22, 2015
Eddie Kämpe
# Contents

1 Introduction ........................................... 1  
   1.1 Thesis motivation ................................. 1  
   1.2 Problem description ............................... 3  
   1.3 Purpose ........................................... 3  
   1.4 Goal ............................................... 3  
   1.5 Benefits, ethics and sustainability .......... 4  
   1.6 Method ............................................ 4  
   1.7 Delimitations ..................................... 4  
   1.8 Outline of the thesis ............................. 4  

2 Theoretical background ................................. 7  
   2.1 Cloud computing .................................... 7  
   2.1.1 Infrastructure-as-a-Service .................. 7  
   2.1.2 Platform-as-a-Service ......................... 8  
   2.1.3 Software-as-a-Service ......................... 8  
   2.2 Virtualization technologies ..................... 9  
   2.2.1 Hypervisor-based virtualization ............ 9  
   2.2.2 Container-based virtualization .......... 10  
   2.3 Web Real-Time Communication ................... 11  
   2.4 Multipoint Control Unit ........................ 12  
   2.5 Kurento Media Server ............................ 12  
   2.6 Chaos Monkey .................................... 13  

3 Methodology ......................................... 15  
   3.1 Research process ................................ 15  
   3.2 Philosophical assumptions ....................... 15  
   3.3 Research methods ................................ 16  
   3.4 Research approaches ............................. 16  
   3.5 Research strategy and design ................. 17  
   3.6 Data collection .................................. 17  
   3.6.1 Quality Assurance .............................. 17
4 Analysis

4.1 Target system .............................................. 19
4.2 Scope ......................................................... 20
4.3 Test environment .......................................... 20
   4.3.1 Hardware and Software ............................... 21
   4.3.2 Benchmarking scenario ............................... 21
4.4 Result ......................................................... 22
4.5 Summary ...................................................... 25

5 Existing solutions ............................................ 27

5.1 Requirements ............................................... 27
5.2 OpenStack .................................................. 28
   5.2.1 Docker Heat plugin ................................... 28
   5.2.2 Nova Docker driver ................................... 29
   5.2.3 Comments .............................................. 30
5.3 Apcera Hybrid Cloud Operating System .................... 30
   5.3.1 Comments .............................................. 31
5.4 Summary ...................................................... 31

6 Implementation ............................................... 33

6.1 Architecture ............................................... 33
6.2 Scope ......................................................... 33
6.3 Management component ..................................... 34
   6.3.1 Template ................................................ 34
   6.3.2 Scheduling ............................................. 36
   6.3.3 Crash policy .......................................... 37
6.4 Node agent .................................................. 38
   6.4.1 Heartbeats .............................................. 38
6.5 Management: Web UI ........................................ 38
6.6 Bootstrapping the system ................................... 39

7 Evaluation ....................................................... 41

7.1 Scope ......................................................... 41
7.2 Test environment ........................................... 41
   7.2.1 Hardware and software ............................... 42
   7.2.2 Test scenarios ......................................... 42
7.3 Result ......................................................... 43
7.4 Summary ...................................................... 43
8 Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Discussion</td>
<td>48</td>
</tr>
<tr>
<td>8.1.1 Reliability and validity</td>
<td>49</td>
</tr>
<tr>
<td>8.2 Future work</td>
<td>50</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Target architecture</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Scaling methods</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>The SPI model</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Virtualization stacks</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Multipoint Control Unit</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Target architecture</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Benchmark scenario</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>CPU usage - Broadcasting</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>Memory usage - Broadcasting</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>CPU usage - Broadcasting with recording</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>Memory usage - Broadcasting with recording</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Docker Heat plugin</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Nova Docker driver</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>Conceptual architecture</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>JSON Template</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>Scheduling process</td>
<td>36</td>
</tr>
<tr>
<td>7.1</td>
<td>Crash policy - Time to take action</td>
<td>44</td>
</tr>
<tr>
<td>7.2</td>
<td>Crash report - From crash to detection</td>
<td>44</td>
</tr>
<tr>
<td>7.3</td>
<td>Scale up - Comparison</td>
<td>45</td>
</tr>
<tr>
<td>7.4</td>
<td>Scale down - Comparison</td>
<td>45</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Benchmark deployments .................................................. 21
4.2 Broadcasting (1-to-N): Resource usage per unit ................... 22
4.3 Broadcasting (1-to-N) with recording: Resource usage per unit 25

6.1 Scheduling strategies ..................................................... 37
6.2 Crash policies ............................................................... 37

8.1 Multipoint Control Unit: Resource usage per stream ............. 49
8.2 Multipoint Control Unit: Resource usage per stream with recording ......................................................... 49
Chapter 1

Introduction

The content of this chapter is meant to familiarize the reader to the context (Section 1.1) that this thesis sets focus on and the problem it aims to solve (Section 1.2). Moreover, the chapter introduces the reader to why the work is done (Section 1.3), what is the expected outcome (Section 1.4), and what impact on the network society it has (Section 1.5). Section 1.6 explains how the work will be carried out, and Section 1.7 describes what boundaries the thesis has. Finally, a complete outline of the thesis can be found in Section 1.8.

1.1 Thesis motivation

Web Real-Time Communication (WebRTC) [1] is becoming a widely adopted technology for real-time media services [2]. WebRTC is an open-source project that provides a common set of protocols with the intention to enable interoperability and communication between browsers, mobile phones and other “Internet of Things” devices [1].

There are many use-cases with WebRTC, however a use-case that enables end-users to share live video streams to one another is of particular interest for this thesis. To be specific, the thesis examines a streaming service, where end-users stream video live from their smart-phones to a Multipoint Control Unit (MCU). The backing architecture of the streaming service is depicted in Figure 1.1. A MCU is an application that accepts incoming streams and allow redistribution of the stream to offload the streamer’s device (i.e. resource usage and battery life). It has also the responsibility to record the stream to a persistent storage for later playback. The streaming session is set up by having the streamer’s User Equipment (UE), in this case a smart-phone, contact a signalling server that controls the available MCU instances.
A recurring aspect of all types of applications and services, not exclusive to real-time media, is scalability; how well can the system scale? Scalability is a significant factor in software development, both from a business and technology perspective, and should not be overseen [3]. Traditionally when reasoning about scale, one often refers to the expression to "scale up" (vertical scaling), that is, provisioning more resources to the server as the load increases. A less common approach historically, that has received attention in recent time, is to "scale out" (horizontal scaling). To scale out rather than to scale up, means adding additional servers to serve the load. The difference between the two methods is illustrated in Figure 1.2.

The NIST Definition of Cloud Computing [4] lists important properties of the cloud, including On-demand self-service and Rapid elasticity. More specifically, it means that adding and removal of compute resources can be done rapidly without communication with a human-being. Horizontal scaling aligns well with this mindset of cloud computing.
This thesis will analyse and measure the MCU in a streaming service with the architecture shown in Figure 1.1 to find a flexible Cloud Execution Environment (CEE) to deploy and run it in.

1.2 Problem description

When it comes to applications and systems, it’s not uncommon that they are manually set up on a fixed number of machines [5]. Each of the machines is allocated to run a specific service that one or more employees, working with operations, take care of. As the business grows, so will the number of machines. In the extent, if machines are treated individually, the system will become unmanageable. What does it mean to scale a system? It means to move from a statically provisioned system with care for individual machines, to a dynamic environment where machines can come and go [5].

This thesis aims to take the necessary actions to find a flexible CEE to deploy and run the streaming service in. Incoming streams should be handled by a separate MCU instance. The following aspects will be evaluated:

- Can the MCU be treated as an ephemeral component?
  - What effect does it have on the overall system?
- How fast can the CEE respond to a crashed MCU?

1.3 Purpose

The thesis presents the process in which a Cloud Execution Environment for WebRTC-based streaming service was chosen. The decision is based on experiments, requirements as well as an evaluation of existing solutions.

1.4 Goal

The goal of the thesis is to evaluate existing CEEs to find the most useful in order to scale the streaming service (Figure 1.1) horizontally with as high resource usage as possible. The result of the evaluation will act as a foundation to the implementation and deployment. Finally the streaming service will be examined in a practical deployment in the chosen CEE under the presence of a so-called Chaos Monkey system [6] that randomly kills components.
1.5 Benefits, ethics and sustainability

Along with the explosion in number of computer systems comes the increasing energy consumption [7][8]. An eye-catching fact is that many machines run at low utilization [9]. One of the aspects in finding a suitable CEE, as this thesis investigates, is to reduce as much of the overhead as possible. The reduction of overhead opens up for additional work to run on each machine, or be it, run on fewer machines. This thesis seeks an environment in which the streaming service can be dynamically scaled, both in and out (horizontally), based on the changing load [10].

1.6 Method

To distribute, scale and manage systems in the cloud, it is necessary to understand what footprint and requirements the system has. Quantitative experiments, namely benchmarking, will be run to measure the resource usage of the streaming service that is considered. The aim of this thesis is to present how the Cloud Execution Environment (CEE) was chosen for the WebRTC-based streaming service. To conclude what CEE is preferrable, a set of requirements will be defined. Finally, a solution to the problem described in Section 1.2 will be proposed based on the comparisons. The proposed solution will be implemented and/or deployed depending on what is the outcome of the research.

1.7 Delimitations

The streaming service uses Kurento Media Server (KMS) [11], further described in Section 2.5, an open-source software implementation of a MCU. The implementation is under active development, meaning that it can not be treated as stable. It has known flaws in the design and regularly crashes under increasing load. This thesis will not include looking for different solutions of the MCU software, but rather takes the approach "let it crash" and the approach provides a level of fault-tolerance at the execution environment layer.

1.8 Outline of the thesis

Chapter 2 provides the reader with a theoretical background to the problem, definitions and explanations about the terminology that is used in the next-
1.8. OUTLINE OF THE THESIS

coming chapters. Chapter 3 discusses the methods and methodologies that
this thesis relies on. Chapter 4 contains in-detail information about the
target system, such as a practical scenario, requirements and constraints,
and benchmarking. Existing solutions are validated in Chapter 5 against the
defined requirements. The result of the system analysis and existing solutions
is used as an input to Chapter 6, which presents the final implementation of
the findings. The implementation includes both an architectural design and
a practical deployment of the system in the chosen execution environment.
Chapter 7 tests the deployed system during a staged failure of MCUs. The
final results as well as validity is discussed in Chapter 8.
Chapter 2
Theoretical background

This chapter is meant to introduce the reader to the concepts and ideas that are necessary to understand in order to find the thesis interesting. The reader might recognize one or several of the sections and may skip the familiar ones. Section 2.1 describes briefly what the cloud is and how it can be categorized. Section 2.2 explains a few of the virtualization techniques used behind the cloud.

2.1 Cloud computing

Cloud computing is a broad term, and there are probably as many definitions of the cloud as there are people working with IT, although a common ground is the SPI model (SaaS, PaaS, IaaS). The SPI model divides and categorizes the cloud into three service layers, each targeting a specific business: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) [4][12]. Figure 2.1 provides an overview of the responsibilities of the different layers. Infrastructure-as-a-Service provides the lowest abstraction of the three, and Software-as-a-Service the highest. A PaaS may depend on IaaS and SaaS may depend on a PaaS [13].

2.1.1 Infrastructure-as-a-Service

In the Infrastructure-as-a-Service (IaaS) layer, users rent infrastructure and resources in remote data centers. The IaaS layer covers the lower half of the stack shown in Figure 2.1, ranging from virtualization, servers, storage and network. The users are free to operate the infrastructure, such as choosing an operating system, middleware and install their applications. This model is closest to on premise server infrastructure. The users are themselves re-
sponsible to make sure that the operating system and software on top of it is patched and kept up to date.

2.1.2 Platform-as-a-Service

The Platform-as-a-Service (PaaS) layer covers everything but the application and data. Servers, networking, operating systems, and more (Figure 2.1) are abstracted away from the users. All the underlying infrastructure is provided and administrated by the company that offers the PaaS. Users may deploy their own applications and data on top of the platform. PaaS enables developers to build and rapidly deploy applications without worrying about the infrastructure beneath.

2.1.3 Software-as-a-Service

The highest abstraction of the listed cloud layers is the Software-as-a-Service (SaaS) layer. SaaS refers to managed applications or services by third-parties, such as Gmail by Google or Salesforce. The service vendors take responsibility for the installation, deployment, and everything underneath. The end-users are not required to have any knowledge of the underlying components.
2.2 Virtualization technologies

The key technology behind the cloud infrastructure is virtualization [14]. Two important factors that motivate the virtualization are isolation and rapid elasticity, both defined in The NIST Definition of Cloud Computing [4]. With virtualized environments, two or more customers can co-exist on the same host without interference [15], so-called single-tenancy. Each of the virtualized environment is restricted to its own context and will not be aware of other environments unless specifically defined on the host. Virtualization per se is not a single solution, but rather a concept that includes a number of techniques [15]. This thesis will categorize and discuss two major techniques, namely server and container virtualization. Figure 2.2 illustrates how the virtualization stacks differ between the two techniques.

2.2.1 Hypervisor-based virtualization

A Virtual Machine (VM) is a machine that runs inside another machine, physical or virtual, and acts like a regular machine but has its hardware em-
ulated. The host machine that runs the VM is referred to as a Hypervisor or Virtual Machine Monitor (VMM) [15]. The hypervisor can be responsible for several VMs, which means that multiple end-users can effectively be isolated and simultaneously served within a single physical machine. Hypervisors are divided and classified into two types: Type-1 or Type-2 [15].

- **Type-1**: The hypervisor is loaded directly onto the physical server, and has direct access to the hardware.
- **Type-2**: The hypervisor is loaded inside an existing operating system that is already loaded on the hardware, just like any normal application.

A Virtual Machine runs its own operating system that does not necessarily have to be the same as the host machine. VMs are based on disk images, these can either be an operating system alone, or packaged together with software [14]. A VM can be paused and its state can be saved to a new image. The created image can then be used to boot the VM on another host or start additional VMs from that disk (state).

**Kernel-based Virtual Machine**

In Linux, Kernel-based Virtual Machine (KVM) is a common choice when it comes to server virtualization [16]. KVM turns Linux into a Type-1 hypervisor that runs guest operating systems within Linux processes with low virtualization overhead. KVM is included in mainline Linux since kernel 2.6.30 [17]. To accomplish the low overhead, KVM depends on hardware virtualization support from the processor (Intel VT and AMD-V technology). Virtual Machines are processes under Linux, and because of this fact, KVM inherits features from the host system such as the process scheduler, memory manager and a network stack.

**2.2.2 Container-based virtualization**

The concept of isolated processes and workloads has been around in the Linux world for a long time. Linux Containers (LXC) [18], OpenVZ [19] and FreeBSD Jails [20] are examples of container implementations. In container-based virtualization, the containers share the operating system with the host, and thus maintain a lower overhead by running on a single kernel [21]. Unlike hypervisor-based virtualization, container-based virtualization does not rely on hardware support to reach its full capabilities.
Docker

Although that many of the implementations of container-based virtualization are mature, the technology has not been widely adopted until the release of Docker. Docker does not try to provide a completely new concept but instead focuses on providing the tools and ecosystem behind it, to make the technology easily accessible. The first versions of Docker use LXC as the container driver. Docker now comes with an open-source project libcontainer as the default container driver [22]. Features in the Linux kernel, such as namespaces and cgroups, are used to isolate environments into containers [23]. The team behind Docker embrace the concept of "batteries included, but pluggable", which means that LXC and possibly other drivers can be switched over to.

Docker provides git-like commands to manage images. Images are layered file systems, that allows for re-use across containers. A container is based on an image, which contains immutable layers. The container adds an extra layer on top of the image layers, which the container can write to [24]. Any changes to the writeable layer, are just like in git, presented as a difference. The changes may then be committed to form a new image. To spread images amongst collaborators, Docker hosts a storage service named Docker Registry. The registry is publicly hosted, but developers can choose to run a private instance. Docker push and pull are analogous to git push and pull, and images are pushed to the remote registry and can later be retrieved with the pull command.

2.3 Web Real-Time Communication

Web Real-Time Communication (WebRTC) is an API that enables web browsers and mobile applications to stream data between them, without relaying the data over intermediary servers [25]. Audio, video and binary data is supported by the WebRTC API. To establish a peer-to-peer connection, a signalling server is required. The signalling server enables the peers to exchange Session Description Protocol (SDP) [26] data. SDP is a protocol used to negotiate properties of the media that should be sent such as the media format, encryption and codecs [27]. In addition to SDP, peers need to exchange data on how they can be contacted, that is, details about their networks. Peers may not have an own public IP, and are often placed behind firewalls and Network Acess Translators (NATs), therefore the need of an external broker, such as a Session Traversal Utilities for NAT (STUN)
server or a Traversal Using Relay NAT (TURN) server [25]. WebRTC uses Interactive Connectivity Establishment (ICE) to find peer candidates. ICE is a protocol for NAT traversal for protocol that uses an offer/answer model [28].

2.4 Multipoint Control Unit

A Multipoint Control Unit (MCU) refers to a unit that manages multimedia content. The MCU can be a piece of hardware as well as a software implementation [29]. The core feature is to redistribute an incoming stream to multiple consumers. In addition, software implementations provide a more extensive set of features, such as transcoding and saving the stream to disk.

Figure 2.3 illustrates how a MCU is used to redistribute a stream to multiple consumers. To move the load from the streamer to the MCU means that the end-user’s device can remain simple and cheap. Furthermore the service provider has better control over the system’s performance. A particularly good use case is to place the unit as close to the consumers as possible to reduce the Round-Trip Time (RTT) [30].

2.5 Kurento Media Server

Kurento Media Server (KMS) is an open-source implementation of a media server with support for WebRTC [11]. The media server is capable of receiving incoming streams, redistribute streams to consumers, recording and
transcoding between various formats. The KMS functionality is exposed to developers by the Kurento API [11]. KMS will take the role of a MCU in this thesis. As mentioned in the delimitation of the thesis (Section 1.7), the software is under active development and thus can not be treated as stable.

2.6 Chaos Monkey

Failures in systems are undesirable, but yet they are hard to foresee and eliminate prior to an incident [31]. Instinctively, failures are avoided as long as possible. Netflix chose to do the opposite, they force themselves to operate in an unpredictable environment [6], using a so-called Chaos Monkey software that randomly kill off virtual machines and processes. They use their destructive Chaos Monkey system during office hours, so that the appropriate staff can be alerted and respond to the error. The project is open-source to encourage other developers to adapt to this concept.
Chapter 3

Methodology

The purpose of this chapter is to provide an overview of the research methods and methodologies that the thesis relies on. Section 3.1 describes the research process. Section 3.2 explains the philosophical assumption that is used, as well as why others are discarded. Scientific methods are listed and held against each other in Section 3.3. The research approach and research strategy are described in Section 3.4 and Section 3.5 respectively. Section 3.6 focuses on the data collection techniques used in this research.

3.1 Research process

To be able to answer and reason about the problem statements in Section 1.2, the thesis refers to and applies a set of research methods, more specifically, quantitative research methods. The nature of the research, to evaluate a system’s performance and explore improvements, is of quantitative character and the thesis applies quantitative research methods to reach the result. The research is conducted on experiments with numerical data rather than peoples opinion or behaviour, hence the need for quantitative research methods over qualitative research methods.

3.2 Philosophical assumptions

In quantitative research, there are two philosophical assumptions that are more relevant than others; realism and positivism [32]. With realism as a mindset, results are sought by observations, that should be repeatable, of an objectified view of the environment. The assumption relies on measurements to deduct truths and detect connections between variables. Positivism is similar to realism in the sense that the research relies on experiments to
discover facts about reality, but differentiates the perception of the object and the object itself. Both of the philosophical assumptions are independent of the researcher’s theories and beliefs [33]. The research in this thesis is conducted from a realism stand-point and objectively observes rather than objectively interpret (positivism) the system that is evaluated.

3.3 Research methods

Research methods in quantitative research can be categorized into four distinct choices: experimental, descriptive, fundamental and applied research [32]. In experimental research, both variables and the correlations between them are of paramount importance. The variables can be altered betwixt experiments to observe the changing output. Descriptive research or ”survey research”, as it is called in some cases, relies on questionnaires and interviews to retrieve data [34]. In both cases, descriptive research requires a group of individuals to share their personal opinion. Fundamental research, often called basic or pure research [32], aims to expand knowledge without necessarily referring to a practical application [35]. Empirical data is gathered to expand and formulate knowledge [34]. In contrast to fundamental research, applied research aims to solve practical issues or answer questions related to practice [35], that can be achieved by building upon a foundation, often existing work or data from previous work [35]. The thesis work sets focus on and examines a particular system (described in Section 1.1), hence why it best relates to applied research.

3.4 Research approaches

The research has a clear foundation and is centred around the problem statements in Section 1.2. Throughout the thesis, processes and methods are used to derive conclusions whether these statements are sound or not. This approach is referred to as deductive reasoning and is one of the common approaches in quantitative research [32]. In addition to deductive reasoning in quantitative research there is the abductive approach. Research that embraces abductive reasoning tries its best to draw conclusions from an incomplete set of data [36].
3.5 Research strategy and design

In order for the research to be direct and concise, there should be a distinct strategy that includes a plan [37]. The research design acts as a guideline and a reference to how the research should be structured and carried out [32]. Experimental, ex post facto and case study are strategies considered in quantitative research. Research with an experimental strategy controls all variables that may influence the outcome [32]. In opposite, in a ex post facto research strategy, the research is based on previously collected data and the researcher is bound to analyse the already existing data [32]. Both strategies are objective, reliable and replicable, but can fail if not the appropriate tools and instruments are used, or if they have been misconfigured. Lastly, a case study is a particularly useful strategy to retrieve in-depth data about a specific case [38]. A case study may be flexible and it is possible that it introduces additional questions. While case studies excel in depth, they lack breadth [34]. Case studies are conducted on human participants, hence there is a chance that the participant will be biased from the experimenter [39]. The thesis work will follow the experimental strategy.

3.6 Data collection

The data collection in this thesis is done through benchmarking. Throughout the thesis there are two experiments that measure the performance of the sought Cloud Execution Environment. One holds and compares deployments of different virtualization techniques against each other, and the latter evaluates the recovery time in case of failure of the Multipoint-Control Units in the chosen environment. The experiments are described in Chapter 4 and Chapter 7 respectively. The test environments are described in detail in the individual chapters, as well as the used hardware and software.

3.6.1 Quality Assurance

Data means nothing without quality assurance. Without any quality assurance, the data could be generated for the sake to support the researcher’s theory. Quality assurance of the collected data in Chapter 4 and Chapter 7 in form of reliability, replicability and validity are further discussed in Chapter 8.
Chapter 4

Analysis

In order to conclude which Cloud Execution Environment (CEE) that is most suitable for a particular system, it is important to understand how the application works: what requirements and constraints the application imposes on its environment. This chapter will extract the requirements of the streaming service and evaluate its resource usage in different virtualization techniques.

4.1 Target system

The system seen in Figure 4.1 will act as a reference point for the evaluation. It consists of a signalling server, a static number of MCUs, and a web system serving the User Equipments (UEs). The end-users stream video live from their smartphones to one of the available MCU instances. The stream session is set up by having the UE talk to the signalling server, which decide on a free MCU instance. When the stream is set up, one or more users can consume it. The MCU instance is also responsible of recording the stream. The recorded video is then saved to a mounted NFS storage, it is however not in the scope of this thesis to change or evaluate this behaviour.

As described in Section 1.1, the initial seed of the thesis was the thought of scaling the MCU horizontally rather than vertically. In the extreme case, which is studied, each new incoming stream is handled by a separate MCU instance. The MCUs can operate completely in parallel because there is no intercommunication between them, thus no need for any synchronization.
4.2 Scope

The MCU plays the role of a multi-purpose component and handles both incoming and outgoing streams, as well as recording and transcoding. To get a better understanding of the resource consumption, a series of benchmarking will be run. Since the MCUs are supposed to treat one incoming stream at a time, it lies in our best interest to examine the resource usage of a single instance of the MCU software.

Kurento Media Server [11] is used as the software implementation of the MCU. The regular crashes caused by the MCU implementation put additional requirements that the execution environment should be able to restart crashed service as quick as possible. The number of available MCU instances is in direct relation to how many incoming streams that can be handled at a certain point in time.

4.3 Test environment

The benchmark does not solely look at the performance of the MCU but rather its resource usage in different deployments. The benchmark consists of three deployments, each with a different virtualization approach; no virtualization, hypervisor-based virtualization and container-based virtualization.
4.3. TEST ENVIRONMENT

4.3.1 Hardware and Software

To make the benchmark fair, the deployments are done on the same computer using the exact same versions of the software. The test computer is an Intel Core i7-4610M 4 cores @ 3.00GHz and 16 GB RAM machine. All of deployments are running Ubuntu 14.04 64-bit LTS (long term support) together with Kurento Media Server v5.1.1 3.g11d388e. Table 4.1 lists the virtualization software that is used in each deployment.

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Virtualization software</th>
</tr>
</thead>
<tbody>
<tr>
<td>No virtualization</td>
<td>-</td>
</tr>
<tr>
<td>Virtual Machine</td>
<td>KVM QEMU v2.0.0</td>
</tr>
<tr>
<td>Container</td>
<td>Docker v1.5</td>
</tr>
</tbody>
</table>

Table 4.1: Benchmark deployments

To measure and collect data from the tests cAdvisor was used. cAdvisor is an open-source project that collects system resource usage, aggregates the data and exports the result [40]. It was configured to push the result into a time series database, InfluxDB [41], so that the result could be retrieved and analysed.

4.3.2 Benchmarking scenario

The test scenario is designed to cover the normal case, that is, one streamer and multiple consumers (1-to-N broadcasting, illustrated in Figure 4.2). The system starts out in an idle state, Kurento Media Server (KMS) is running but has no load. It provides valuable information about the minimal resource requirements of KMS. The benchmark of the different deployments, is done in two versions: one with recording of the stream and one without. The flow of test scenario is the same and the steps are defined as:

1. Kurento Media Server is running in an idle state
2. one user starts to stream
3. one consumer opts in
4. Repeat step 3 gradually until there are ten consumers
4.4 Result

The first set of benchmarking was done without recording the stream. Figure 4.3 and Figure 4.4 show the CPU and Memory usage respectively of the different deployments as the load increases. Table 4.2 summarize the result and splits it into usage per single unit (streamer and consumer) to show the expected raise in terms of each resource separately. The second set of benchmarking was done with recording of the stream. The CPU and Memory usage seen in Figure 4.5 and 4.6 present that the consumption is almost the same as without recording. The results are summarized in Table 4.3, and split into usage per a single unit, both incoming and outgoing individually. There were no notable differences in network consumption between the different virtualization techniques, nor between the scenarios. An incoming stream consumes 75 kbps and an outgoing stream (watcher) consumes 80 kbps, these values are averages.

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Type of stream</th>
<th>CPU (%)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No virtualization</td>
<td>Incoming</td>
<td>3,90</td>
<td>11,00</td>
</tr>
<tr>
<td></td>
<td>Outgoing</td>
<td>1,81</td>
<td>6,73</td>
</tr>
<tr>
<td>Virtual machine</td>
<td>Incoming</td>
<td>3,48</td>
<td>38,00</td>
</tr>
<tr>
<td></td>
<td>Outgoing</td>
<td>2,36</td>
<td>-</td>
</tr>
<tr>
<td>Container</td>
<td>Incoming</td>
<td>3,98</td>
<td>10,20</td>
</tr>
<tr>
<td></td>
<td>Outgoing</td>
<td>1,85</td>
<td>5,99</td>
</tr>
</tbody>
</table>

Table 4.2: Broadcasting (1-to-N): Resource usage per unit
4.4. RESULT

Figure 4.3: CPU usage - Broadcasting

Figure 4.4: Memory usage - Broadcasting
Figure 4.5: CPU usage - Broadcasting with recording

Figure 4.6: Memory usage - Broadcasting with recording
4.5 Summary

The resource usage of the broadcast with recording compared to without, shows that there is a minimal overhead in doing so, regardless of the type of deployment. The low overhead indicates that there is no need to separate the recording from the multiplexing functionality.

Figure 4.3 and Figure 4.5 demonstrate that the CPU usage increases linearly with the number of consumers. The difference between the deployments is the virtualization overhead, where the deployment with a Virtual Machine (VM) is consuming most resources. The resource usage of the Container deployment is nearly identical with the no virtualization counterpart. In terms of memory consumption, Figure 4.4 and Figure 4.6 display that the usage between no virtualization and the container deployment is close to the same. The consumption is once again linear with increasing load. The Virtual Machine deployment is an exception in this case. The VM consumes practically the same level of memory throughout the test, even though the load increases. The VM seemingly allocates memory in advance which makes it hard to determine what exactly is being used by the application. None the less will the deployment with Virtual Machines consume more memory than the other deployments, especially since the idea is to spawn a new instance for every incoming stream.

Based on the benchmarking result, the thesis will focus on solutions using Container technology. It outperforms the Virtual Machine deployment in both CPU and Memory, and performs almost identically to the no virtualization deployment. The small overhead that the container means, is weighted up by the isolation that the container gives. Related but independent research [42] [21] [24] shows that container-based virtualization outperforms hypervisor-based virtualization under other circumstances than what have been discussed in this chapter.

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Type of stream</th>
<th>CPU (%)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No virtualization</td>
<td>Incoming</td>
<td>4.43</td>
<td>14.10</td>
</tr>
<tr>
<td></td>
<td>Outgoing</td>
<td>1.92</td>
<td>7.57</td>
</tr>
<tr>
<td>Virtual machine</td>
<td>Incoming</td>
<td>3.97</td>
<td>43.00</td>
</tr>
<tr>
<td></td>
<td>Outgoing</td>
<td>2.33</td>
<td>-</td>
</tr>
<tr>
<td>Container</td>
<td>Incoming</td>
<td>4.88</td>
<td>9.90</td>
</tr>
<tr>
<td></td>
<td>Outgoing</td>
<td>1.85</td>
<td>6.10</td>
</tr>
</tbody>
</table>

Table 4.3: Broadcasting (1-to-N) with recording: Resource usage per unit
Chapter 5

Existing solutions

With the conclusion from Chapter 4, that container-based virtualization performs almost identically to running the MCU without any virtualization, this chapter will look at existing CEE solutions with Docker support. Section 5.1 enumerates what the necessary requirements for such solutions are. Section 5.2 sets focus on Docker integration in OpenStack while Section 5.3 focuses on Docker integration in Apcera Continuum.

5.1 Requirements

To evaluate the existing CEEs, more specifically OpenStack (Section 5.2) and Apcera Continuum (Section 5.3), a set of requirements are inferred from the streaming service. The existing CEEs are installed and the streaming service will be deployed on top of the CEEs to test their capabilities.

1. **Private Docker images**
   
   The source code of the the streaming service should not be publicly available, hence the need to store Docker images privately.

2. **Privileged mode**
   
   The Multipoint Control Units mount a NFS disk to store recorded streams persistently. Mounting inside a Docker container is considered a privileged action. The CEE should allow containers to be started in 'privileged mode', to enable this action.

3. **Flexible scheduler**
   
   It should be possible to provide the scheduler with hints about which nodes that the scheduler should consider. The MCU containers should
be separated from the signalling server, and spread out over the remaining nodes.

4. **Container intercommunication**

Containers should be able to be linked together, so that they know where to send packets. The MCUs have to be able to connect to the signalling server.

5. **Health monitoring**

The health of the streaming service has to be monitored. Containers will periodically die, and the system should be able to detect these crashes and react accordingly.

6. **Topology awareness**

There are dependencies between components in the streaming service. Given a topology, the CEE should start the components in a correct order, so that dependencies are fulfilled.

### 5.2 OpenStack

OpenStack is an open-source project that controls resources within a cluster. OpenStack started as a joint project by NASA and Rackspace, but is now managed by OpenStack Foundation and is backed by over 500 companies and 24000 contributors [43]. OpenStack has a modular architecture, and contains a collection of components, including Nova (compute), Neutron (networking), Glance (image service), and Heat (orchestration). The components of OpenStack are separated and are developed in individual projects. The components are tied together and communicate over RESTful APIs. The current Docker integration in OpenStack can be divided into two approaches, namely through the Heat and Nova components.

#### 5.2.1 Docker Heat plugin

Heat is a component in OpenStack that is used to orchestrate applications. Heat uses templates as input to set up and deploy applications. Templates are used to describe an application’s topology, which can consist of composite services. Heat accepts both templates that are compliant with the Amazon Web Services (AWS) CloudFormation template format [44], and its own format, Heat Orchestration Template (HOT). Templates are parsed and results in the necessary OpenStack API calls to start the application’s stack.
5.2. OPENSTACK

OpenStack resources (instances, ip addresses, volumes and more) are allocated and set up accordingly. The Docker Heat plugin enables a new resource type, the Docker Container, to be specified in the templates. An overview of the plugin’s placement in OpenStack is seen in Figure 5.1.

![Figure 5.1: Docker Heat plugin (Adapted from [45])]()

5.2.2 Nova Docker driver

The second alternative to integrate Docker in OpenStack, is through OpenStack’s Nova component. Nova is the main component of OpenStack, which provisions virtual compute instances in the cluster, normally Virtual Machines (VMs), but also LXC and bare metal instances [46]. The compute instances are tracked and managed. Nova is the OpenStack equivalent to Amazon’s EC2. Each compute node has a hypervisor that controls the node’s instances. The Docker driver for Nova Compute is a hypervisor (seen in Figure 5.2) that utilizes the Docker Remote API to transparently provide the same functionality as other hypervisors, such as KVM and Xen. The driver was initially included in the Havana release, but is currently out-of-tree (Icehouse, Juno) [47], which means that it is not included in the distributions by default. Docker images are managed and controlled by Glance.
5.2.3 Comments

The Docker Heat plugin enables Docker containers to be placed on virtual or bare metal machines. The plugin does not provide a way to avail the Nova scheduler, which means that the developer (or end-user) has to manually specify through the template, on which of the statically provisioned machines the containers should be placed and run. Moreover, in this solution, OpenStack is only aware of the health of the host, not the containers in particular. The Docker Heat plugin violates requirement 3 (Flexible scheduler) and 5 (Health monitoring), and thus will not be considered as a suitable CEE for the streaming service.

The purpose of the Nova Docker driver is to encapsulate the functionality of the Docker Remote API to act as any other hypervisor. It is however, a difference between hypervisor-based virtualization and container-based virtualization, which makes the integration lack certain aspects. The functionality of the Docker Remote API is not fully exposed in the driver, hence the limitation of its capabilities. There is no way to start a container in privileged mode, nor can environment variables be specified. The driver lives out-of-tree, and needs more development before it is mature enough to be used. The integration fails to meet requirement 2 (Privileged mode), and will therefore not be further evaluated.

5.3 Apcera Hybrid Cloud Operating System

Apcera has launched its Hybrid Cloud Operating System (HCOS), which is a operating system for the cloud that targets the hybrid cloud. HCOS manages
compute resources within a cluster. The cluster can be a mix of public and private clouds. Apcera HCOS takes a policy-first approach to enable business rules to be applied. A policy can be either a quota (RAM, disk, networking), access control, staging (the application need to pass through defined pipelines), and bindings between services. Applications written in the most common programming languages are supported as well as Docker containers and bare operating systems. Application and services are dynamically bound, so that they can continue to interact even when a service endpoint is moved. HCOS treats Docker containers just as any other workload in the cluster.

5.3.1 Comments

Deployment and linking of Docker containers in Apcera Continuum are easy tasks, but the platform has restrictions that prevent the solution to be used as the CEE for the streaming service. Apcera Continuum violates requirement 1 (Private Docker images), 2 (Privileged mode), and 3 (Flexible scheduler). To run containers with custom images, the image has to be uploaded to the public Docker Registry. Moreover, there is no way to influence the scheduler where the containers are placed, the scheduler is limited to distributing the containers randomly. Finally at the time of writing this thesis, the platform is not fully compatible with Docker’s Remote API, so that containers can’t be started in privileged mode.

5.4 Summary

In this chapter, three integration points in two platforms have been tested and evaluated with respect to a collection of requirements inferred by the streaming service. All of the tested approaches have their individual drawbacks, and a common fact that the Docker ecosystem has not been around long enough for the integration projects to grow mature. There are many more projects that are evolving around container-based virtualization, like Google Kubernetes and Apache Mesos, however this thesis delimits its viewpoint to match its time limit. Since none of the evaluated approaches matched the requirements, the thesis work continues with an own implementation of a CEE.
Chapter 6

Implementation

This chapter will in detail go through the design of the orchestration system that was built. It includes the reader in what decisions that were taken, the reasons behind, and an explanation how the system works.

6.1 Architecture

Figure 6.1 depicts the architecture that came to be the foundation of the orchestration system. It consists of the services: a \textit{management component} described in Section 6.3, a storage, an image registry (Docker Registry) and node agents described in Section 6.4. Each of the services have been encapsulated into individual containers, and pushed to the private image registry on one of the nodes, explained more in detail in Section 6.6).

6.2 Scope

Prior to the design of the software architecture of the orchestration system, the scope was set carefully to prevent the thesis project from ballooning. The orchestration system will operate in the Platform-as-a-Service (PaaS) layer and assumes that a number of physical or virtual machines can be accessed over Secure Shell (SSH). To a beginning, the machines are expected to run an Ubuntu cloud image, but should be built to have additional distributions added later. In the case of machine failure, the system should be able to detect the crash, but will not try to boot and configure another as a replacement (more in detail in Section 6.3.3). As the orchestration system operates in the PaaS layer, it should be unaware of any application logic.
6.3 Management component

The management component is the heart of the system, it is responsible of bookkeeping of nodes, scheduling and monitoring. A node in this case refers to a machine, virtual or physical. Essentially, the management component exposes an Application Programming Interface (API) that can be consumed either through the Web UI or an external application to achieve the same effect. API centric design is a way to turn the system into a flexible solution that can be further integrated with and built upon. This type of architecture can also be found in OpenStack [48].

6.3.1 Template

A template is a way to define a topology of one or more tasks, the relation between them, scheduling hints (Section 6.3.2) as well as crash policies (Section 6.3.3). A task represents a service and defines the container configuration as well as the number of instances it should have. Templates are defined in JavaScript Object Notation (JSON) format and can be used to deploy the topology repeatable times.

An example of a template can be seen in Figure 6.2. The template defines a single task with the name redis. The redis task should have 4 containers running with the configuration specified under the attribute "config". The remote image "redis:latest" will be fetched from the remote Docker Registry.
6.3. MANAGEMENT COMPONENT

Figure 6.2: JSON Template

```json
{
    'redis': {
        'policy': 'reschedule',
        'instances': 4,
        'config': {
            'image': 'redis:latest',
            'port_bindings': {
                '6379': '2540'
            },
            'ports': ['6379']
        },
        'scheduler': {
            'filter': {
                'exclude': {
                    'tags': [['management', 'registry'], 'something_else'],
                    'ip_addr': ['10.0.0.3']
                },
                'include': {
                    'tags': ['dev']
                }
            },
            'strategy': 'Spread'
        }
    }
}
```

if not found locally. The Docker image exposes the port 6379, but is mapped to the host port 2540. In addition, the scheduler hints are read as:

- Include only nodes that have the tag 'dev'
- From the matching nodes, remove nodes that have
  - both the tag 'management' and 'registry'
  - or the tag 'something-else'
- Remove the node with the IP address '10.0.0.3'

Finally, spread the 4 containers evenly among the nodes that match the criteria, and if a container should crash, make sure it will be rescheduled with respect to the same scheduling hints.
Flexible scheduling was one of the requirements described in Section 5.1 that was violated by existing solutions. It should be possible to influence the scheduler where the containers are placed. An example is that a task requires the containers to read data that is only accessible from a subset of the nodes marked with a tag. The containers of the task should mount a volume that it can read from, or else it would crash. Another use case is that a specific node in the cluster should be dedicated to run one task only, to make sure the task gets the full capacity of the node.

The scheduler can be altered in terms of filtering and strategy. The filtering uses the "include" and "exclude" keywords, where "include" hints to the scheduler that only nodes that match the criterias should be included, whereas "exclude" hints to the scheduler that nodes matching the criterias should be excluded from the result. Multiple filters of both kinds can be chained. A scheduler strategy refers to the placement strategy amongst the node result from the filtering. Table 6.1 lists the three available strategies.

The combination between filtering and placement strategies makes the scheduler cover most use cases. The "filter" and "strategy" directions are specified in the templates (as seen in Figure 6.2).
### 6.3. MANAGEMENT COMPONENT

#### 6.3.3 Crash policy

Crashes are inevitable, it is in the nature in software, however what distinguishes good software from bad is how the crashes are handled. Crashes can be categorized into different fault domains: application, machine, rack, datacenter. The orchestration system focuses on machine and rack failures, where a single container will be referred to as a machine in the analogy and the node referred to as a rack. Crash policies are ways to deal with crashing machines and are defined in the template on a per task basis. There are four supported policies: restart, reschedule, delete and do nothing. An overview of the action that each policy take is seen in Table 6.2.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart</td>
<td>Restarts the container on the same node</td>
</tr>
<tr>
<td>Reschedule</td>
<td>Delete the container and pass a request for a new instance</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete the container completely</td>
</tr>
<tr>
<td>Nothing</td>
<td>No action</td>
</tr>
</tbody>
</table>

Table 6.2: Crash policies

The management system does not look to application failure but rather the environment the application runs inside. A container in Docker is aware of and runs one process until it stops, then the container stops with it. One could utilize the system to its full extent by embracing the "crash-only" concept [49], by letting an application error shut down the process, and thus also the container. Candea et al. [49] points out that it may be more tedious to try to recover from an incorrect state than starting over from a well-known state. If the crash policy is set to reschedule, another instance of the task will be passed through the scheduling pipeline and launched. In this fashion, the management system will make sure that there is always a certain number of containers backing a task at any point in time. Containers under a task with the crash policy set to ‘restart’ will be restarted on the same node as before. The container can be restarted $N$ times before it’s marked as crashed, where $N$ is configurable at installation.
6.4 Node agent

An agent is installed on each of the nodes. The agent’s responsibility is to retrieve and process jobs. A job is described in a JSON representation, and may instruct the node to start new, restart existing, or delete containers. The agent is responsible to keep track of the status of each container, report crashes to the management component, as well as regularly send heartbeats to inform about its existence. When a node agent retrieves a job to spawn new containers, it will parse the configuration and look at what image should be used. At first, the node agent will try to resolve the image dependency locally, but may failover to the private Docker Registry within the orchestration system. If the sought image is not yet in the private Docker Registry, the node agent will try to fetch it from the Docker’s publicly hosted Registry. As described in Chapter 2, Docker images consist of multiple layers, where each layer is cached individually. This behaviour supports the idea that layers can be shared between images. A node agent will fetch layers that it not yet controls, but may skip layers that are already cached.

6.4.1 Heartbeats

A node agent continuously sends heartbeats to inform the management component that the node is still in an operating state to receive additional work. The time that the contract between the node and the management component will hold, can be configured when upon installation. If the node’s heartbeat misses the deadline, the node will be suspected to have crashed. It will still be included in the system but will have low probability to have additional work scheduled on it, if not explicitly defined in the template scheduling hint. If the node fails to deliver three consecutive heartbeats it will be marked as ’down’, and will be excluded from the scheduling process. To deal with the consequences of the crash is left to solve in future work.

6.5 Management: Web UI

An administrator can control the system from the either the Management API or a Web UI. The functionality of the Web UI builds upon the Management API, but provides an easier way to interact with the system. The Web UI has features to add or remove nodes in the cluster, upload private Docker images, define and deploy templates. Deployed tasks can be easily scaled up and down, to fit the administrator’s needs.
6.6  Bootstrapping the system

To install the necessary components for the orchestration system to work without hassle, a component was built to automate the process. The bootstrapping component consists of a Web UI that allows the administrator to add nodes to the cluster. The allocation of nodes is done outside the context of the orchestration system. Nodes can be either bare metal or virtual machines, the orchestration system is unaware of any virtualization. During the thesis, the orchestration system has been tested only with a cloud Ubuntu 14.04 LTS image, but it should work with earlier and upcoming releases as well. The nodes are paired with asymmetric keys to grant the orchestration system access over Secure Shell (SSH). In addition, nodes can be tagged with arbitrary strings that can later be used to differentiate nodes in the cluster via scheduler hints (as described in Section 6.3.2).

One of many possible use-cases is to tag known low performing nodes in the cluster as "experimental". Workload that is of experimental character can then be hinted to the scheduler through the template to run on these nodes. Once the administrator has added at least one node, the process can continue. The installation procedure consists of setting up a private Docker Registry, the management component and finally the agent component on each node. When the installation completes, the bootstrapping component can be shut down. The Web UI of the management component offers identical functionality to add new nodes.
Chapter 7

Evaluation

The Cloud Execution Environment (CEE) described in Chapter 6 should react to application failures that escalate to container failures. To withstand failures, the system implements crash policies (restart, reschedule, delete and do nothing). This chapter evaluates the orchestration system’s ability to respond to failing containers. It measures the time that it takes for the orchestration system to restart or reschedule a containerized Multipoint Control Unit (MCU). The results are complemented with data about the crash discovery time, that is, from that the container is killed (by the Chaos Monkey) until it is detected and reported.

7.1 Scope

The measurements in this chapter exclusively target containers that run Kurento Media Server (KMS) and acts as Multipoint Control Units. A prerequisite to the experiments is that all of the nodes already have a cached version of the MCU Docker image. When a node has a cached version of the Docker image, it won’t have to fetch the image again from the Docker Registry.

7.2 Test environment

The test suite consists of the Cloud Execution Environment (CEE) with a cluster of four nodes. The streaming service (described in Section 4.1) is deployed on top of the CEE. The signalling server, the web system and the MCU are packaged into separate Docker images. The signalling server and the web system are enforced to be scheduled on the same node, whilst the MCUs are spread out over the remaining three nodes. This is done through
the template functionality described in Section 6.3.1. The reason behind the separation of the MCUs and the other software is to ensure intercommunication between the nodes. Furthermore, the placement strategy is set to ‘Random’ so that containers are not bound to, but may be rescheduled onto the same node.

7.2.1 Hardware and software
The four nodes are allocated machines from an OpenStack deployment, and each of the machines runs with 1 CPU core (i7-4610M @ 3.00GHz) and 2 GB RAM. The instances use an Ubuntu 14.04 64-bit LTS cloud image, and they have Docker installed on them. One of the nodes acts solely as a controller node with the management component and the private Docker Registry installed. The three other nodes are considered compute nodes and will accept workload. All of the instances are located within the same private network.

7.2.2 Test scenarios
The Cloud Execution Environment (CEE) is evaluated by the time it takes to handle actions, such as a crash, within the system. More specifically, the time it takes for crash policies (reschedule and restart) to complete is measured (1). In addition to the crash policies, the time it takes between that a container is killed by the Chaos Monkey until the CEE is aware of the crash is observed (2). Last but not least, the capabilities to scale up and down are measured (3).

(1) The experiment will measure 50 occurrences of a ‘reschedule’ action and 50 ‘restart’ occurrences. The time is measured from that the management component initiates the action until an acknowledgement is received from the node that carries out the action.

(2) Each of the agents has a monitoring process that keeps track of the containers running on the node. The monitoring is simple, it checks what containers are alive at a certain point in time and compares the result with the last known list of containers. The interval that the monitoring process runs, is configured at installation. There will be three data series in the “crash/report/time” experiment, with the interval set to 5, 2 and 1 second. The Chaos Monkey will kill three containers every minute. Due to the fact that the polling interval of the monitoring process is fixed, the kills are unevenly distributed so that the monitoring will not detect the crashes at the
same point in each cycle every time.

(3) The scale up and scale down actions are measured in 6 data series each. What differentiates the data series is the number of containers that are affected. The first data series will scale up and down 1 container at a time, the second data series handles 2 containers at a time, and so on. The last data series will affect 10 containers. The placement strategy is set to 'Random'.

7.3 Result

Figure 7.1 shows the elapsed time from that the management component issues a crash action until it completes. There is only a small variation in the time it takes to do a restart. The diagram shows that rescheduling is more expensive to perform, and the variation is higher. The rescheduling is a two shot action, a 'delete' job followed by a 'create' job, while the restart command is a one shot action. The two shot action is bound to take more time because the 'delete' and the 'create' commands are executed sequentially.

The node agents continuously poll the number of containers running on the node. Figure 7.2 presents the time that it takes for the CEE to receive information that the container has crashed. The time is measured from that the Chaos Monkey kills the container until the management component receives the crash report. The diagram shows three data series, each with a poll interval on the node agents. The larger the delay is, the higher variation of the results.

Figure 7.3 and 7.4 display the time it takes to scale up and scale down respectively. In total, 650 containers were started and stopped, over a course of 6 data series. The results shows that the more containers that are started in parallel the higher variation can be expected.

7.4 Summary

The tests were carried out on containers running the MCU Docker image, which was cached and available locally on the nodes. The scope of the experiments did not include evaluation of other Docker images nor the time it takes to fetch the images the first time, before they are cached. It should be stated that Docker images vary in size and thus the fetching would vary in time. To connect back to reality, the nature of the streaming service is so
Figure 7.1: Crash policy - Time to take action

Figure 7.2: Crash report - From crash to detection
Figure 7.3: Scale up - Comparison

Figure 7.4: Scale down - Comparison
that the images would be prepared on each of the nodes before it is used.

Figure 7.2 shows that the time it takes to discover and report a crashed container is in average half the polling interval of the node agent. This fact comes as no surprise, in a random distribution in time of crashes, the results will span between the worst case to the best case scenario. A subset of the data points are close to zero, which indicates that the report flow has no slow running code.

Based on Figure 7.3 and 7.4, it can be noted that a delete action takes more time than to create. Figure 7.1 showed that reschedule is considerable slow because of its two shot action (delete and create). The CEE could benefit if the two actions were separated. On a second thought, deletion of a crashed container does not have to be done immediately. The container is crashed and will not execute any code, which means that it will not affect anything negatively.
Chapter 8

Conclusion

The intention of this thesis was to find a suitable Cloud Execution Environment (CEE) for real-time media applications but with a particular streaming service in mind. In order to efficiently scale the Multipoint Control Units (MCUs) in the streaming service horizontally, an important aspect was to keep the resource usage each of the instances at a minimal level. The resource usage of a single MCU was examined in three deployments: no virtualization, container-based virtualization and hypervisor-based virtualization. Container-based virtualization in shape of Docker has caught a lot of attention from both developers and dev-ops. Docker containers proved themselves to perform almost equally to its counterpart with no virtualization. The benefits of a container’s isolated context makes the low overhead worth the cost.

With the decision to use Docker for virtualization, the thesis work continued to look for a flexible environment that could manage Docker containers. At the time of the thesis, Docker is still a fairly young project (start March 2013). All the attention around Docker, have led to many evolving platforms to orchestrate containers. Requirements were extracted from the streaming service to set a distinct line, which the platforms were evaluated against. The thesis delimits itself to explore two platforms, OpenStack and Apcera Continuum, where OpenStack is a community-driven open-source project and Apcera Continuum a commercial alternative. The platforms were tested practically by trying to deploy the streaming service upon them. Both of them suffered in various forms of being in an early stage of development. OpenStack and Apcera Continuum were neglected as suitable CEEs.

From the defined requirements, an architecture of a custom CEE was designed. The design was primarily to support the particular streaming service, but with generalization in mind. Sound ideas from both OpenStack
and Apcera Continuum were adopted into the architecture. Due to the fact that Docker containers are self-contained, the CEE turned out to be a generic solution. Any type of workload that is containerized in Docker images can be deployed and run. Finally, an evaluation of the newly developed CEE was conducted.

8.1 Discussion

The success of finding a suitable Cloud Execution Environment (CEE) for the streaming service, is based in the thesis experimental way of conducting the research. The initial thought was to find an existing solution, but it turned out that due to the immaturity of the existing projects that were examined, the best path was to put them aside and develop a custom-tailored CEE. If the thesis project had been strictly bound to existing solutions, it would have ended up in a situation where the requirements had to be revisited and functionality dropped. To iteratively move forward based on previous results relates back to the software industry, where agile development has became a hot trend. To continuously review and steer a project, seems to be a good path to go in the rapidly evolving field of computer science.

Chapter 7 presents numbers on the CEE’s capabilities to respond to failure. The seed of the thesis work was the unstable MCU software. A MCU should handle one and only one incoming stream at a time, and instead let the number of MCU instances be the controlling factor of allowed streams. The results from the experiments show that new MCU instances can be created on the fly as more resources are needed. It takes less than a second from that a scale up command is issued until it finishes. Container crashes are handled by defined crash policies in the CEE. From that a container crashes, it can be restarted in less than a second, or rescheduled within 1.5 seconds in average. From the CEE’s point of view, MCU instances can be treated as ephemeral units. The streaming service needs to provide fault-tolerance in the context of streaming, that is, implement a failover so that when a unit crashes, the end-user may continue to stream without interruption.

The test results from Chapter 4, summarized in Figure 8.1 and 8.2, can be used to approximate how many MCU instances that a cluster can provide, although the numbers are most relevant for the test environment. The CPU usage is hard to translate to other processors, but at least gives a hint about the consumption. There is a flat 200 MB cost for every instance, this is what Kurento Media Server (MCU software) consumes in an idle state. At
larger scale, the management component in the CEE is most likely to be the bottleneck. All of the node agents send heartbeats back to inform about their existence. The heartbeats are sent over HTTP to the management component. At a certain point, the number of heartbeats will be too much for the web server to handle, and thus limit the system. To tackle a growing number of compute nodes, another instance of the management component could be started, and a load balancer added to distribute the load between the instances, although this is a task for future work.

<table>
<thead>
<tr>
<th>Type of stream</th>
<th>CPU (%)</th>
<th>Memory (MB)</th>
<th>Network IO (kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incoming</td>
<td>3.98</td>
<td>10.20</td>
<td>75</td>
</tr>
<tr>
<td>Outgoing</td>
<td>1.85</td>
<td>5.99</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 8.1: Multipoint Control Unit: Resource usage per stream

<table>
<thead>
<tr>
<th>Type of stream</th>
<th>CPU (%)</th>
<th>Memory (MB)</th>
<th>Network IO (kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incoming</td>
<td>4.88</td>
<td>9.90</td>
<td>75</td>
</tr>
<tr>
<td>Outgoing</td>
<td>1.85</td>
<td>6.10</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 8.2: Multipoint Control Unit: Resource usage per stream with recording

8.1.1 Reliability and validity

The experiments in Chapter 4 can be reproduced by the interested reader. The chapter carefully presents the steps that are included in the experiments, as well as the versions of the software. The experiments support reproducibility rather than replicability. Replicability is a weak form of validation [50], in which the exact same results can be achieved once again. Reproducibility on the other hand, means that the experiment can be reproduced to give the same fact, but not the exact same set of data. Similar comparisons between the container-based and hypervisor-based virtualization can be found in independent research [42] [21] [24]. The experiments in Chapter 7 are harder for the reader to reproduce, because it is based on custom software that is not open to the public. The results are not used to prove a hypothesis, and thus convince the reader about a fact, but are rather used to show to the reader what have been accomplished by the author.
8.2 Future work

The Cloud Execution Environment (CEE) is capable of managing crashing containers, but does not provide fault-tolerance itself. There are several single points of failures that can bring down the system, such as the private Docker Registry, the storage and the management component. If one of these components fail, the system will continue to operate, but can not accept any new workload to be deployed. In future work, these points could be addressed, so that the CEE could continue to accept new workload even during failures.

The evaluation in Chapter 7 showed that a 'delete' action is considerably slow compared to other commands in Docker. A reschedule of a container consist of a 'delete' followed by a 'create'. The 'delete' action is not required for the container to be rescheduled, and could be done at a later time in a background process.
Bibliography


[34] Donald Ary. *Introduction to research in education*, 2010.


