UiO : **Department of Informatics**
University of Oslo

A study of Linux Containers and their ability to quickly offer scalability for web services

Using Kubernetes and Docker

Ravn Steinholt

Master's Thesis Spring 2015

A study of Linux Containers and their ability to quickly offer scalability for web services

Ravn Steinholt

18th May 2015

Abstract

The purpose of the thesis is to explore and investigate Linux container's ability to allow for quick scaling of web services. This is important for many web service providers, because there is a desire to keep a low amount of resources running when there are low amounts of traffic. This can have economical and environmental benefits. Being able to scale up the amount of resources quickly, is important when the traffic increases to keep within service level agreements and to keep the service running for the customers. Linux containers are a relatively new type of technology that allows for virtualization and running of applications. The focus of the thesis became to compare this new type of technology with a more commonly used technology, which is virtual machines based on templates. Two experiments were designed to test the two types of technologies. The first experiment wanted to test how long it would take to launch a new web server, while the second experiment was used to test how long it took to replace a web server with a newly launched database server. Kubernetes and Docker were the software used for Linux containers and MLN and Openstack were used for the virtual machine templates. The results showed a significant difference in relation to how quickly these two types of software combinations were able to scale. The results and analysis showed that Kubernetes was able to scale faster than MLN. Given the results, it became evident that Kubernetes should be chosen over MLN when considering quick scalability. Even though research and theory regarding Linux containers gives an indication of it being significantly faster than virtual machines and templates, it is impossible to definitely conclude that all software that uses Linux containers would perform better than software using virtual machines and templates. It is however likely to assume that it would be the case in most setups.

Contents

List of Figures

Preface

In this section I would like to show my appreciation for some of the persons that has supported me throughout this thesis:

My supervisor, Ismail Hassan: For supporting and motivating me and providing great insight.

My family: My parents, Kristine and Jan, and my brother and sister, Vanja and Varg, for great support my whole life including this thesis and this 5 year study programme.

My friends: For offering me support throughout the project.

Part I

Introduction

Chapter 1

Introduction

In the later years there has been an increased focus on web services such as online banking, video streaming, online trading and so on [10]. Services such as banks, tax registration, games, television etc are all gradually being more and more accessed through the Internet instead of through traditional methods. A survey done by Statistisk sentralbyrå (SSB) showed that approximately 90 % of the norwegian population between ages 16-74 was using online banking in 2014, 2nd quarter [27].

Downtime of a service can have a big financial impact on the corporation responsible. Customers may become disgruntled with and unsubscribe from the service or the company might have to give out compensations [24]. Whether a compensation has to be given out or not, depends on the quality of service requirements that are manifested in a service level agreement (SLA). The quality of service (QoS) requirements could be values such as response time, throughput etc. A violation of the values agreed upon, would force penalties on the service provider [10].

Due to the nature of web services, there will be a fluctuation in the amount of load during different time periods [3]. One way to deal with this fluctuation, is through the method of capacity planning. Capacity planning is to determine the amount of resources that are necessary in order to fulfill the quality of service agreement. It is based on an estimate of the maximum amount of users. Considering that it is based upon a maximum amount of usage, resources will be wasted in periods with less traffic. Even though it helps to ensure that the QoS are met, it does have its cost. Firstly the energy consumption. In large server farms a lot of electricity is required to run the computers and cooling facilities [24]. Electricity is not free neither in terms of finances nor the environment [15]. In 2013 the ICT (information and communications technology) eco system was measured to account for a little under 10 % usage of global electricity consumption [15]. Another issue is that the amount of uptime is directly related to the life span of hardware, which means that using more computing resources than necessary will reduce the lifetime of hardware [25]. Production and installation of new hardware is also costly for a company or organisation in terms of climate and financial resources. Also the larger amount of computers and services, the more time is spent by system administrators maintaining these systems which is time and resources spent unnecessary when there is no need for all that capacity. The mentioned problems are both true in the case of corporations renting cloud services and those using their own server farms, but there is also a cost in terms of finances when renting resources from a cloud provider. The web service provider then has to pay directly for the amount of resources used, and running more resources than required would be an unnecessary financial cost [24].

Considering all the problems with capacity planning, a more dynamic approach is desired by service providers. Dynamic allocation of resources would allow a web service to be able to deal with the most extreme loads, as well as reducing the amount of resources used when the traffic is lower [24]. One technology that has enabled system administrators to implement a more dynamic approach, is virtualization. By splitting the available physical resources into virtual machines, system administrators are able to both isolate important applications in the infrastructure as well as utilizing hardware more efficiently than when separating applications by physical machines [23].

Configuration management allows for better management and deployment of software by allowing new machines to be quickly deployed with all the necessary software for a specific service, faster and less error prone than when done manually [29]. It also simplifies the management of a large number of machines. This is ideal for web services with a large variance in load where new servers have to be configured and maintained regularly when in periods with an increased amount of traffic.

Conventional configuration management tools such as Puppet, Chef etc, implement infrastructure as code. Infrastructure as code means that most configuration actions can be automated through scripts and programming. Infrastructure as code targets to achieve automation and cost reduction of service management. The problem is that these types of configuration management tools are not specialized in dealing with web services running in cloud environments [29]. These conventional configuration management tools look at a group of computers as a group of individuals as opposed to cloud management tools. For web services with unstable traffic which require large and complex systems, it is better to abstract the hardware resources and look upon them as a pool of resources instead [7]. Both the process of writing the code for complex and large systems as well as executing the scripts, are time consuming [29]. This is problematic when the system should be flexible and fast to meet the quality of service agreements [24].

Virtual machine (VM) forking and templates are better equipped for cloud environments. VM forking is the process of cloning a VM into multiple replicas which then can be run on different hosts [13]. This is ideal when dealing with web services with fluctuating traffic, because it allows swift instantiation of more resources when needed [13]. New virtual machines can also be created from pre configured templates. It could either be an image template or a text file template. An image template will be based on a working virtual machine and new machines can then be instantiated using that template [3]. A text file template will have a specific syntax that would specify characteristics of the virtual machine [19].

A new type of software based on Linux Containers, Docker, is a new technology that challenges virtual machine cloning/templates. An application and its dependencies can be packed inside a container [23] and distributed and launched depending on traffic requirements. While linux containers also are virtualized, they do not virtualize hardware [23]. However, containers still provide the same isolation of applications as virtual machines[23] and they can be easily stopped and started like applications [17]. The resources used by a container can also be changed and configured while the container is running. This should in theory mean that Docker should bring some improvements over virtual machine forking/templates in scaling scenarios.

As this study is focused on web services that have quick and large variations in their traffic load, performance becomes an essential variable for these cloud management systems to increase the uptime of the web service. It is essential to get the service up and running as fast as possible to reduce financial costs because of unsatisfied users or to satisfy the quality of service agreements [10][24]. Depending on the service and the traffic, one more server or a couple of hundred may be required. It is therefore important that the cloud management tools are performing well in different scaling scenarios [10].

1.1 Problem statement

The problem of this study is to find out if Linux Containers are able to scale quickly when there are heavy decreases or increases in the amount of traffic for a web service.

The following problem statement was defined:

"Can Linux Containers provide quicker scalability for demanding web services with fluctuating traffic than traditional virtual machine templates?"

1.2 Thesis contributions

The desired impact of this study given significant results, is that it will aid system administrators of large scale systems with traffic fluctuations to make a better decision in terms of which software solution to implement for their system. Correct decision and usage of that solution will then increase uptime and flexibility of the services provided and allow for better usage of the available resources.

1.3 Thesis outline

The thesis is divided into five chapters:

Chapter 1 (Introduction) presents the motivation behind the research, the problem statement and discusses possible impacts of the research.

Chapter 2 (Background) will explain concept and technology that is important in regards to web services, Linux Containers, cloud management and virtual machines.

Chapter 3 (Approach) will explain how the experiments were designed, the reasoning behind them and how the setup was configured to perform the experiments. It will also discuss limitations and challenges of the experiments.

Chapter 4 (Results and Analysis) will present the data from the experiments and then use statistical methods to further analyze the data. The output of the statistical methods will then be further explained and investigated in terms of what they say about the data from the experiments.

Chapter 5 (Discussion and Conclusion) will look at what the results represent, further research, a summary and a conclusion of the thesis.

Chapter 2

Background

This section will use background literature and research to explain concepts and technologies in regard to web services, cloud computing, software configuration and virtualization.

2.1 Web service

A web service is described as "a software system designed to support interoperable machine-to-machine interaction over a network" [28]. Figure 2.1 shows the concept. Many web services have defined minimum performance requirements for their services. The specific requirements are known as a quality of service (QoS) and they are a part of a service level agreement (SLA). QoS could be variables such as response time, error rate, uptime etc. Breaching the SLA agreement will force a financial penalty on the service provider or the ones responsible for hosting the service if the maintenance of the service has been outsourced to a third party. The service providers therefore want to keep in line with the service level agreements, while at the same time use as little resources as possible [24].

In a multilevel web service, different parts of the architecture will be divided into isolated environments (virtual machines, physical machines or containers). A web application can for instance be divided into presentation and data layer [10]. The presentation layer could be provided by a web server and the data by a database server. This example is visualized in Figure 2.2. This means that the whole web service depends on different isolated services for it to function correctly, which means that each service must have been allocated enough resources to meet the SLA.

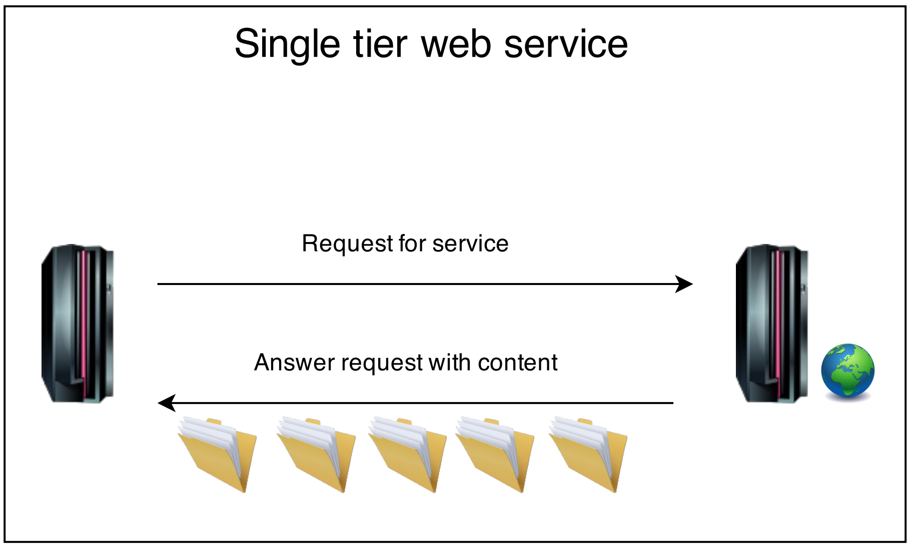


Figure 2.1: Single tier web service

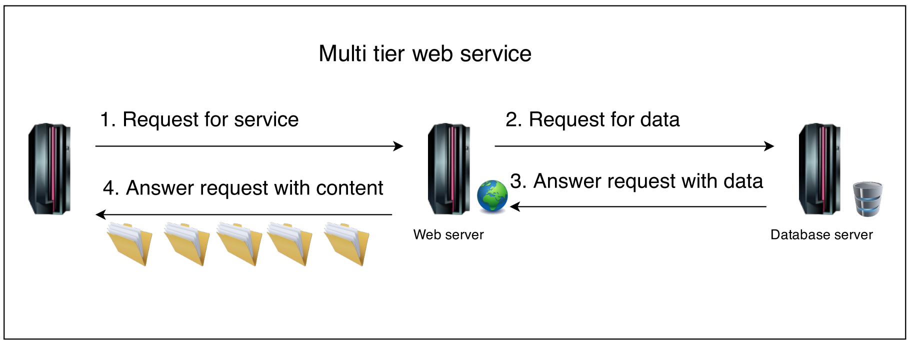


Figure 2.2: Multi tier web service

2.2 Scalability

In cloud management, scalability is achieved by either horizontal or vertical scaling. In horizontal scaling, the system is scaled by scaling up or scale down the amount of servers. It does not allow the system to alter the amount of resources a machine is allocated while it is running. This is opposed to vertical scaling where a machine's amount of allocated resources (e.g. RAM, CPU) can be altered without rebooting the machine [24].

2.3 Cloud computing/clusters/data center

A cloud can be described as a set of virtualized resources that can be managed as a unit. The cloud relies on the use of virtualization to more effectively allocate resources [9].

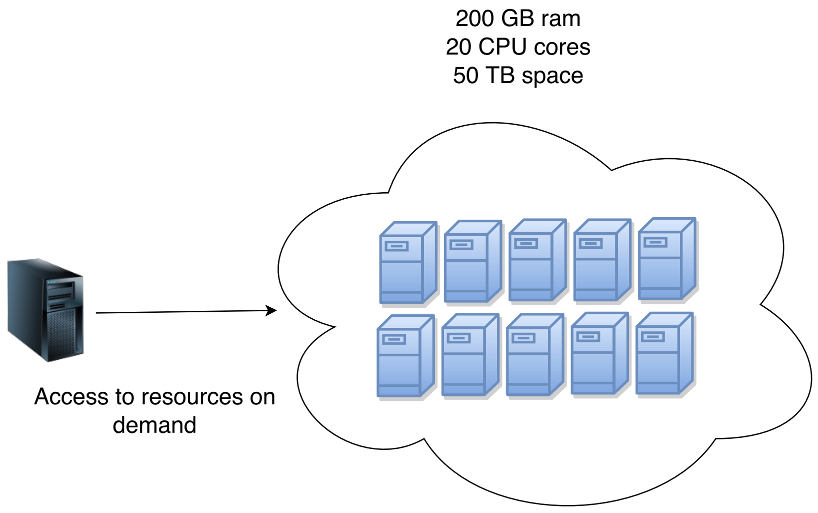


Figure 2.3: Visualization of a cloud

A cloud can be either public or private. A public cloud is a pool of resources that can be accessed through the Internet. The cloud provider owns the infrastructure of the cloud and resources are accessed through subscription based fees [9]. A private cloud is an internal data center only available to that business or organisation. Even though it belongs to a certain organisation or business, it can still be managed and administered by a third party [9].

2.4 Software Configuration Management

Software configuration management (SCM or CM) is used to centrally manage and deploy software solutions on machines in a network. It can be used for tasks such as installing every software needed for a web server, mail server, normal client PC etc. One of the main purposes of configuration management is to automate the configuration of machines [14]. It does this through the use of infrastructure as code [29]. The problem with manual configurations is that once a new machine is added to the network, then the same configuration procedure has to be done all over again even though the machine's function should be the same as an already configured machine [14]. Doing the same procedures multiple times is both time consuming and prone to errors [14]. It is time consuming because the script to configure for instance a web server can be run several times, but during a manual configuration all the commands has to be run again, configuration files has to be edited and so forth. Such a method is also prone to errors because the administrator might do a typo in the configuration file, install the wrong package and so on. Reducing time consumption and error rates are central values to companies or organizations because it indirectly increases productivity. Having pre-written scripts for installation processes also help to provide flexibility to the system administrator [14]. If, for instance, CM scripts have been written to install a web server on Ubuntu 12.04, but the new servers are running Ubuntu 14+, then it would be an easy task for the system administrator to change some of the code in the configuration script instead of having to do the whole procedure of installing and configuring files for each machine.

Another strong point of configuration management is its ability to allow servers and clients to restore a functional state [14]. If for instance the database server in the network had a software issue that required a reinstall of the operating systems, then the machine could automatically be restored into a functioning database server once more a lot more efficiently than using manual configuration and with less potential for errors [14].

Software configuration management consists of certain principles shown in the Figure below (2.4).

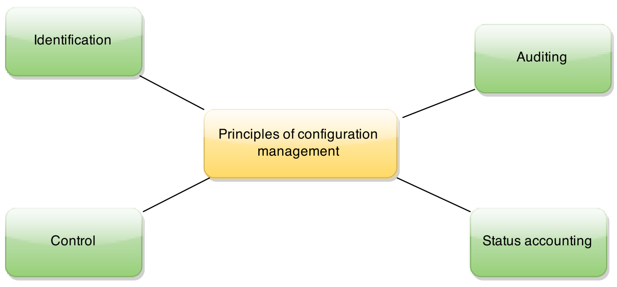


Figure 2.4: Principles for software configuration management

Identification is the process of identifying the different software items in an information technology system [12]. Identifying all the items and their version is necessary for the next principle which is control, because in order to control or change the item there has to be a way to reference that item. For instance if a web server should be stopped, then it would be necessary to identify the httpd process. Control is the actual managing of the different items in the IT system [12]. Status accounting is the process of recording and reporting all the changes to the configuration items. It includes information about what has been changed, when it has been changed, which items were affected by the change and what the status of the change is [12]. Audit is the process to ensure that the actions taken have actually been implemented according to plan [12].

These principles describe what an ideal configuration management system should consist of. However a lot of the CM tools are not able to fulfill all these principles on their own and need support from other applications as well. These applications together with the CM tool then become a part of the whole CM system. Employees and documentation is also part of this system that the CM tool cannot implement by itself. CM tools has to as a minimum implement the principle of control and identification, because the tool has to be able to identify the different parts of software in the IT system as well as being able to manage and change them. Several tools also implements part of the status accounting principle through logs and response messages.

2.4.1 Conventional configuration management tools

Conventional configuration management tools such as Puppet, Chef etc, implements infrastructure as code. They have been used and can be used to get web services automatically up and running [29].

The most common setup for environments using configuration management tools, is pictured in Figure 2.5.

The figure shows a client-server architecture which is used in many CM solutions such as Puppet and Chef. The PC in the middle of the figure is the configuration management server. It will contain all the scripts that will be used to configure the clients. The clients receive instructions either by the server pushing out the scripts to each client over a network, or by the clients doing regular pulls from the server to look for new or updated instructions.

The advantages of such an approach is first and foremost centralized management. Centralized management allows for simplicity, because it is easier to control one or a few machines instead of all the machines in a network. The system administrator will have access to all the scripts for that network in one place and security and access rules only need to be configured there. It also gives a better overview of the machines and their configurations [16]. With a good structure of the different configuration scripts and CM experience, the system administrator will be able to see what is installed and configured on the different machines. This will help when troubleshooting, which has to be done when there are errors in the configuration. The system administrator will be able to see all the steps previously done, whereas when doing manual troubleshooting many things are often tested at once with the administrator forgetting many of the steps that were taken. The disadvantages of such an architecture is that it has a single point of failure, meaning that if the server breaks down in any way, then the configuration management system in that network environment will cease to function [16].

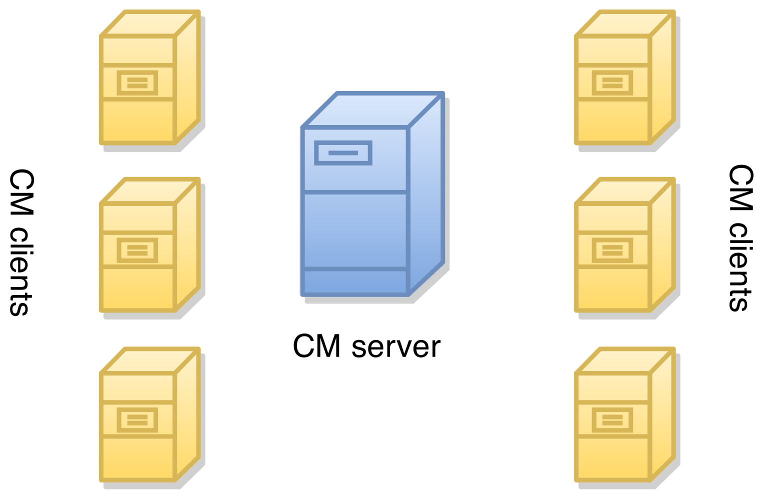


Figure 2.5: Common configuration management setup

In these kind of CM setups, the computer environment is being controlled through the use of different scripts. In these scripts two things are described: What should be configured and on which machines [29]? Each CM tool usually has its own unique syntax which are later compiled into other scripting languages. Depending on the software being used, the CM clients will either receive the script and the CM client software will then compile it into scripting languages such as Python or Perl, or receive already compiled scripts. The instructions are then executed by the client [21].

CM tools that function in a client-server architecture use either a pull or push method. A general consensus of the pull method for a CM system is shown in Figure 2.6.

As shown in the drawing, the client initiates the connection [20]. The CM software that uses the pull method are set up to send regular requests to the server asking for new instructions [20]. The server then has to find out which machine the request is from [20]. In many systems instructions are assigned to a certain machine by using their hostnames. If any instructions are found for that machine, then it has to check if the changes has already

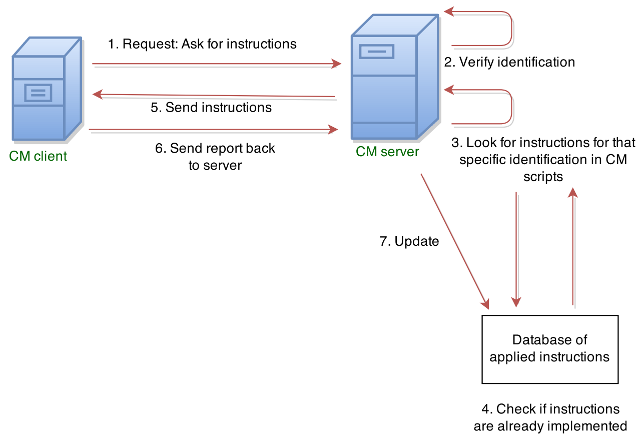

1. Request: Ask for instructions
2. Verify identification
5. Send instructions
6. Send report back to server
CM client
CM server
3. Look for instructions for that specific identification in CM scripts
7. Update
Database of applied instructions
4. Check if instructions are already implemented

Figure 2.6: Pull architecture

been implemented or not. In some systems this can also be done by the client [20]. Depending on the outcome of that action, it will either send the instructions to the client or reply that there are no more instructions to be implemented. The client will then send a report message back to the server that explains whether the instructions could be implemented or not along with other variables [20]. The drawing only specifies that instructions are found and transferred to the client and not exactly how this is done, because it varies between the different software solutions. In for instance Puppet, this list of instructions to be applied is called a catalog [20].

In a push architecture, all connections are initiated by the CM server. The CM server will look for instructions in scripts that are not yet implemented, and push those changes. The clients have to report back to the server, because the server needs to know whether it has to push the changes again or if scripts have to be modified. Figure 2.7 shows the general principle, but it will slightly differ between different configuration management software solutions.

One major advantage of the push architecture, is that it enables the administrator to send instructions to all CM clients with commands only executed at the server [8]. It will also instantly provide feedback if something went wrong and can then be corrected quickly [8]. It can also create a lot of load on the network during that period. One problem with the push architecture is that since more work is being done on the server, it can become overloaded quickly when it administers a lot of computers [8]. It also requires the server to know about all the hosts in the network, which can be a problem when booting new machines that have to be configured [8]. One advantage of the pull method is that there will be a stable load on the CM server, because the CM clients will ask

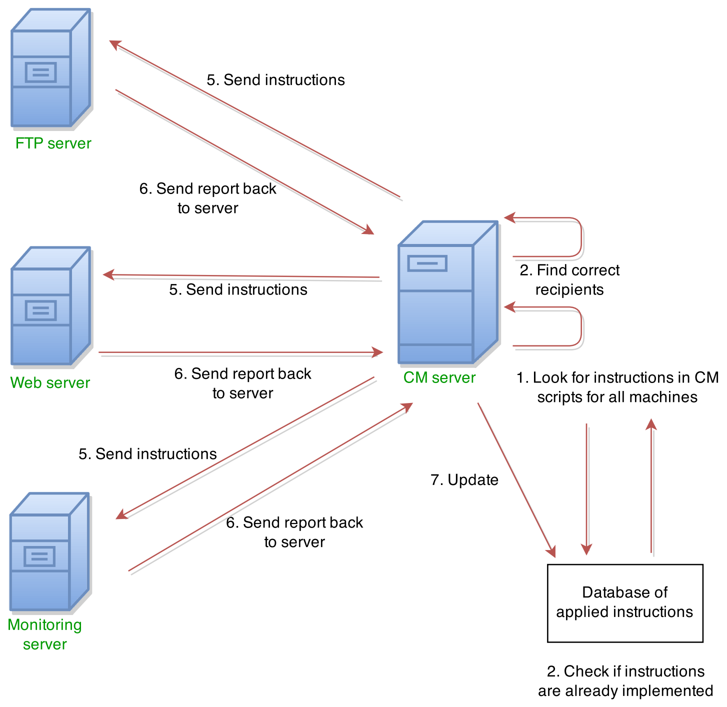

FTP server
5. Send instructions
6. Send report back to server
CM server
2. Find correct recipients
Web server
5. Send instructions
6. Send report back to server
1. Look for instructions in CM scripts for all machines
5. Send instructions
7. Update
6. Send report back to server
Monitoring server
Database of applied instructions
2. Check if instructions are already implemented

Figure 2.7: Push architecture

for instructions independently of each other [8]. It also allows for newly spawned machines to automatically receive instructions from the server [8]. Another advantage of the pull technology, is that the pull request from the client in itself identifies that the client is ready to receive instructions. A pull system has the problem that it is difficult to apply changes to many computers at the same time. For these reasons, pull systems are often better when small changes to a system are made regularly. For more drastic changes, a push system is often preferred.

2.5 Cloud and configuration management

The problem with the configuration management systems mentioned in the previous section, is that they are not specialized in dealing with web services running in cloud environments. This is because they are slow in execution of the scripts and the scripts can in large networks become very complex [29].

2.5.1 Hypervisor and virtual machines

A hypervisor is a software platform which allows several virtual machines to run on one physical host. Each virtual machine will have its own operating system and kernel. The hypervisor keeps the virtual machines isolated from each other and allows them to interact with the physical hardware. Each virtual machine has access to its own virtual hardware which then has to be translated into commands for usage of the physical hardware by the hypervisor [9].

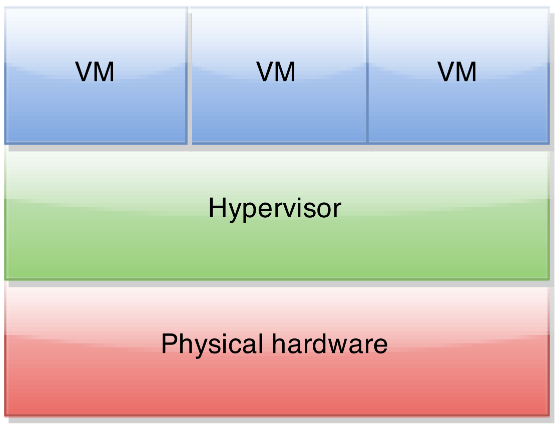


Figure 2.8: Hypervisor and virtual machines

Figure 2.8 shows the concept of the hypervisor and the virtual machines running on top of it. The hypervisor can also run on top of an operating system, it does not have to run directly on specialized hardware as shown in the Figure [9].

Many virtual machine technologies offers more than isolation between the different machines and effective resource management. By using virtual machine templates, a system administrator is able to easily start fully configured virtual machines [11]. A system administrator could for instance create a virtual machine and install a web server on it and save the image (often referred to as snapshot). The next time a new virtual machine is configured, it could be booted using that template which then would, after the initial boot process, create a fully operational web server [3]. This is known as image-based provisioning [3]. This can be applied to cluster or cloud management of web services by being able to boot new machines using pre configured templates based on the need for resources determined by traffic load [11] [3].

Even though virtual machines allow for isolation of machines for different roles in the network and allows a system administrator to allocate resources more effectively, they do have the problem that running multiple kernels uses more resources. Considering that it is the applications at the virtual machines that are the most important thing to separate when hosting web services, using resources on several kernels are resources spent unnecessary [23]. A problem with using virtual machine templates, is that each virtual machine still has to be booted using the template which consumes time [11].

2.5.2 Openstack

Openstack is an open source bundle of software tools that together allow an administrator to create and operate a cloud. Virtual machines can be launched using the resources available in the cloud which allows for horizontal scaling [18]. Image templates can be created using Openstack by saving a running virtual machine as a snapshot, which then can be booted from when new virtual machines are created.

2.5.3 MLN

MLN stands for manage large networks and it was developed to simplify deployment and management in a large scale virtualization structure. It allows the user to perform actions on groups/clusters of virtual machines instead of individual handling of virtual machines, which is the standard of most systems. These groups in MLN are called projects. File systems in MLN are copied from templates, which helps to automate the process of formatting disks when installing new systems. MLN has support for several virtualization platforms such as VMware, KVM, Xen, User-Mode Linux, Amazon EC2 and OpenStack. MLN has functionality for both managing (starting, deleting, upgrading) and deployment of virtual machines. Managing is done through the use of the *mln* command. New machines are deployed through the use of mln scripts(.mln) and the scripts are used as arguments for the mln command [2]. Figure 2.9 below shows how MLN works.

2.5.4 Linux containers

Linux containers is a relatively new virtualization technology that differs from the virtual machines. It is implemented in the Linux Kernel. A container is able to hold isolated processes and resources without the need

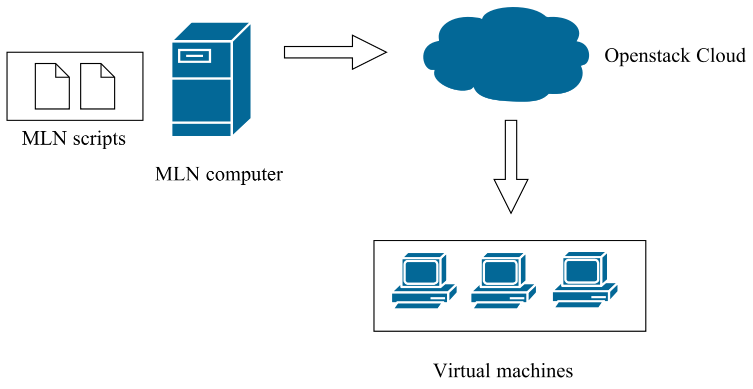


Figure 2.9: Visualization of MLN

for virtualization of the hardware. Containers are able to run their own operating system, but the kernel is shared between all the containers. Each container will have its own filesystem and network interfaces [23]. Each container will also have its own networking layer and processes. Considering that Linux Containers are only a virtualiazation layer of software on top of an operating system, they can easily be started and stopped like other applications [17]. Containers enable the administrator to have the means for both vertical and horizontal scaling.

Figure 2.10 shows the concept of containers. Since the kernel is shared between the containers, it therefore uses less resources than virtual machines [23].

2.5.5 Docker

With the increased focus on containers, they are being explored in terms of configuration management. Docker is a piece of software that allows an administrator to pack an application into a container with all its dependencies [23]. This differs from more conventional configuration management systems which would provide a list of dependencies to the client and rely on the client's package management system to handle it [22]. One thing Docker lacks compared to configuration management tool such as Puppet and Chef, is the distribution. In a standard Docker system, containers have to be transferred to the client through manual methods such as ssh [22]. This functionality is however implemented by Kubernetes,

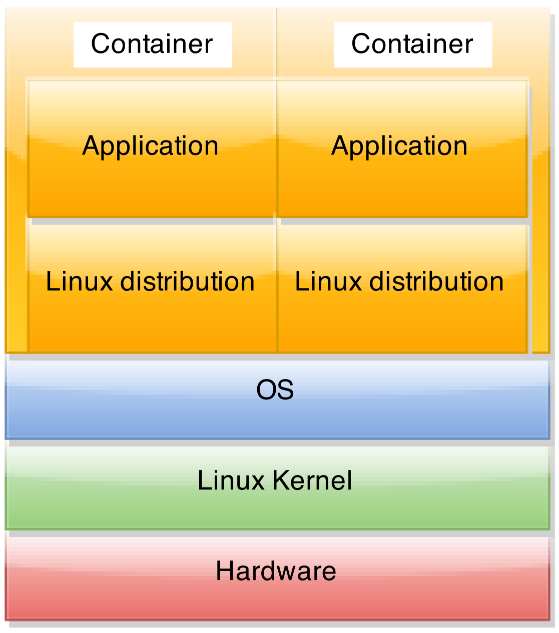


Figure 2.10: Linux containers

and other docker orchestration tools.

When the Docker service is started, it creates a virtual network interface called docker0. Docker0 creates a virtual ip address range and new containers are created within this network. The bridge allows the containers to communicate with each other in addition to the host. Without this bridge, the containers would not be able to connect to other containers or to the Internet and they would not be accessible from any container or host. Figure 2.11 shows the concept [4].

In order to run a container using Docker, the command *docker run <baseimage>* is used. The base image could either be a local customized image or an image that will be downloaded from the Docker hub, which is a centralized database of pre-configured docker images.

2.5.6 Apache Mesos

Apache Mesos perceives a number of machines as one abstract block of resources. This allows the software to easily manage large data centers/cloud

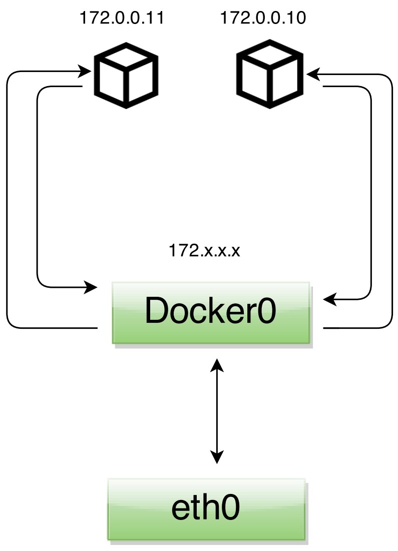


Figure 2.11: Network setup in Docker

environments. Mesos is split into three different components: Master, slave and framework. The Master is a daemon responsible for controlling the slaves, the slaves are daemons running a task and the framework is a mesos application. A framework consists of a scheduler and an executor [22]. Some of the frameworks are able to distribute and manage docker containers by using the block of resources available.

2.5.7 Kubernetes

Kubernetes is a cluster manager for Docker. It allows the system administrator to schedule and launch containers, while Kubernetes automatically selects the servers on which the containers are launched [6]. Similar to Mesos, it abstracts physical machines into a pool of available resources. The two most important roles in a cluster managed by Kubernetes, is the master and the minion roles. The master is the machine which has control over a group of minions. Minions are responsible for running tasks given to them by the masters [6].

The minions report backs to the master about eventual errors or successes

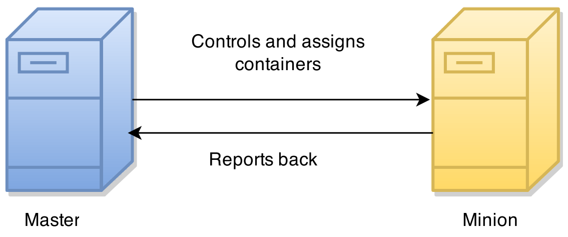


Figure 2.12: Master-minion relationship in Kubernetes

in regards to the creation of containers. Kubernetes mainly consists of three different work units: Pods, Replication Controllers and Services. Pods represents a single container or a group of containers. The containers in a pod are based on the same application and will be running the same Docker image. Replication Controllers handles the pods. This means that the Replication Controllers are responsible for keeping track of their assigned pods and manage them so that the specified amount of pods are always running in the Kubernetes cloud [5]. Figure 2.13 shows the concept of replication controllers.

Services are used to direct traffic to the appropriate containers. Containers in Docker will be using a virtual network on the physical host and services are responsible for directing that traffic to the containers from an external machine. By being assigned to a Replication Controller, they can redirect all traffic to the correct machines and then to the containers [5]. It is done by the master machine sending a request to the minions to change their firewall to forward traffic to the right containers. Figure 2.14 shows the concept of services in Kubernetes.

There are some important commands in Kubernetes that are important to know in order to operate the cluster. */opt/bin/kubectl get po* will show all the pods running on the cluster. In order to shut down a container in Kubernetes, the following command has to be used: */opt/bin/kubectl stop pods,replicationControllers -l run-container=apache4*. *run-container=apache4* is

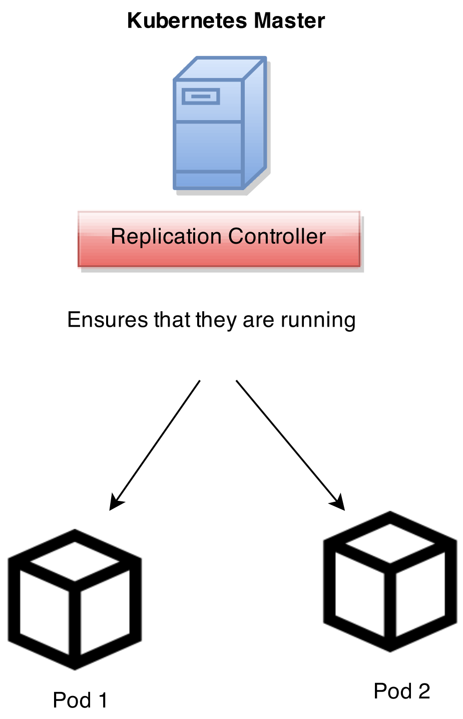


Figure 2.13: Replication controllers in Kubernetes

the name of the label of the replicationController. In order to run a container, */opt/bin/kubectl run-container apache –image=ravn/<dockerimage>* can be used.

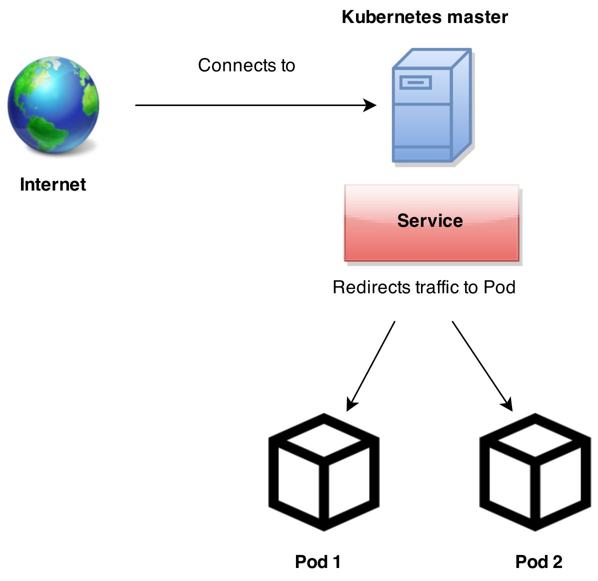


Figure 2.14: Services in Kubernetes

Part II

The project

Chapter 3

Planning the project

This section aims to explain how the experiments were designed, the reasoning behind them and how the setup was configured to perform the experiments. It will also discuss limitations and challenges of the experiments.

3.1 Approach

This study will gather quantitative data. For the experiment, several infrastructure and software solutions are required. Considering that the experiment will be run on virtual machines, there is a need for both physical resources and hypervisors. This is provided by the use of a cloud running Openstack. The Openstack cloud is being used because it has resources that are easily available to the researcher, free of cost and based on open-source software. Using the cloud, an internal network of virtual machines has been created called ravnnett. All the required virtual machines will be created and connected within this network.

The experiments will be divided into two parts. The first part will be using Kubernetes and Docker combined to launch new containers according to specifications. The seconds part will consist of launching a new virtual machine from a template using MLN according to the specifications. Each part will consist of two scenarios designed to test performance in relation to scalability. In the first scenario, a new web server will be initialized first using containers and then using templates. Using Kubernetes in the first scenario, there will be cluster with one Kubernetes master and one Kubernetes minion running. A third virtual machine, benchmarker, within the same network will then be running a script to see how long it takes after the Kubernetes instructions to launch a container is executed, until its web server responds. This will then also be tested using MLN to boot from a snapshot that has a web server installed and configured to see the differences, using the same benchmarker machine. The collected data will be the number of seconds until a HTTP reply is received. Figure 3.1 shows the setup for scenario 1 using both Kubernetes and MLN.

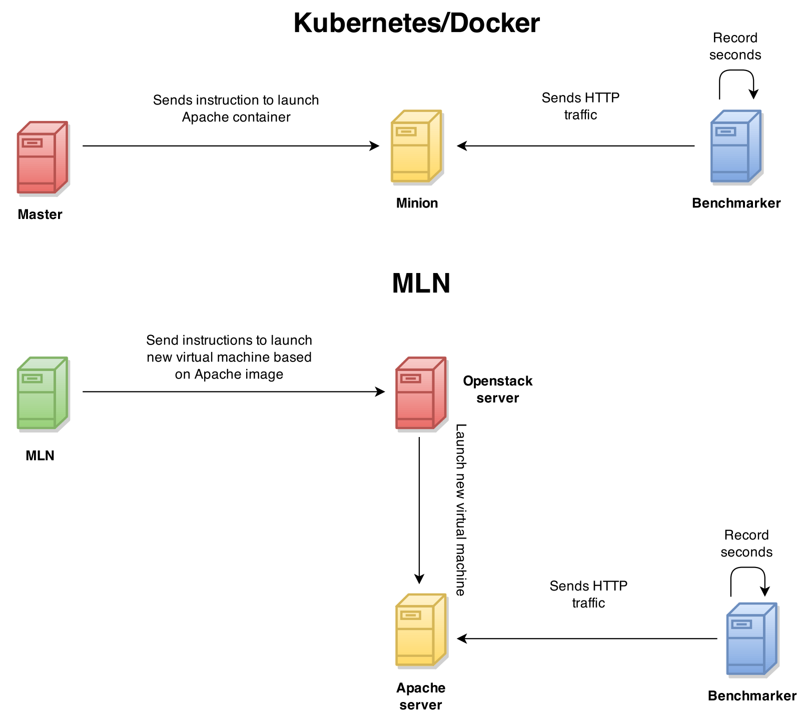


Figure 3.1: Scenario 1

In the second scenario the focus is on re-balancing the use of resources. In this scenario there is a need for more database capacity and less web servers. The web server will be shut down and replaced by a database server. A SQL request will then be sent and the amount of seconds it takes before the new server responds will be recorded. Using Kubernetes, the web server container will be shut down and a database container will be launched on the minion in its place. Using MLN, the virtual machine running the web server will be stopped, and a new database server will be created using a pre-configured snapshot.

Each scenario for both the Kubernetes and template experiments, will have a sample size of 30. 30 was chosen as the sample size because it is often said that the sample size should be larger than 30 for the statical analysis to provide a more reliable output. Any extreme results will be removed from the sample. New data points from the experiment will be put in their place. This is done to try to remove any errors, which devalues the calculated means, standard deviations and confidence intervals.

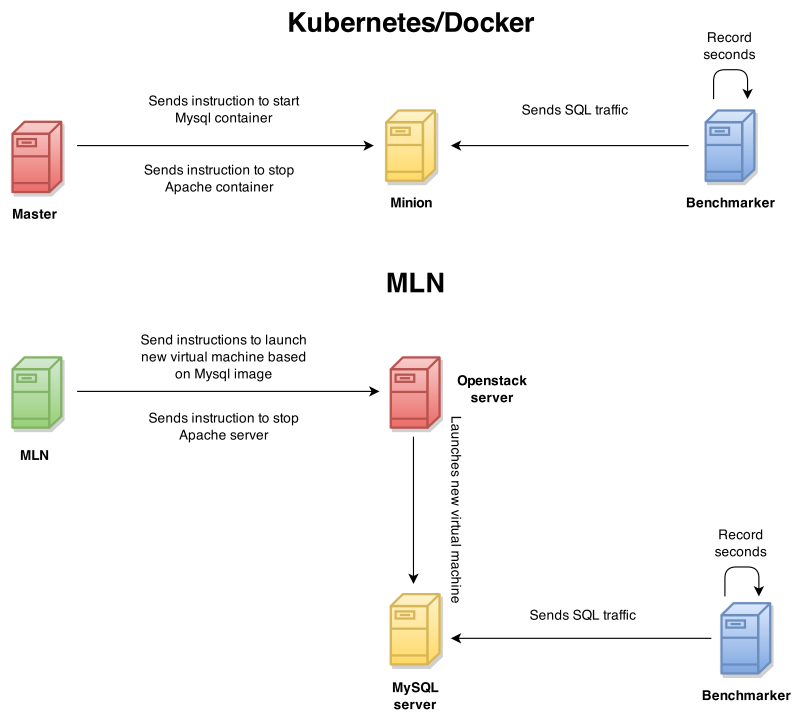

Kubernetes/Docker
Record seconds
Sends instruction to start Mysql container
Sends instruction to stop Apache container
Sends SQL traffic
Master
Minion
Benchmarker
MLN
Send instructions to launch new virtual machine based on Mysql image
Sends instruction to stop Apache server
Openstack server
Launches new virtual machine
MLN
Record seconds
Sends SQL traffic
MySQL server
Benchmarker

Figure 3.2: Scenario 2

Docker was chosen because it focuses on using containers to run applications and services, which is in line with this project focusing on web services. Docker also has a good community which provides information and guides regarding the software. The official documentation is well structured and Docker is highly popular, which is helpful when searching for help and guidance on upcoming problems. Kubernetes was chosen because it is a system that is built around using Docker containers. There are other orchestration tools for Docker, but Kubernetes is a large project supported and created by large and influential companies such as Google, which often is a sign of reliability and stability. This is important considering that Docker and orchestration tools are relatively new technologies and software. Kubernetes is also based on open-source development and its main focus is on running applications in containers which is in line with this project based on web services.

MLN was chosen because it was one of the tools that were available to use with the cloud provided by the educational institution. The experiments were originally thought to just showcase how containers could contribute to quick scalability without testing other types of technology that are presently being used. However, it became apparent later in the project that it would be more interesting to compare a technology using templates, which is one of the most used methods today, with Linux Containers. Considering that both Kubernetes, Docker and cluster management in general have a pretty steep learning curve and will be time consuming, it was necessary to use a technology the researcher had some previous experience with. MLN is also more similar to some other template technologies than writing very complex nova commands (to communicate with the Openstack server). It is also a tool that could be used on several different cloud technologies in case the cloud provided to the researcher would not work correctly, which there were some problems with initially.

These scenarios have been designed in order to test both systems in terms of how quickly they can offer scalability and specifically the ability to scale up to meet an increase in users of a web service. As outlined in the introduction, there are lots of reasons why a cloud should not be running on full capacity when it is not necessary. It is therefore assumed that there are more resources available up to a certain point, when the amount of users of a web service increases. In modern web services, the traffic loads are fluctuating and it is therefore important to be able to quickly scale up when the traffic increases [3]. These fluctuations are very difficult to plan ahead [3]. When the amount of users increases, there are mainly two methods of dealing with that problem. One way is to simply use more of the available resources, which is the purpose behind the design of scenario 1. The other method is to better re-balance the resources that are already in use and that is the purpose of the design behind scenario 2. The general principle is that there is a need to make more resources available [3].

The approach for testing the scenarios was to focus on that the web service should be up and running. It was therefore important to actually test that the required web service was running and not only relying on Kubernetes reporting on the status of the containers. The same applies to the virtual machines.

There are some limitations in regard to these experiments. These limitations are mainly caused by the fact that it is difficult to directly compare two technologies that, even though they perform similar tasks, are very different in the way they are constructed. It is therefore very difficult to design an experiment which is a 100 percent fair and where the setup is exactly the same. The focus on this experiment has therefore only been to focus on that the web services should be accessible to the user in the same manner regardless of the underlying technology. Another limitation is that the recorded time of the scenarios are likely to vary between each time the experiments are run. The reasons for this is mainly that there are several different computers and software that need to communicate with each other. For instance, when running one of the Kubernetes experiments, the speed of the script will depend on: Execution of the script, network layer between several virtual machines and containers, Kubernetes master and minion communications, Kubernetes to Docker communication, Kubernetes to iptables/firewall communication. An operating system will be running plenty of others processes than those who are the main focus of this experiment. All actions will therefore have different queue and execution times between the runs of the experiments. The data from the experiments, even though they will vary to a certain degree between the different runs, should however show whether or not there is a significant difference in time consumption between the two technologies. Hence, the purpose of the experiments is therefore more about seeing if a trend can be plotted instead of concluding that a container always uses a certain amount of seconds.

The assumption before going into these experiments is that containers, based on the literature available on the subject, should perform better in these experiments than virtual machine templates.

3.2 Configuration and setup of the environment

3.2.1 MLN

In order to install MLN, the following was done.

The first thing that had to be done, was to add the havana cloud archive to apt-get and then update apt-repository list. This is done in order to be able to download the nova tool.

```
apt-get install python-software-properties
add-apt-repository cloud-archive:havana
apt-get update
```

Then the operating system and the packet repository should be updated.

```
apt-get upgrade
apt-get dist-upgrade
reboot
```

Then the nova tool can be installed, which will be used by MLN to communicate with the OpenStack server API.

```
apt-get install python-novaclient
```

MLN is provided as a git repository, and Git therefore has to be install in order to be able to download the software. The command *git clone* was used to download the MLN git repository.

```
apt-get install git
git clone https://github.com/kybeg/mln.git
```

Next the actual MLN software can be installed:

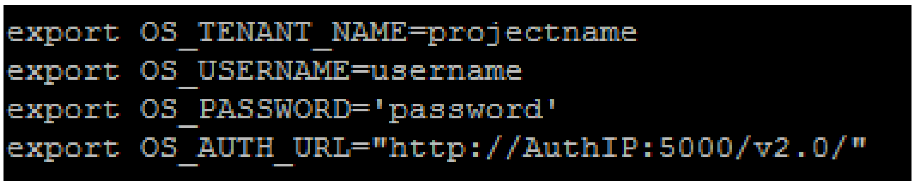
```
cd mln
./mln setup
```

During the first prompt, option 2 (Entire system), should be selected. The rest should be kept as default by just pressing enter until the prompts finishes.

In order to verify that MLN has been installed correctly, the following command can be executed:

```
mln write_config
```

In order to communicate with the Openstack Server, a .openstack file has to be created as shown in Figure 3.3.



```
export OS_TENANT_NAME=projectname
export OS_USERNAME=username
export OS_PASSWORD='password'
export OS_AUTH_URL="http://AuthIP:5000/v2.0/"
```

Figure 3.3: .openstack file

Then source has to be run to use the file as input for the nova commands.

```
source .openstack
```

In order to test that the source file contains the correct information, a nova command can be tested.

```
nova list
```

The installation process is based on the guide from the MLN guide at Github [1].

3.2.2 Kubernetes

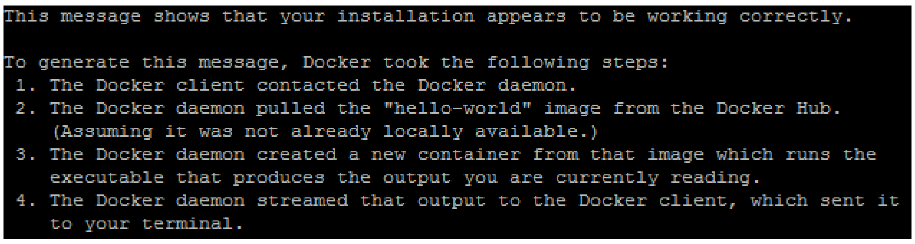
The following steps apply for both the Kubernetes master and the minion. First Docker has to be installed together with git and make by:

```
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
 --recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E9
sh -c "echo deb https://get.docker.com/ubuntu docker main >
 /etc/apt/sources.list.d/docker.list"
apt-get update
apt-get install -y lxc-docker git make
source /etc/bash_completion.d/docker
```

sudo docker run hello-world will download a test container and run it to see if Docker is installed correctly. [26] This command should present a similar output to the one shown in Figure 3.4.

Then the Kubernetes binaries has to be installed:

```
wget https://github.com/GoogleCloudPlatform/kubernetes/releases/
download/v0.12.0/kubernetes.tar.gz
```



```
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (Assuming it was not already locally available.)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

Figure 3.4: The output from the hello-world command

```
tar xfvz kubernetes.tar.gz
cd kubernetes/cluster/ubuntu-cluster/
./build.sh
mkdir /opt/bin
sudo cp ./binaries/* /opt/bin
./configure.sh
```

configure.sh will prompt the user and and request IP-addresses of the master and minions. Once finished, the minions should show in the output of */opt/bin/kubectl get minions*.

3.2.3 Benchmarking scripts

Scenario 1

In scenario 1, the purpose was to test how long it takes to launch a new web server using Kubernetes and MLN. The web servers should be accessible in a similar fashion, which means that the script can be used for testing both Kubernetes and MLN. The Python script shown below was designed for that purpose.

```
import httplib

while True:
        try:
                connection=httplib.HTTPConnection("ipaddress")
                connection.request("HEAD","/")
                status=connection.getresponse()
                statuscode=status.status

                if statuscode==200:
                        break
        except:
                pass
```

The script, while running, tries to continuously connect to the web server that is initialized. It does this by looking for the status code 200, which indicates that the web server is up and running and that everything is working correctly. When a status code 200 is received, it will break out of the infinite loop. Error handling was added with *try* and *except* to handle replies that states that the socket is closed when the container or virtual machine is being initialized. Using the function *time python scenario1.py*, it will print out a real value that calculates how long the script used to finish executing. This value will then be added to the results scenario 1. When designing this script, it was important to try to limit the amount of input/output operations that are done, because these operations will vary between different runs of the experiment. The script was therefore designed not to print out any information and just execute the necessary operations to test the web server.

Then *time python scenario1.py* was used to run the python script from the benchmarker at the same time as MLN and Kubernetes initiates a new server to see how long it takes before a response from the web server is received. The *real* output was then used to determine how long the script used to execute connecting to the http server. This process was then repeated 30 times for both technologies.

Scenario 2

In scenario 2, the purpose was to test how long it takes to stop a web server and then launch a MySQL server using Kubernetes and MLN. In order to connect to the MySQL server, the python module for communicating with MySQL has to be downloaded. This is done with *apt-get install python-mysql-connector*. The following python script was created:

```
import mysql.connector
from mysql.connector import Error


def connect():
    try:
        conn = mysql.connector.connect(host='ipaddress',
                                       database='testdb',
                                       user='username',
                                       password='password')
        if conn.is_connected():
            return True

    except Error as e:
        pass

while True:
        if connect()==True:
                break
```

The design of the script follows a lot of the same guidelines as the script for scenario 1. The script will run an infinite loop to try and connect to the MySQL server and to the database testdb. When it is able to connect, the script will finish. Error handling was added with *try* and *except* to handle replies that states that the socket is closed when the container or virtual machine is being initialized. Using the function *time python scenario2.py*, it will then print out a real value that calculates how long the script used to finish executing. This process is then repeated 30 times for both technologies. It was not necessary to test that the Apache server had been stopped, because the operation to do that is queued before the launching of the new container or virtual machine.

3.2.4 Templates and initialization scripts

The purpose of this subsection is to explain the steps taken to create the image templates for the containers and virtual machines, as well as the commands to initiate them. Even though ssh is not a service that is tested by the scripts, it is the best way to connect to the newly configured machine in Openstack, which is helpful when testing the scripts and searching for errors. Ssh is not necessary to connect to the containers, but it seemed fairer to also launch the ssh service on the containers when it was done on the virtual machines. The ssh service is queued first in order for it to launch before the apache service.

Scenario 1 : Kubernetes

The containers launched in this experiment will be running a web server, apache2 and ssh, and an image for this has to be created before containers can be launched with it. This is done by first creating a Dockerfile with instructions for the container. In order to run the Apache service inside a container, there is a need to first specify the operating systems upon which the custom Docker image should use as a base. This is done through the use of *FROM*. Then there is a need to install Apache and ssh because the image is based on a clean ubuntu installation without Apache/ssh pre-installed. This command is run through the use of *run*. The installation of Apache will then be a part of that image. Apache and ssh is kept running inside the container through the use of supervisor, which allows several processes to run inside a container. *CMD* specifies the command to be run when the container is launched. The whole Dockerfile used in this experiment is shown below.

```
FROM ubuntu:14.04
MAINTAINER ravn@example.com
RUN apt-get update && apt-get install -y openssh-server
apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd
```

```
/var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
EXPOSE 22 80
CMD ["/usr/bin/supervisord"]
```

When using supervisor to run several services inside a container, a configuration file has to be created to specify which services to run.

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars
 && exec /usr/sbin/apache2 -DFOREGROUND"
```

The custom Docker image can now be built using the Dockerfile with *sudo docker build -t <ouruser>/<nameofimage> <sourceofimage>*. The image can now be used as a basis for new containers using Kubnernetes and Docker.

Using a bash script, the container can then be launched and made available from other machines in the network using:

```
/opt/bin/kubectl run-container <podname> --image=<imagename>
/opt/bin/kubectl expose <rcname> --port=80 --container-port=80
--public-ip=192.168.127.21 --selector=run-container=apache
```

Scenario 1 : MLN

First a snapshot was made by first booting a new server with Ubuntu 14.04 and installing apache on it with *apt-get install apache2*. Creating startup scripts for the new virtual machine running Apache can then be done using *mln build -f scenario1.mln*. The MLN script will create one small virtual machine(1 CPU and 2 GB RAM) based on the snapshot created of the Ubuntu 14.04 machine with Apache running and connect it to the virtual network in the Openstack cloud.

Listing 3.1: scenario1.mln

```
global {
                project project1
}

             host machine1 {
                 openstack {
                      image Apacheimage
                      flavor m1.small
                      keypair gate
                 }
                 network eth0 {
                      net ravnnett
                      address dhcp
                  }
}
```

Then the command *mln start -p project1* will be run to start the virtual machine. For each time the experiment is done, the machine will be deleted. Then it will be started again with the *mln start -p project1* and the time spent recorded by the benchmarking server.

Scenario 2 : Kubernetes

An image with a fully configured MySQL has to be created before new containers can be launched. This is done in the same way as in scenario 1 by creating a Dockerfile with instructions for the image to be created. The image is created with *docker build -t <iamgename> <Dockerfilelocation*.

Listing 3.2: Dockerfile scenario 2

```
FROM ubuntu:14.04
MAINTAINER ravn@example.com
RUN apt-get update && apt-get install -y openssh-server mysql-server
    ↪ supervisor
RUN sed -i -e"s/^bind-address\\s*=\\s*127.0.0.1/bind-address␣=␣
    ↪ 0.0.0.0/" /etc/mysql/my.cnf
RUN /usr/sbin/mysqld & \
    sleep 10s &&\
    echo "GRANT␣ALL␣ON␣*.*␣TO␣monty@'%'␣IDENTIFIED␣BY␣'12345'␣WITH␣
        ↪ GRANT␣OPTION;␣FLUSH␣PRIVILEGES" | mysql
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
EXPOSE 22 3306
CMD ["/usr/bin/supervisord"]
```

This Dockerfile is based on the script provided by http://txt.fliglio.com/2013/11/creating-a-mysql-docker-container/ but slighly customized.

Then the instructions to start the services has to be added to *supervisord.conf*.

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D
```

```
[program:mysql-server]
command=/usr/bin/mysqld_safe
```

The custom Docker image can now be built using the Dockerfile with *sudo docker build -t <ouruser>/<nameofimage> <sourceofimage>*.

The container can then be launched using:

```
/opt/bin/kubectl stop replicationcontroller apache
/opt/bin/kubectl stop services apache
/opt/bin/kubectl run-container mysql --image=ravn/scenario2
/opt/bin/kubectl expose mysql --port=3306 --container-port=3306
--public-ip=192.168.127.21 --selector=run-container=mysql
```

Scenario 2 : MLN

A snapshot of a working mysql server has to be created so that it later can be booted from using MLN. First *apt-get install mysql-server* has to be run to install the server. Then log in to root with *mysql -uroot -p<password>*. Then a new user has to be created with *CREATE USER 'username'@'%' IDENTIFIED BY 'password';*. Then it has to be granted access with *GRANT ALL PRIVILEGES ON *.* TO 'username'@'%'*. After that a database has to be created with *CREATE DATABASE databasename*. Lastly change the *bind-address* in /etc/my.cnf to the IP of the interface. Then a snapshot of the current configurations is taken and can now be booted with a virtual machine using *mln build -f scenario2.mln*.

Listing 3.3: scenario2.mln

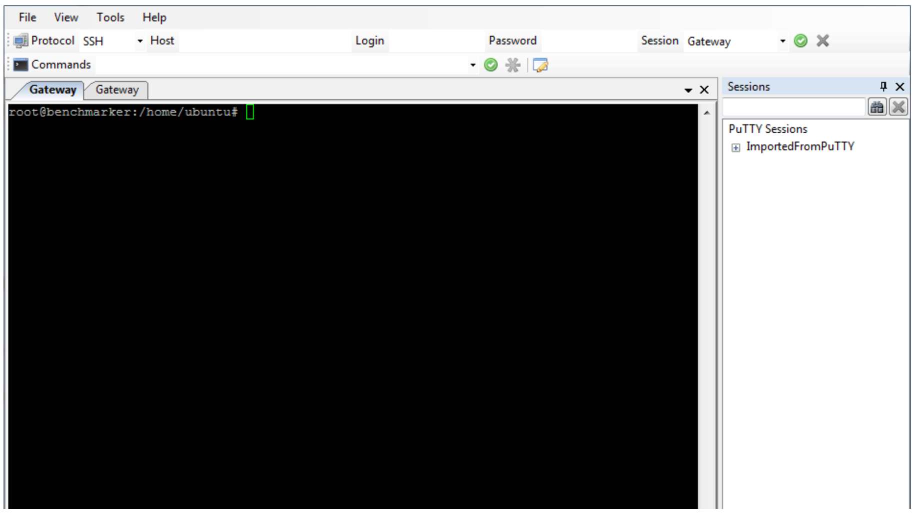
```
global {
                project project2
}

            host databaser {
                openstack {
                     image Mysqlworking
                     flavor m1.small
                     keypair gate
                }
                network eth0 {
                     net ravnnett
                     address dhcp
                }
}
```

On the MLN-machine, a script was created to stop the apache machine and start and create the mysql machine.

```
mln stop -p project1
mln start -p project2
```

3.2.5 SuperPuTTY

SuperPuTTy is a program that was used in order to ensure that both the benchmarking script and the launching of container/virtual machines starts simultaneously. It allows the user to send a command to several machines through ssh at the same time.



```
root@benchmarker:/home/ubuntu#
```

Figure 3.5: Superputty GUI

3.2.6 Versions of software

The following versions of software will be used in all the experiments:

- Kubernetes master and client version 0.12
- Docker version 1.6
- MySQL-server version 5.5
- Apache version 2.4.7

Part III

Conclusion

Chapter 4

Results

In this chapter, the results for the experiments designed and explained in the approach section, will be presented. This data will then be analyzed using different statistical methods and tools.

4.1 Results and analysis

All of the statistical analysis is done using the program R. Values for the results are shortened to just one decimal when presented in graphs or charts.

The statistical analysis and tools that will be used are sample mean, standard deviation, confidence intervals and the welsh two sample t test if the distribution of the data are verified as being normally distributed.

The sample mean is calculated by:

$\bar{x} = \frac{x_1+x_2+x_3...}{N}$

where x_n are the individual results from the sample and N is the number of repetitions of the experiment. When the data set is normally distributed, the mean becomes interesting, because a majority of the values in the experiment should be close to it (1 standard deviation away). If assuming that the results are close to a realistic representation of the experiment, the mean together with the standard deviation should give an indication of where most new data points for this experiment should be within.

The standard deviation of the sample is being calculated with the formula:
$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$

N is the number of results, x is a specific result and $\bar{x}$ is the mean of the sample. Standard deviation is a statistical values that gives insight in terms of how spread out the data points are.

In the process of analyzing whether the result data are approaching a normal distribution, the following formulas are being used:

$P(\mu - \sigma \leq X \leq \mu + \sigma) = 0.68$

$P(\mu - 2\sigma \leq X \leq \mu + 2\sigma) = 0.95$

$P(\mu - 3\sigma \leq X \leq \mu + 3\sigma) = 0.99$

If the results are close to being distributed in this manner, they will be defined as normally distributed.

The confidence interval for the tests can be calculated using:

$\bar{x} \pm Z * \sigma \div \sqrt{N}$

Confidence intervals are an indicator on the precision and consistency of the data. It gives an interval as output which, depending on the percentage of the confidence interval (0.99 will be used here), will give an interval where most sample means of the experiment should be within.

The welsh two sample t test is calculated using: $t = \frac{\bar{x}_1 - \bar{x}_2}{sqrt(\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2})}$

This test is used to see how much of a time difference there is between the results using two hypotheses. The first hypothesis is that the time difference between the two samples is not significant and that it is likely that a sample mean for Containers could be within the same interval as the MLN test. The alternative hypothesis is that there is a significant difference and that is unlikely that the two software types could have a sample mean within the interval of the other. The p value shows how likely it is that the two data sets have a sample mean within the same interval.

4.1.1 Scenario 1

In scenario 1 the experiment was about testing out how long it would take to spawn a new web server using Kubernetes together with Docker and MLN together with Openstack.

The histogram 4.1, shows the distribution of the data from scenario 1 using MLN:

The histogram 4.2, shows the distribution of the data from scenario 1 using Kubernetes:

Looking at the distributions of the results, it is necessary to further investigate which kind of distribution the results lie within. The MLN histogram seems to be approaching a normal distribution from its shape, but the Kubernetes histogram is more difficult to interpret. It should therefore be further investigated. To do that there is a need to know the standard deviation which is shown in Figure 4.3. Approximately 67 % of the values for Kubernetes are within 1 standard deviation away from the mean (interval between 16,72 and 22,8). The rest of the values are

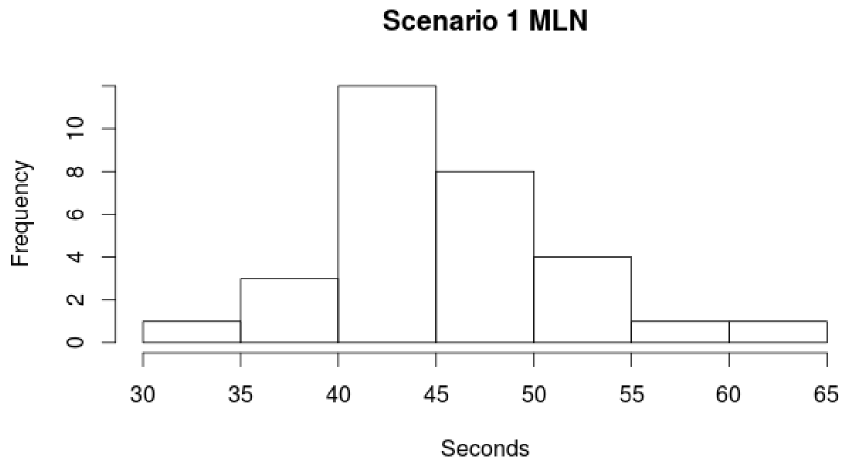


Figure 4.1: Histogram for scenario 1 using MLN

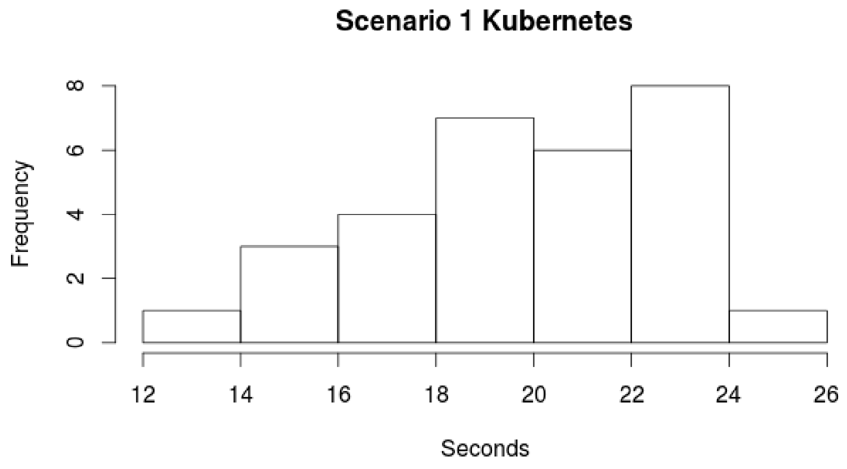


Figure 4.2: Histogram for scenario 1 using Kubernetes

within 2 standard deviations away from the mean. The Kubernetes results are therefore normally distributed because the values of sd+-1 are close to 68 % and the values of sd+-2 are close to 95 %. Regarding the MLN distribution, about 73 % of the values are within 1 standard deviation away from the mean (interval between 39,43 and 52,29). Approximately 23 % of the data points are within 2 standard deviation away from the mean (interval between 33 and 58,72). The results from the MLN experiments also points toward a normal distribution of the results.

The Kubernetes experiment has a lower standard deviation than the MLN experiment. A higher standard deviation means that there is a higher spread of the data points, which implies that the amount of seconds it takes are less consistent than with Kubernetes.

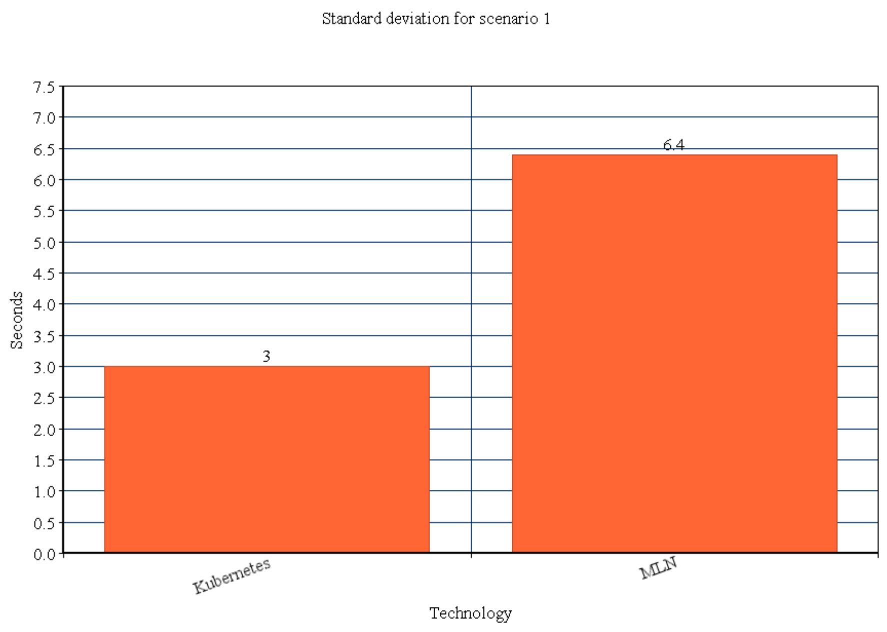


Figure 4.3: Standard deviation for scenario 1

Looking at the mean in 4.4, Kubernetes highly outperformed MLN in this experiment with the mean being more than half of the mean of the experiment using MLN. Kubernetes performed in average scenario 1 26,1 seconds faster than MLN.

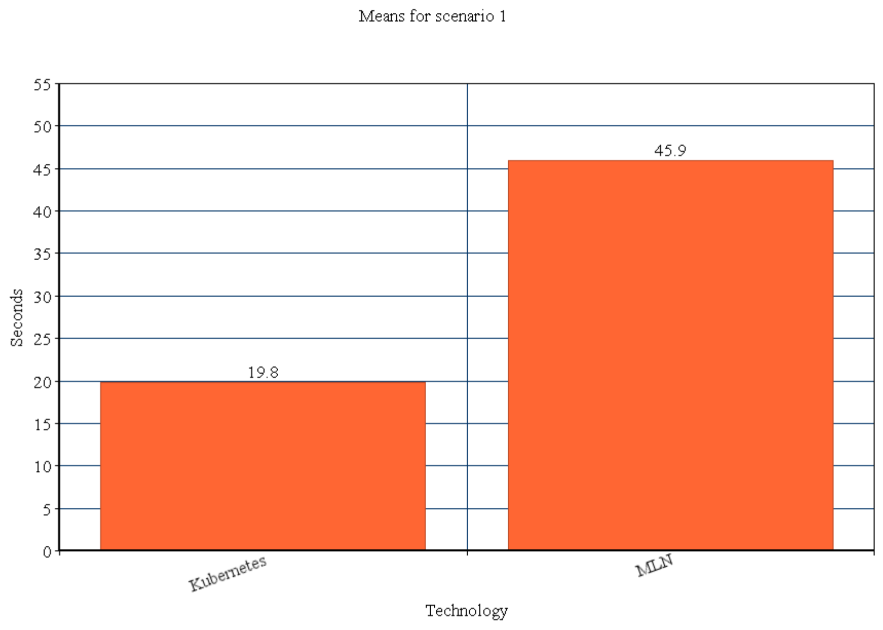


Figure 4.4: Mean for scenario 1

Analyzing the highest and lowest points of the experiment in Figure 4.5, it becomes clear that Kubernetes highest data point is 8,6 seconds faster than the lowest data point using MLN. It further enhances the impression after the comparisons of the means, that there is big difference in terms of time consumed between the two technologies.

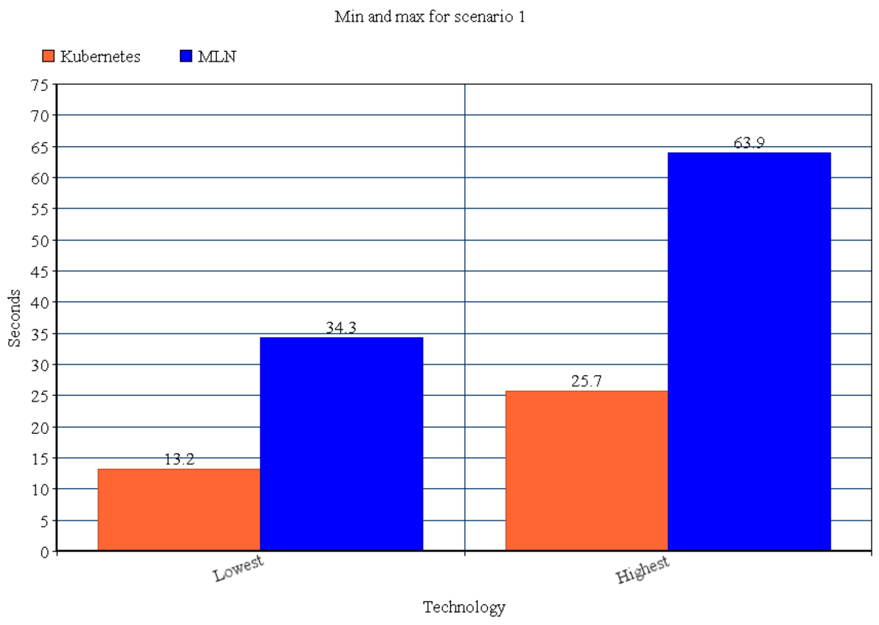


Figure 4.5: Min/max for scenario 1

Looking at the confidence intervals for scenario 1 in 4.6, it shows that Kubernetes has a lot lower values with the interval being 18.2-21.3 compared to 42.6-49.1. The interval for Kubernetes is around half that of MLN. The interval in itself is also a lot more narrow with being 3.1 seconds wide, while the interval for MLN is 6.5 seconds wide. With Kubernetes having a more narrow interval, it shows that it is more consistent than MLN.

Using the welch two sample t test to compare the results between the two experiments, a p value of 2.2e-16 is received. 2.2e-16 is an extremely low value which further underlines how different the results using MLN and Kubernetes are and the alternative hypotheses is true.

4.1.2 Scenario 2

The histogram 4.7, shows the distribution of the data from scenario 2 using MLN:

The histogram 4.8 shows the distribution of the data from scenario 2 using Kubernetes:

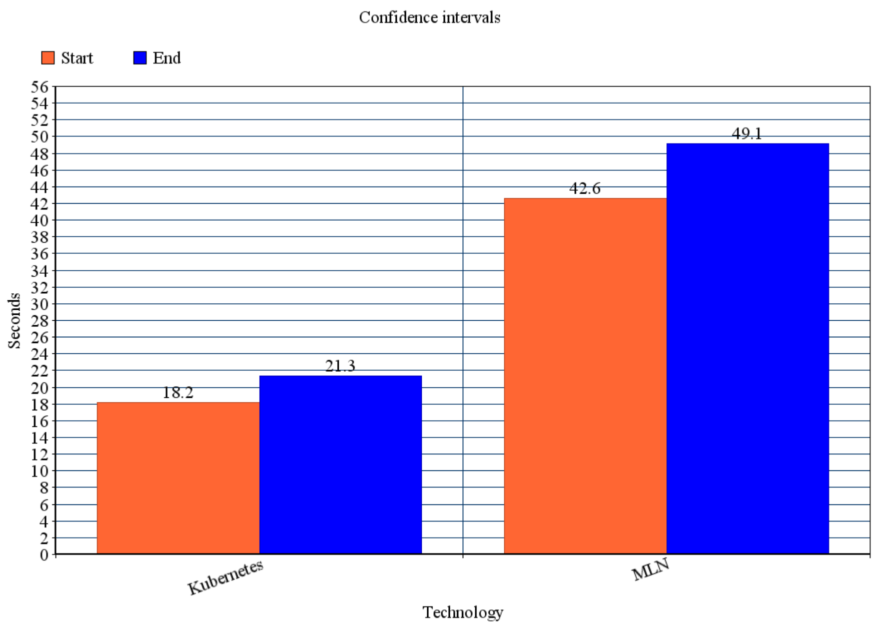


Figure 4.6: Confidence intervals for scenario 1

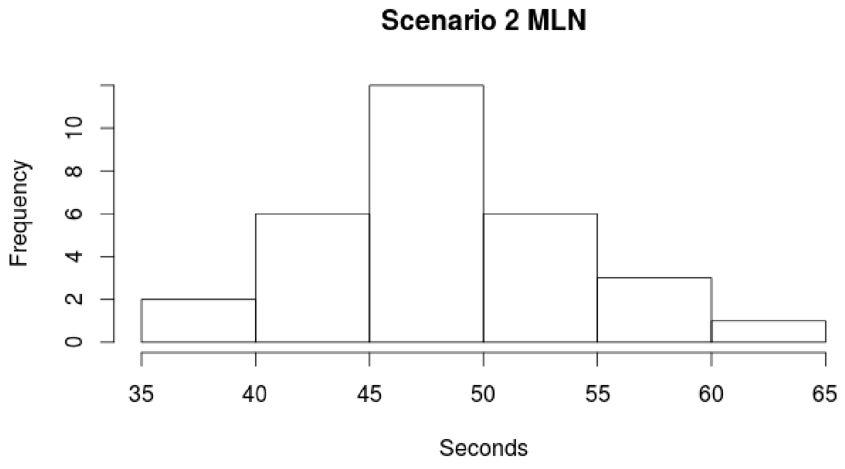


Figure 4.7: Histogram for scenario 2 using MLN

Both of the histograms shows signs of approaching a "bell curve" which often represents a normal distribution. However to further investigate what kind of distribution it is, there is a need to know the standard deviation which is shown in 4.9. Approximately 73 % of the values for Kubernetes are within 1 standard deviation away from the mean (interval between 21,41 and 27,49). Approximately 96 % of the values are within 2 standard deviations of the mean (interval between 18,37 and 30,53). When doing the same for the MLN part of the scenario, 80 % of the values are

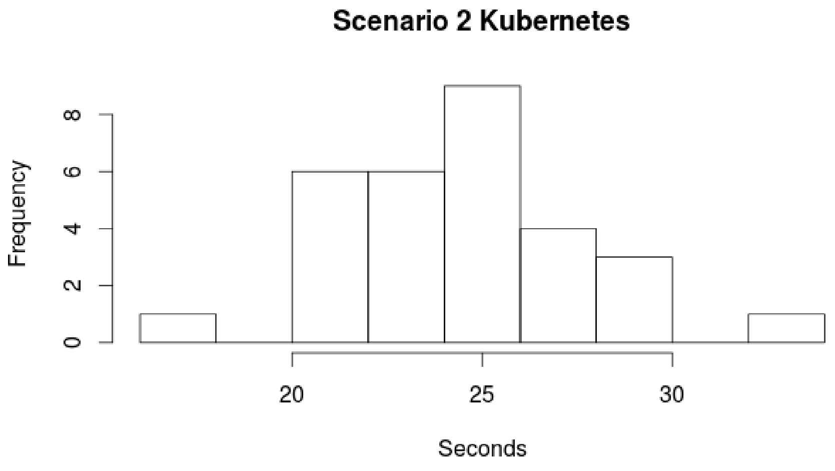


Figure 4.8: Histogram for scenario 2 using Kubernetes

within 1 standard deviation of the mean (interval between 43,2 and 54,04). Approximately 16 % is within 2 standard deviations of the mean (interval between 37,78 and 59,46). Even though 80 % within 1 sd of the mean is quite high for a normal distribution, there are only 30 data points in this experiment so the percentage will vary quite a bit. It still shows a pattern of being normally distributed. The fact that 70-80 % of the results for both Kubernetes and MLN are within 1 sd of the mean shows a certain degree of predictability in the results and it shows that there is a high chance of getting a result within +- 1 sd when using both technologies.

The Kubernetes experiment has a lower standard deviation than the MLN experiment. A higher standard deviation means that there is a higher spread of the data points, which means that the time spent is less consistent than with Kubernetes. The difference is however less than in scenario 1.

The bar diagram 4.10 shows the means for both of the technologies. Looking at the mean, Kubernetes performed considerably better than MLN with the difference being 22.1 seconds, or almost twice as many seconds.

Analyzing the lowest and highest data points of the experiments, the Figure 4.11 further amplifies the huge difference in time spent between Kubernetes and MLN. The lowest value of MLN is over twice as big as the value of Kubernetes and the highest point of Kubernetes is approximately 6 seconds less than the highest point of MLN.

Next up is the 99 % confidence intervals for scenario 2 shown in 4.12.

The confidence interval for Kubernetes is calculated to between 22.9-26 and the interval for MLN is 45.9-51.3. Kubernetes has narrower interval and provides more consistent results. MLN does however better than in scenario 1 with the interval difference being less than in the first scenario.

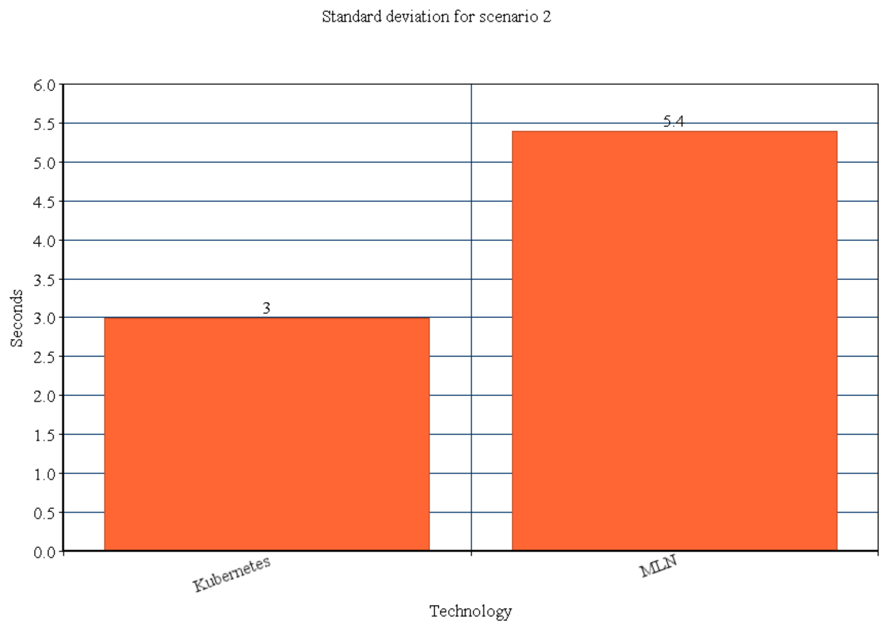


Figure 4.9: Standard deviation for scenario 2

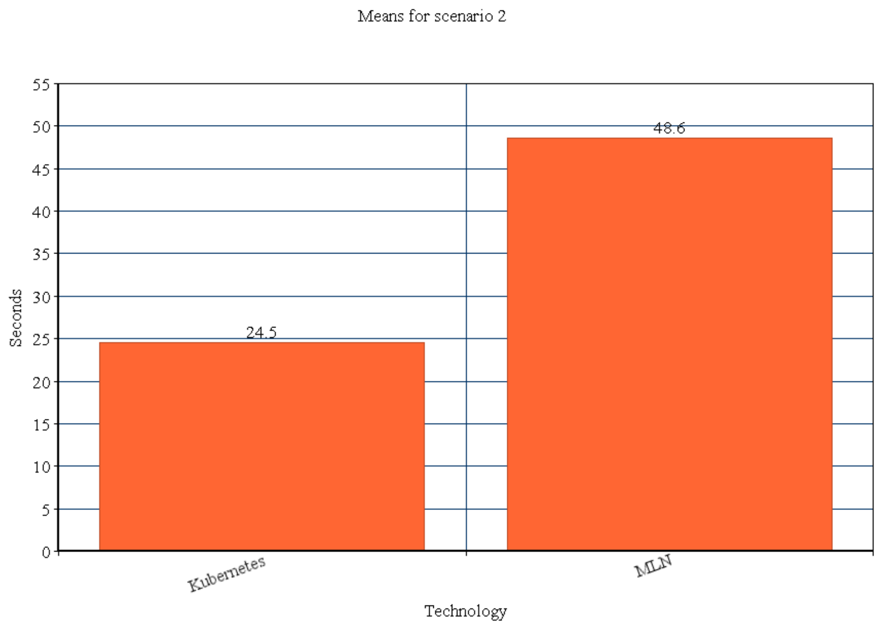


Figure 4.10: Mean for scenario 2

Using the welch two sample t test to compare the results between the two experiments, a p value of 2.2e-16 is received. 2.2e-16 is a very low value which further underlines how different the results using MLN and Kubernetes are, just like in scenario 1.

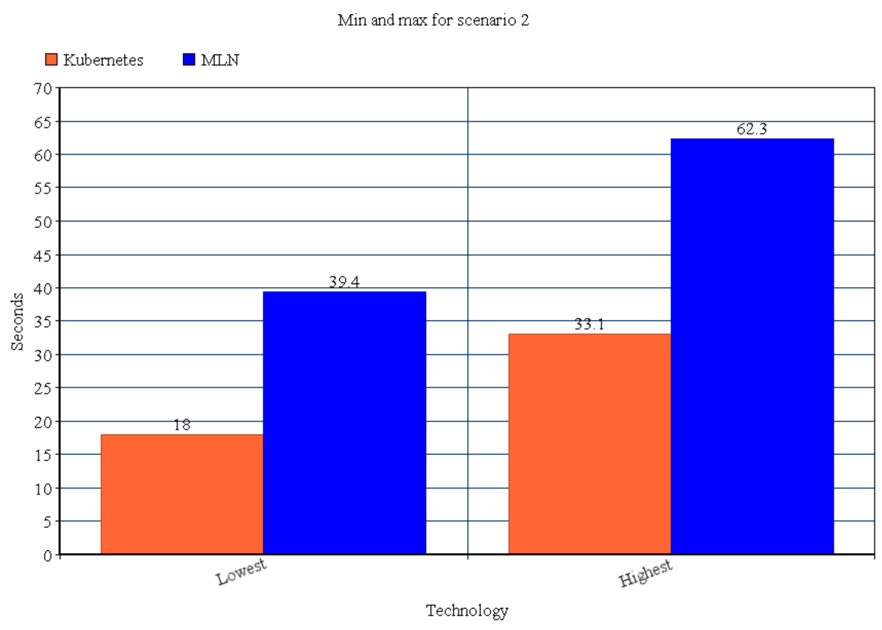


Figure 4.11: Min/max for scenario 2

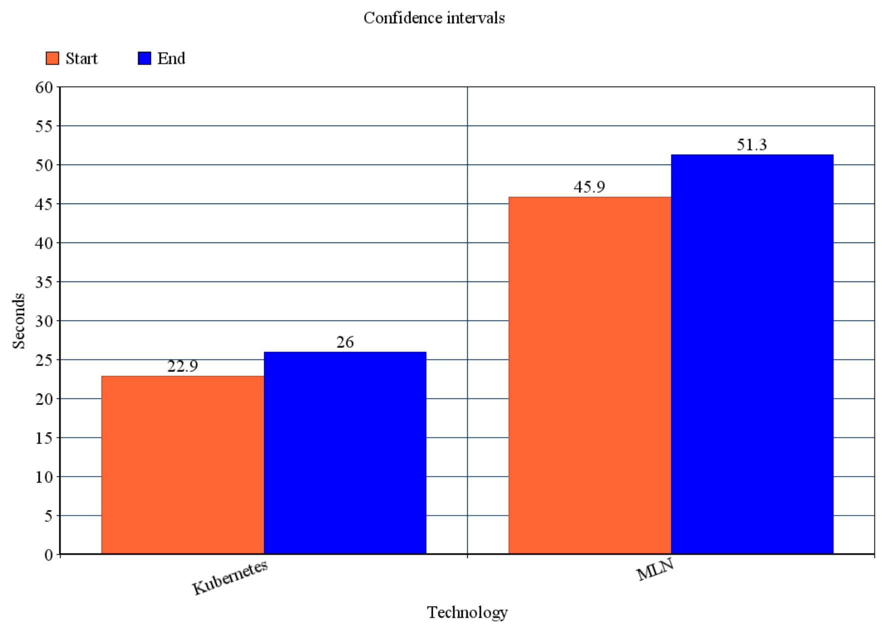


Figure 4.12: Confidence intervals for scenario 2

Chapter 5

Discussion and conclusion

This section will discuss what the output from the results and analysis represents and eventual problems with them and what future research that can be done within this topic. It will also summarize the thesis and try to answer whether the problem statement has been answered or not.

5.1 Discussion

The analysis shows that Kubernetes clearly performs better when adding more resources in scenario 1, and when the amount of resources are rebalanced between different services in scenario 2. This is in line with theory and literature on the subject that describes Linux Containers as lightweight virtual machines, which has every prerequisite to actually outperform software using templates and virtual machines. The difference in performance using Kubernetes and Docker, gives Linux Containers an edge over virtual machines when it comes to being able to quickly scale up and rebalance the use of resources for web services. Service Level Agreements will contain different requirements for a web solution from company to company, but values such as response time and uptime requirements are often present. Given the nature of web services, traffic can to a certain degree be unpredictable and fluctuating [3]. Being able to adapt quickly is therefore essential to fulfill uptime and response time requirements [3], and Kubernetes and Docker clearly performs better than using MLN and Nova. In both the scenarios, the mean using containers is almost half or less than that of MLN which is a substantial difference.

It is interesting to note that the difference between the means and standard deviations values, are less in scenario 2. The difference is however quite a lot, and the welsh t test states there is almost 0 % chance of the sample means using these two technologies that they will be within the same interval.

This thesis has been conducted as a comparative study between one type of software that uses linux containers and one that uses virtual machines and templates. The study has however also been investigative in the sense Kubernetes had to be set up in an Openstack cloud. The documentation on configuring a Kubernetes cloud for a different cloud provider than the likes of Digitiloccean, Amazon and Google, was not that easy to come by. A lot of the documentation on Kubernetes was also outdated, which meant that several scripts and commands mentioned, had been removed or replaced. This thesis, as well as showing the differences between Kubernetes and virtual machines, will hopefully also help introduce how to configure Kubernetes in a private cloud. Given the results of this study, it might persuade more web service providers to experiment using Linux containers.

In this project, containers and virtual machines have been tested in terms of how they deal with horizontal scaling. There are however other factors that also can contribute to scalability such as vertical scaling and load balancing. Virtual machines have a set cap of resources available to them, including disk space, RAM and CPU. While they will not use all those resources when it is not necessary, they cannot go above that cap. Therefore, if for instance a web server running on a virtual machine needs 100 MB more RAM than it has been assigned during a period of heavy load, a new virtual machine with new specifications has to replace that virtual machine if resources are limited in the cloud.

Another thing that may affect scalability, is that with using Kubernetes services, there is a natural load balancer already implemented in the cloud. The advantage of having to use the Kubernetes master as the load balancer is that it knows about the state of the services running in the cloud. When a new container is being launched, then Kubernetes already knows that it is present and up and running. This will however not be the case with an external load balancer that either has to be modified manually or by sending out requests to check if servers within a certain IP range are available in a certain interval.

Containers can also help with scalability, by using less resources because they share the Kernel with the host operating system, instead of each container having its own like it is the case with virtual machines. This can reduce resource consumption which in theory could provide more resources to other services, thereby increasing the amount of users the web service can handle.

One interesting thing to note from these experiments, was that there were more results that were removed from the experiment that used Kubernetes than the ones that used MLN. When referring to errors here, it means either that the container could not be launched or that a given result had a value that was too high compared to the other data points in the sample. The error rate was however pretty low and errors only occured a couple of times during the running of the experiment.

One problem with this experiment is that even though the focus is on comparing Linux containers to virtual machines in their ability to provide web services, is that it only tests Kubernetes and Docker against MLN and Openstack. According to the theory of containers, comparing other types of software should also provide a significant difference between containers and virtual machines. However at this point there are too few experiments testing these two types of technologies against each other like it has been done in this thesis. This is something that could be further explored in other research projects. There are several software solutions that are being developed and created in regards to orchestration of Docker containers. One such tool is the new orchestration tool that is being developed by Docker. This could actually have been used and explored in this project, but it had its first release near the end of February which was after the master project had begun. Further research on the topic could also be to test containers and virtual machines in large scale scenarios where there is a need to launch a lot of new containers and virtual machines to maintain the stability of a web service. Further research on Kubernetes and Docker down the line will also be interesting, because they are both relatively new types of software and they might receive updates which enhances their performance, which would provide different test results. There are also a lot of new projects that are using Linux Containers, including Core OS, Mesosphere and Project Atomic, which can be tested in terms of their ability to scale fast in future studies and research.

Kubernetes and Docker works well with other cloud management software, such as Apache Mesos. With a cloud management software like Mesos, parts of the physical computers in the cloud can be shut down when there is not a need for all the available resources. When they are turned back on, they become once again a part of the cloud. This enables cloud providers to save electricity when there is no need for the capacity. Using these kinds of software can therefore integrate well with Kubernetes and Docker, because it allows for quick downscaling while Containers can quickly scale the system back up when traffic increases again.

5.2 Conclusion

This thesis project has focused on how web services can be quickly scaled up to meet the problems that fluctuating traffic creates. The focus was to notice the difference in how software using an image template performs versus software using Linux Containers. Kubernetes and Docker was used to test Linux Containers, while MLN and Openstack was used to test image templates. This particular research problem was found important to investigate, because of the demands of modern web services [3]. Modern web services should be able to cope with differentiating amount of users, while at the same time use as little resources as possible when web traffic is lower [3].

The experiments were designed to see if Linux Containers could in any way help with the problems of uneven traffic loads, to maintain uptime and help keep the service providers within the service level agreements.

The results and analysis shows that Kubernetes and Docker clearly outperforms MLN in terms of being able to scale quickly, both when using more resources and re-balancing the resources already in use. Kubernetes does not only scale faster, its results are also more consistent. With the confidence interval stating that 99 % of means from the same experiment should be within 18 and 21 for scenario 1 and 23 and 26 for scenario 2 using Kubernetes, it should be over twice as effective when many tests are performed compared to MLN and Openstack.

The problem statement for this thesis was: Can Linux containers provide quicker scalability for demanding web services with fluctuating traffic than traditional virtual machine templates? After getting the results of the experiments, it is clear that Kubernetes will scale faster than using MLN. The problem is however that this cannot be said about Linux Containers and virtual machine template technologies in general, because only two combinations of software have been tested. The results are however in line with the theory about Linux Containers, that describes them as lightweight virtual machines with a shared kernel. So even though it can be assumed that Linux Containers will scale faster in most scenarios, it is not possible to definitely conclude that this is true in all cases.

Bibliography

[1] Kyrre Begnum. *MLN + OpenStack Havana + Ubuntu 12.04*. 2015. URL: https://github.com/kybeg/mln/blob/master/README.rst.

[2] Kyrre Begnum, Nii Apleh Lartey and Lu Xing. 'Cloud-Oriented Virtual Machine Management with MLN'. In: *Proceedings of the 1st International Conference on Cloud Computing*. CloudCom '09. Beijing, China: Springer-Verlag, 2009, pp. 266–277. ISBN: 978-3-642-10664-4. DOI: 10.1007/978-3-642-10665-1_24. URL: http://dx.doi.org/10.1007/978-3-642-10665-1_24.

[3] T.C. Chieu et al. 'Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment'. In: *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*. Oct. 2009, pp. 281–286. DOI: 10.1109/ICEBE.2009.45.

[4] Docker. *Network Configuration*. 2015. URL: http://www.ehow.com/info_12213044_disadvantages-centralized-network-scheme.html.

[5] Justing Ellingwood. *An Introduction to Kubernetes*. 2014. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.

[6] Laura Frank. *What is Google's Kubernetes and How to Use It*. 2014. URL: http://www.centurylinklabs.com/what-is-kubernetes-and-how-to-use-it/.

[7] Marisol García-Valls, Tommaso Cucinotta and Chenyang Lu. 'Challenges in real-time virtualization and predictable cloud computing'. In: *Journal of Systems Architecture* 60.9 (2014), pp. 726–740. ISSN: 1383-7621. DOI: http://dx.doi.org/10.1016/j.sysarc.2014.07.004. URL: http://www.sciencedirect.com/science/article/pii/S1383762114001015.

[8] Grig Gheorghiu. *Automated deployment systems: push vs. pull*. 2010. URL: http://agiletesting.blogspot.no/2010/03/automated-deployment-systems-push-vs.html.

[9] Md. Iqbal Hossain and Md. Iqbal Hossain. 'Dynamic scaling of a web-based application in a Cloud Architecture'. MA thesis. KTH, Radio Systems Laboratory (RS Lab), 2014, pp. xv, 105.

[10] Dong Huang, Bingsheng He and Chunyan Miao. 'A Survey of Resource Management in Multi-Tier Web Applications'. In: *Communications Surveys Tutorials, IEEE* 16.3 (Mar. 2014), pp. 1574–1590. ISSN: 1553-877X. DOI: 10.1109/SURV.2014.010814.00060.

[11] Wei Huang et al. 'A Case for High Performance Computing with Virtual Machines'. In: *Proceedings of the 20th Annual International*

Conference on Supercomputing. ICS '06. Cairns, Queensland, Australia: ACM, 2006, pp. 125–134. ISBN: 1-59593-282-8. DOI: 10.1145/1183401.1183421. URL: http://doi.acm.org/10.1145/1183401.1183421.

[12] T. Kilpi. 'Choosing a SCM-tool: a framework and evaluation'. In: *Software Engineering Environments, Eighth Conference on*. Apr. 1997, pp. 164–172. DOI: 10.1109/SEE.1997.591828.

[13] Horacio Andrés Lagar-Cavilla et al. 'SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing'. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 1–12. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519067. URL: http://doi.acm.org/10.1145/1519065.1519067.

[14] Cory Lueninghoener. 'Getting Started with Configuration Management'. In: 36.2 (Apr. 2011), ??–?? ISSN: 1044-6397. URL: https://www.usenix.org/publications/login/april-2011-volume-36-number-2/getting-started-configuration-management.

[15] Mark P. M. 'The cloud begins with coal'. In: *CoRR* abs/1411.5077 (2013). URL: http://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf.

[16] Matt McGew. *The Disadvantages of a Centralized Network Scheme*. 2015. URL: http://www.ehow.com/info_12213044_disadvantages-centralized-network-scheme.html.

[17] Dirk Merkel. *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. May 2014. URL: http://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[18] Opensource.com. *What is OpenStack?* May 2015. URL: http://opensource.com/resources/what-is-openstack.

[19] Openstack. *Heat*. 2015. URL: https://wiki.openstack.org/wiki/Heat.

[20] Puppetlabs. *Learning Puppet — Basic Agent/Master Puppet*. 2015. URL: https://docs.puppetlabs.com/learning/agent_master_basic.html.

[21] Puppetlabs. *Puppet Internals*. 2015. URL: https://docs.puppetlabs.com/guides/puppet_internals.html.

[22] Jake Sanders. *Applications at scale: Running Docker Containers on Apache Mesos*. 2015. URL: http://www.livewyer.com/blog/2015/02/04/applications-scale-running-docker-apache-mesos.

[23] *Virtualization and Containerization of Application Infrastructure: A Comparison*. Vol. 21. June 23. University of Twente, 2014.

[24] Yasir Shoaib and Olivia Das. 'Performance-oriented Cloud Provisioning: Taxonomy and Survey'. In: *CoRR* abs/1411.5077 (2014). URL: http://arxiv.org/abs/1411.5077.

[25] C. Tradowsky et al. 'Determination of on-chip temperature gradients on reconfigurable hardware'. In: *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*. Dec. 2012, pp. 1–8. DOI: 10.1109/ReConFig.2012.6416738.

[26] unknown. *Docker installation*. 2015. URL: https://docs.docker.com/installation/ubuntulinux/.

[27] W3. *Bruk av IKT i husholdningene, 2014, 2. kvartal*. 2014. URL: http://www.ssb.no/ikthus.

[28] W3. *Webservice definition*. 2015. URL: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice.

[29] Johannes Wettinger et al. 'Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA'. In: *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013,8-10 May 2013, Aachen, Germany*. SciTePress, 2013, pp. 437–446.