Exploring the effects of content scaling techniques for large-scale websites

Jostein Løvbråten
Master’s Thesis Autumn 2016
Exploring the effects of content scaling techniques for large-scale websites

Jostein Løvbråten

December 12, 2016
Abstract

Scaling in the cloud is a well researched field within the realm of information technology, however these traditional scaling methods require more resources in the form of money or computational power. Content scaling is a method which reduces the quality of the service provided, effectively allowing for more connection per server. The goal of this paper is to test how this scaling method perform and how it can be implemented within a system architecture. This method was tested by creating a framework within a virtual architecture which utilizes different strategies to manage the content scaling method. These strategies seemed to reduce the response time from 60-70 seconds to 10-15 seconds. This scaling technique can be used in conjunction with traditional scaling methods to better handle a fluctuating number of connections or it can be used by IT administrators with limited resources.
Acknowledgments

I would like to express my sincere gratitude and appreciation to:

• My supervisor Kyrre Begnum for his support and willingness to share his experience and knowledge throughout this project.

• The rest of the professors that have taken their time to help me during the whole master program.

• My fellow master students for all the help, motivation and encouragement during my time as a student.

• And least but not least my friends and family for their unconditional love, support and patience throughout this project.
## Contents

1 Introduction ........................................... 1  
  1.1 Chapter summaries ........................................ 2  

2 Background ................................................... 5  
  2.1 System architecture ............................. 5  
  2.2 Virtualized architecture and their economics .......... 5  
  2.3 Service components .............................. 6  
  2.4 User demand and satisfaction ...................... 7  
  2.5 Load balancing and scaling ....................... 7  
    2.5.1 Reactive scaling ................. 8  
    2.5.2 Proactive scaling ............ 8  
    2.5.3 Content scaling .............. 9  
    2.5.4 HAProxy ...................... 9  
  2.6 Related work ........................................ 9  
    2.6.1 Abstraction .................... 11  

3 Approach .................................................. 13  
  3.1 Objectives ......................................... 13  
  3.2 Design phase ....................................... 14  
    3.2.1 Modelling .......................... 14  
    3.2.2 pseudo code ................. 15  
    3.2.3 Requirements and limitations .... 16  
    3.2.4 Design of strategies .......... 16  
  3.3 Implementation ..................................... 16  
    3.3.1 Environment .................... 16  
    3.3.2 framework .................. 17  
    3.3.3 Data gathering ............... 17  
  3.4 Experiments ....................................... 18  
    3.4.1 Functional testing ............ 18  
    3.4.2 The exhaustive strategy ..... 18  
    3.4.3 The greedy strategy .......... 18  
    3.4.4 The panic strategy .......... 19  
  3.5 Expected results ..................................... 19  
  3.6 Measurements and Analysis ........................ 19  
  3.7 Appraising properties ............................. 20  
    3.7.1 Create a framework ............ 20  
    3.7.2 content scaling strategies .... 20
3.7.3 Quality of service ........................................... 20
3.8 Possible challenges ........................................... 21

4 Results I: Design .................................................. 23
  4.1 Model .......................................................... 23
  4.2 Requirements .................................................. 23
    4.2.1 The separated components model ................. 24
    4.2.2 The control panel model ......................... 25
    4.2.3 Introducing locks ................................... 28
    4.2.4 Final model ........................................... 29
  4.3 Strategies .................................................... 30
    4.3.1 Deciding factors ................................... 31
    4.3.2 Exhaustive strategy ................................ 32
    4.3.3 Greedy strategy .................................... 33
    4.3.4 Panic strategy ....................................... 34

5 Results II: Implementation ...................................... 37
  5.1 Implementation - Creating the environment .......... 37
    5.1.1 Loadbalancer ......................................... 37
    5.1.2 Web server ........................................... 39
    5.1.3 Image server ......................................... 40
  5.2 Crawler ......................................................... 40
    5.2.1 Program logic ....................................... 41
    5.2.2 Calculating averages ................................ 42
    5.2.3 Crawling mechanism ................................. 43
  5.3 Crawler scheduler ............................................ 44
  5.4 Manager ........................................................ 44
    5.4.1 Logical structure ................................... 45
    5.4.2 Manager initiation .................................. 46
    5.4.3 Control panel initiation ............................ 46
    5.4.4 Webserver initiation ............................... 47
    5.4.5 Thinktank .............................................. 48
    5.4.6 Exhaustive strategy ................................ 50
    5.4.7 Greedy strategy ..................................... 51
    5.4.8 Panic strategy ....................................... 52

6 Results III: Measurements and analysis ....................... 55
  6.1 Functional testing .......................................... 55
    6.1.1 Unit testing ........................................ 55
    6.1.2 Integration testing ................................ 56
    6.1.3 System testing ....................................... 57
  6.2 Strategy 1: Exhaustive approach ......................... 57
    6.2.1 Flat: 40 connections ............................... 57
  6.3 Strategy 2: Greedy approach ............................... 60
    6.3.1 Flat: 40 connections ............................... 60
    6.3.2 Experiment results ................................ 61
    6.3.3 Second trial ......................................... 62
  6.4 Strategy 3: Panic approach ................................ 63
## 6.4.1 Flat: 40 connections

## 6.4.2 Experiment results

## 6.4.3 Second trial

### 7 Discussion

- **7.1 Background and approach**
- **7.2 Design and implementation**
  - **7.2.1 Manager**
- **7.3 Crawler**
- **7.4 Measurements and analysis**
- **7.5 Effects of content scaling**
  - **7.5.1 Advantages of content scaling**
  - **7.5.2 The weaknesses of content scaling**
- **7.6 Affected fields**
  - **7.6.1 Web design**
  - **7.6.2 Security**
- **7.7 Future work**

### 8 Conclusion

### Appendices

- **A Manager.py**
- **B Exhaustive strategy**
- **C Greedy strategy**
- **D Panic strategy**
- **E Crawler-schedule.py**
- **F Crawler.py**
- **G changeConfig.py**
- **H delConfig.py**
- **I Index.php for webserver**
## List of Figures


3.1 Example of a data flow diagram ............................................. 15

4.1 Figure showing the manager operating separated service components .................................................. 25
4.2 Figure showing the manager operating with a control panel .................................................. 27
4.3 Figure showing an operation changing state .................................................. 29
4.4 Figure the final model to be implemented .................................................. 30
4.5 Visualization of scale used by the strategies .................................................. 31
4.6 An example of how an algorithm can orient itself by using the chart .................................................. 32
4.7 The exhaustive strategy will try every combination and pick the most optimal. In this example, configuration 5 is the most optimal .................................................. 33
4.8 The greedy strategy will continue to degrade the quality until the performance value is acceptable. In this example iteration 4 is accepted .................................................. 34
4.9 The panic strategy will immediately set the highest quality value possible and then continuously decrease the quality. In this example iteration 3 is accepted .................................................. 35

5.1 Figure showing how Crawler calculates the response time with 10 workers .................................................. 43
5.2 Figure showing the objects in the Manager program and how they relate .................................................. 45
5.3 Figure showing the tree created when Manager is mapping out all possible configurations. This example have three different format configurations and three different image quality configurations .................................................. 49
5.4 Figure showing that the configuration pointer changes configuration by the index of the list .................................................. 52

6.1 Chart comparing all of the results from the flat exhaustive experiment with 40 connections .................................................. 58
6.2 Spaghetti plot comparing each trial from the exhaustive experiments with 40 connections .......................... 59
6.3 Figure comparing all of the results from the flat exhaustive experiments with 40 connections .......................... 59
6.4 Figure comparing all of the results from the flat exhaustive experiment with 40 connections .......................... 61
6.5 Spaghetti plot comparing each trial from the greedy experiments with 40 connections .......................... 62
6.6 Figure comparing all of the results from the flat exhaustive experiment with 40 connections .......................... 63
6.7 Spaghetti plot comparing each trial from the panic experiments with 40 connections .......................... 65
6.8 Figure comparing all of the results from the flat exhaustive experiments with 40 connections .......................... 66
7.1 Figure showing the concept of using daemons to present the system metrics to the Manager. .......................... 72
List of Tables

5.1 Table showing the order of how the configurations are tested 50

6.1 Table showing the Manager decisions during the first trial
  with the exhaustive strategy .............................. 60
6.2 Table showing the Manager decisions during the second trial
  with the greedy strategy ................................. 63
6.3 Table showing the Manager decisions during the second trial
  with the panic strategy ................................. 67
Chapter 1

Introduction

In the modern IT-climate, large scaling infrastructures serving complex services are a common occurrence. Such services can do anything from serving dynamic and rapidly changing websites to video-streaming services. What these services have in common is the fact that they all rely on several internal mechanisms to function correctly. If one of these mechanisms get overloaded or malfunctions, the service as a whole will falter or stop completely. A common way to combat this issue is to move this service into a virtualized environment, more commonly known as "the cloud". This allows for the administrators to duplicate servers to ensure availability and tolerance to spikes in traffic.

Though such a structure offer a lot of possibilities, creating a secure and fault-tolerant virtual site is a daunting task with its own set of specific problems. Some of these problems are backup, recovery, capacity planning, and VM sprawl among several others[23]. Still the one problem that first pops into mind when talking about virtualization is scaling.

Scaling is a method where the virtualized site responds to the amount of traffic the site is receiving at any given time. An effective implementation of a scaling method should not only handle a large constant stream of traffic, but also be able to tolerate large fluctuations of traffic, especially in cases where the site serves large web-based services.

There have been conducted research in this field before, but with the focus from a component perspective. This is scaling where machines are added or removed to achieve a higher acceptance of requests. This is a proven concept, though is can introduce its own set of problems, and often makes the site more complex. This paper will look more into changing the configurations of the machines them-self to ensure a satisfactory service. An example of such a configuration would be to reduce the image resolution or to reduce the video quality of a streaming service.

Since there have been not conducted any research about content-scaling specifically, the effectiveness of such a method is unknown. Content-
scaling will not solve all the shortcomings of scaling, but it will hopefully serve as an intermediary solution while for example other machines are primed and made ready for production. This means that the administrators does not need to have web server immediately available, thus having more freely available machines to the sites disposal.

Content scaling will also be a cheaper alternative to more traditional scaling methods due to its very nature. Reducing the quality of the service will not have any immediate costs, since no more resources needs to be allocated. This means that content scaling can used to remedy load problems when the resources for other scaling methods are not available.

**Problem statement** How is it possible to create a framework which utilizes different content scaling strategies to ensure quality of service in spite of fluctuating user demand?

A framework is in this problem statement describes a system which will be created in this thesis. The purpose of this framework is to create a system which can simulate a site in production. This framework will also provide the basic tools for the experiments that will be conducted.

Content scaling strategies will be the different approaches to mitigate the effects of large spikes in traffic which will be simulated in the framework. Different strategies will have different strengths and weaknesses depending on the current state of the framework.

Quality of Service is a term commonly used throughout IT. Quality of service means that metrics such as error-rates, bandwidth and availability can be measured, and to some extent guaranteed in advance.

Fluctuating user demands does in this context describe a number of requests towards the framework which will vary in type and intensity. This is to better imitate real world scenarios where traffic rarely are constant and are usually very unpredictable.

### 1.1 Chapter summaries

This section will serve as a brief summary of each chapter in this paper.

**Implementation**

The background chapter introduces the reader to the technologies and principals used throughout this project. This chapter does also explain what content scaling is and how it differs from more traditional scaling methods. This chapter does also cover prior research and how content scaling was applied during 9/11 where many news sites where overrun with requests.
Approach
The approach chapter covers how this project will try find an answer to the problem statement. This is done by creating a framework and three different strategies. One exhaustive, one which gradually reduces the quality of the content and one which immediately sets the quality as low as possible and then gradually increases the quality of the content.

Design
The design chapter presents the logic of the framework. This chapter focuses on how it is possible present computer metrics in an abstract way so it will be easier to develop different strategies. This is solved by splitting logic of the program into two parts: One part called the Manager which is responsible for carrying out the strategy and making decisions. And the control panel which is responsible for interpreting the decisions into actual computer commands. This chapter does also present the logic behind the three strategies mentioned in the approach chapter.

Implementation
The implementation chapter covers the process of converting the design into a functioning framework. The chapter presents each component of the framework and focuses on how certain aspects of the design where implemented.

Measurements and analysis
Measurements and analysis presents the results of the different strategies. The results show that content scaling do affect the overall response time of the site.

Discussion
The discussion chapter reflects upon the findings during this project and how content scaling will affect the IT industry. The strengths and weaknesses of content scaling as also discussed. The chapter does also discuss the tool used to test the framework and its shortcomings.

Conclusion
The chapter summarizes the project as a whole and concludes that content scaling has potential, but that requires more research to be able to determine the true potential of the method.
Chapter 2

Background

The purpose of this chapter is to introduce the reader to concepts and challenges in the field of system architecture design and management in order to better understand the framework proposed in this paper. This chapter will also present the concept of load balancing, virtualization and other relevant technologies utilized by the framework.

2.1 System architecture

System architecture can be described as a conceptual model of a system that defines the structure, behavior, restrictions and compatibility with other systems[10]. In this paper, the word system refers to an IT system architecture. An IT system architecture, is a system of several mechanisms with an end goal of providing one or more services[16]. It is important to not confuse system architecture with the term software architecture. Software architecture is more focused on the software itself and the particular service delivered by that software. While system architecture look beyond the software itself and takes the whole infrastructure into account[12]. This means in addition to the software, the system architecture also reviews the servers, hardware, location etc.

2.2 Virtualized architecture and their economics

With the possibilities of current hardware, it is common to run several separated operative systems on the same set of hardware[4]. This is accomplished by emulating hardware for each of the operative systems on the physical machine[1]. One such emulated machine is known as a virtualized machine(VM)[4]. When several VMs are connected together through emulated network connections, it will be known in this paper as a virtualized architecture.

When working with a virtualized architecture there are many variables to consider. The most evident is the amount of hardware resources available to the administrators disposal. If the system runs on a cloud owned by
the company itself, the limitation is just simply the hardware boundaries. Though this kind of setup is rare since the cost of an effective virtualization computer is expensive because of the hardware resource requirements[21].

A more common way of running a virtualized architecture is to rent available space from a cloud provider[11]. Instead of resorting to invest in hardware, the company can opt to rent computing power from a third party. This removes the concern for hardware limits, but it creates another: The administrator now has to take the hourly cost of the network into consideration. If too many machines are in production at the same time, the cost of simply running the network can be too high. One way to ensure that the cost of running the site is at a minimum, is to only create more machines when required. This is known as scaling.

No matter the service provided, there will always be a time of day that is known as peak time. Peak time in the field of information technologies is the time interval within a 24 hour period where the amount of traffic is at its highest[6]. It is important for an IT administrator to ensure that the site can withstand the amount of traffic during this time, since this is usually the time the service provided are generating the highest amount of revenue. Though during times with little traffic, the infrastructure created to handle peak time will be far exceeding the requirement for just idly running the site. This is not cost efficient since most providers charge for the number of machines running and its specifications, not by actual cycles. Scaling solves this problem. An architecture that utilizes scaling adapts to the amounts of traffic the network is currently experiencing. This allows for a minimum amount of virtual instances running during off peak and creating more VMs when needed. Though this is much more cost efficient it is important to consider the amount of time it takes to prime a VM for production. If the site suddenly experiences a major spike in traffic, the site may struggle to keep up due to the time it takes to acclimatize the architecture. This will be discussed in greater detail in 2.5 Loadbalancing and scaling.

### 2.3 Service components

An architecture consists of several service component which provide a certain service[3]. This service is not necessarily one that is provided to a consumer, it can also be a mechanism required by the site to work properly. An example of the latter is the loadbalancer. A loadbalancer is responsible for dividing the traffic evenly among servers, assuring that no machine is idle or overworked if preventable. As mentioned, this service is not notable to consumers. Though if it where to malfunction or completely halt, users would experience a severely reduced service if any service at all.

Each service component consists of a subset of servers which provides the same service to the network. It is easier to work with service components than actual servers when administrating a network. The reasoning behind this is that the amount of actual machines will constantly change, but the service components will always be present in the network.
An example of a service component would be the web service. This service provides user access to the web page served by the site. There are several web servers on the site, but since all of the servers provide the HTTP service it is considered to be one service component.

No matter how many web servers that exists on the site, they will all fall under the same service component as long as they serve the same function. A database server does not serve the same function as a web server and falls under another service component.

All of the different service components are a part of the complete architecture.

2.4 User demand and satisfaction

In the field of program testing there is a set of principles known simply as "the 7 principles of testing"[17]. One if these principles are the "Absence – of – errors fallacy". Which reads:

"If the system built is unusable and does not fulfill the user’s needs and expectations then finding and fixing defects does not help."

This principle is also true for system administrators. It does not matter how well the site runs or how low the response time is, if the service provided does not satisfy the users. Factors like usability and other quality aspects like image resolution will affect the conversion rates. Lower conversion rates will again affect the overall revenue.

Imagine the following scenario: An image hosting site is getting user complaints about the size limitations of the images that can be uploaded to the site. In fear of losing users to competing sites, it is decided to increase the limit size, even though this will affect the site’s performance.

2.5 Load balancing and scaling

Load balancing is a technology utilized in virtualized environments to scale the size of the infrastructure based on the amount of traffic. This allows for having a minimum amount of VMs in production during off peak when
the traffic is low. When the amount of traffic starts to regenerate, the number of servers can be increased accordingly. This technology works by having images of readily available machines, which can be activated and primed for production in a matter of minutes. The principle is pretty simple: If the load is too high, create a new machine to alleviate the load on the other machines. The challenge lies within the implementation of the scaling. Making an algorithm that correctly creates and removes machines is vital for a site’s success. There are mainly two approaches to create such an algorithm.

### 2.5.1 Reactive scaling

One approach is to monitor key factors in the network and react to thresholds set by the administrator. These factors could be anything that will signify the amount of traffic or server load. For example CPU load, number of connections, response time, etc. When the factors are known, the administrator has to decide at what thresholds the automated scaling system will take action. One of the benefits with rule-based scaling is the ease of implementation and predictability of the site. The administrator will always know what state the site is in by looking at the factors. One of the major drawbacks is that the system is reactive and not proactive. When the system has to wait for a threshold to be reached, it will risk of lagging behind the actual traffic. Especially during steep increases. In these situations, the site needs to catch up with the amount of users instead of the other way around, which is undesirable.

### 2.5.2 Proactive scaling

This problem can partly be solved by proactive scaling. Proactive scaling is a scaling method where, instead of reacting to breached thresholds, the site will proactively predict the traffic and try to set up the required resources beforehand. If this technology is implemented correctly, the consumers of the service will not be negatively affected by the scaling of the infrastructure.

A challenge of using this method is the amount of preparation and data gathering needed to effectively detect traffic. A solution to this problem is to use the rule based approach first and then phase over to auto-scaling. There are however one problem that still exists. Sudden unexpected spikes in traffic will still force the proactive scaling to fall back to a form of reactive scaling and thus create a delay in the infrastructure. This might not at first seem like a major issue, since these spikes usually are rare. Though these incidents are rare, it is often critical to keep the service at a satisfactory level at these spikes, since this can potentially grant more users. If the service fails to deliver in this critical moment, the users might perceive the service to be unsatisfactory, thus creating low conversion rate for the site.
2.5.3 Content scaling

Content scaling is a method which possibly can compensate for the deficiencies of the former scaling techniques. Instead of only focusing on the number of actual servers, content scaling is a method that adjusts the service provided by a server. By degrading the service, the server is now able to serve more users. An example could be to limit the video resolution on a video site to allow for more users. This can be used as an intermediary measure to other scaling techniques to ensure that users will still have access to the site while new servers are being created.

2.5.4 HAProxy

HAProxy is an open source software offering high availability, loadbalancing and proxy services[8]. HAProxy are designed for TCP and HTTP connections and it is easily installed on a Linux system, making it ideal for this project. HAProxy is also popular amongst big IT companies such as GitHub, Imgur, Instagram and Reddit[9]. Using software which are commonly found in the IT industry will makes the framework more accurately simulate an actual service provider.

2.6 Related work

The goal of this project is to attempt to test a method observed by major web sites during the attacks on the empire state building the 9.September 2011. The web sites where struggling to serve all the traffic that was visiting the site and users where advised to instead use the radio or television for more information[20]. This was in a time where cloud technology was only just starting to hit the market[18] so the system administrators could not just create new web servers. They had to think smart, and that they did.

They knew that it was the 9/11 related content that was the cause of the majority of the traffic, meaning that any changes done to those particular pages would probably affect the site performance as a whole. How the administrators went about this issue of minimizing the site content differ from site to site. Some papers like France’s Le Monde used an extremely low quality image and no text formatting[20]. The New york times went about the issue by instead reducing the number of images on the site in the effort in keeping the site up and running.
These actions did possibly keep the sites running, but this aspect of 9/11 is poorly documented. It has never been seen as anything more than a curious "last ditch effort" done by desperate IT administrators, and thus has not ever been researched further. This also happened at the verge of the age of cloud computing. Amazon Web Services launched the year after[18], which possibly took the focus away from other methods of scaling.

All though there was not a lot of research that could be directly connected to this project, there have however been conducted much research on virtual scaling as a field. The most related scientific article about content scaling is the paper "Adaptive, Model-driven Autoscaling for Cloud Applications"[5]. This paper proposes a new cloud service named Dependable Compute Cloud (DC2), which is a service that automatically scales the virtual infrastructure based on user demand. This service achieves this by implementing Kalman filtering, which is used to automatically learn the system parameters of each application used in the network, so that the site can be scaled more optimally than regular rule based scaling. This method is also proactive instead of reactive, making it more effective as long as the traffic is somewhat predictable.

This project is not based on this paper, though it has build upon the general philosophy of it. Trying to address the deficiencies of rule based scaling without creating more work for an administrator. Another similarity between the paper an this project is the focus on the applications and not just the VMs. Though this is where the similarities end. Where the DC2 anal-
yses the applications to best scale the virtual environment, content scaling revolves around configuring the applications themselves in an effort to not scale the virtual environment.

2.6.1 Abstraction

Discussing complex objects in IT such as a virtualized infrastructures without using a certain level of abstraction can make core principles harder to understand. Using abstraction to make a set of different components with a wide range of properties into a single abstract object makes it easier for the human mind to understand the core principles that is being conveyed.

Though if every paper used their own set of abstractions or descriptions it would eventually be hard for a reader to orient them self in the slew of different terms. This is why it is important to look into related work to find reusable terms. In the case of this project, the term service component, used in the paper "Service component reusability in cloud architecture: A graph based approach"[2], is an abstraction for a one or more components that deliver the same type of service.
Chapter 3

Approach

This chapter will further look into the problem statement proposed at the introduction. **How is it possible to create a framework which utilizes different content scaling strategies to ensure quality of service in spite of fluctuating user demand?** This problem statement highlights the possibility to expand the current scaling strategies to one that does not require to change the VMs themselves, but rather the content they provide. The approach to research the possibility of this will consist of these main objectives.

- Design of model
- Design of strategies
- Implementation of the model
- Experiments
- Measurements and analysis
- Expected results

3.1 Objectives

The project will consist of three main objectives. The design phase, the implementation phase and finally measurement and Analysis. Each objective will be described in more detail in each own designated chapters. This list will only grant the reader a quick overview of the general idea of what each stage is supposed to achieve.

1. Design of framework
   
   (a) Create a model of the framework designed for a virtualized environment
   (b) Improve the model to encompass all the scenarios and experiments intended to be tested.
   (c) Identifying tools, prerequisites and features of the framework
Identifying strengths and possible weaknesses of the chosen model

2. Implementation

(a) Configure the virtual network
(b) Create the different web pages to be used in content scaling
(c) Implement the program according to the model made at the design phase
(d) Implement the strategies for the Manager

3. Experiments

(a) Conduct functional testing
(b) Test the exhaustive strategy
(c) Test the greedy strategy
(d) Test the panic strategy

4. Measurements and Analysis

3.2 Design phase

Before starting the implementation of the framework, it is important to decide the key features of the project. The design phase is the part of the project where possible solutions and challenges are explored. After a successful design phase, the implementation can proceed with the assurance that no major issues will halt the project due to poor planning or incompatible technologies.

3.2.1 Modelling

The framework created in this project will consist of several different entities. There are the main artificial intelligence, the service components and all the actual machines that these service components represent. Using modelling to graphically visualise the framework will grant a better understanding of the system as a whole.

There are many different modelling languages used to represent a system. This project will use Data Flow Diagram (DFD)[7]. These diagrams are used to represent the flow of data through a system, which is essentially what the framework is. Receiving data to validate the necessity to take action, and sending data to execute said action.
When creating diagrams that represents the framework, problems with the current design will become more evident. This could be that there are some required mechanisms that is not feasible with the current design. Another issue might be that the complexity of the model is so great, that the design needs to be streamlined. This is the reason that the design chapter will contain several models, with the one deemed most suited at the end of the chapter.

### 3.2.2 Pseudo code

Another common way of representing the design of the framework is pseudo code. Pseudo code is a non-graphical representation of the mechanisms within a system. The text is structured as code, but using common English words instead of code syntax in an effort to make the code more readable. Pseudo code is often used as a high level representation of a particular mechanism within the program. This is helpful when highlighting the implementation or the underlying logic of the program.

```plaintext
1 Number of servers = 0
2 IF Number of servers = 0:
3   Notify administrator
4 ELSE:
5   Number of servers = Number of servers - 1
```

A combination of DFD and pseudo code will be used throughout the design chapter. Models will be used to grant a understanding of the framework as a whole. Pseudo code might be used to highlight some particular mechanism that is taken into consideration when designing the framework. Though pseudo code will probably be mostly used in the implementation chapter.
3.2.3 Requirements and limitations
Defining the requirements and limitations of what the framework will be capable of will be important to ensure that time will not be spent on unnecessary functions on the framework. Mapping out these are also important to ensure that every phase will not step out of the scope of the project. These requirements will be mentioned throughout the design chapter, but will not be summarized in any fashion. The reasoning behind this is that requirements and limitations is not information that is either vital or necessarily interesting for the reader.

3.2.4 Design of strategies
The framework will have different strategies that approach content scaling in different ways. This will hopefully grant more insight in the effectiveness of content scaling. These strategies will also prove that the framework is versatile enough to utilize different strategies for content scaling. Each of these strategies will be an algorithm that will do the following:

- Find the optimal configuration
- Find an acceptable configuration by starting from the highest quality settings and then iteratively degrade the quality of the service
- Find an acceptable configuration by starting from the lowest quality settings and then iteratively upgrade the quality.

These strategies will be explained in greater detail in the design chapter.

3.3 Implementation
The implementation phase is the part of the project where the concepts of the design phase is implemented into the actual framework. The implementation phase is not only the coding of the program, but also the configuration of the virtualized network and the different software used by it.

3.3.1 Environment
The entire problem statement centers around content scaling strategies. This means that the framework will be run in a virtualised environment. In addition to being the kind of environment that is commonly associated with scaling strategies, it is also more predictable to a certain degree. Conducting experiments in physical networks have the risk of being affected by random occurrences that may pollute the data. This could be anything from vendor peculiarities to other traffic affecting the site. Note that there is still a risk of random occurrences, but these will hardly be enough to fatally affect the framework or the data collected from it.
There are several different virtualization solutions that offer the functionality of running virtualized networks. The more known of these are the Amazon Elastic Compute Cloud (Amazon EC2), Microsoft azure and Google Cloud Platform[1]. These platforms runs on a business plan known as platform as a service. Meaning that a user will pay for a virtual space which will contain the network. An alternative to these services is to use an open source solution instead, which is the case for this project.

This framework will be hosted in Openstack. Openstack is an open source cloud computing software installed and hosted by Oslo and Akershus University College. This will provide the project with enough resources to create and test the framework without having to take costs into the equation.

Using Openstack instead of a commercial solution, makes implementation less constricted since budget issues will not be a concern. It will also be easier to track down defects between the framework and the platform, since administrators at HIOA have direct access to Openstack. One of the drawbacks of using Openstack is that it is not exactly similar to the commercial solutions, which this project is aimed at. Though this environment will be sufficient to answer the problem statement.

### 3.3.2 Framework

The framework consists of two major parts. The first is the VMs themselves. The second is program performing the content scaling and having the responsibility of managing the virtual network. The VMs will be set up and configured first. Configuring the instances with the necessary software and computing power. Creating images and other miscellaneous preparations is also considered to be a part of this step in the implementation phase.

The last part is implementing the program. The program will be written in Python 2.7 with the necessary libraries which will further explained in the implementation chapter. The code will be written in an object oriented manner to make the code easier to write and understand.

### 3.3.3 Data gathering

It is important to have in mind that data needs to be extracted from this framework. Failing to take this into account may make the gathering more difficult and in worst case impossible. The data will gathered directly from the VMs by the program. The framework will maintain logs of the data extracted continuously on its own local VM.

A testing tool which are able to generate traffic is also needed. This tool must be able to not only generate the traffic, but also be able to measure the response time. The response time will then be used in addition to the data extracted from the framework, to show how the response time is affected by the difference in content.
3.4 Experiments

The experiments will be divided into four separate categories. Functional testing and one for each strategy. These categories are ranked from the most rudimentary to the most complex. The strategies starts at the more rudimentary exhaustive approach and becomes more reactive.

3.4.1 Functional testing

Before conducting any of the actual experiments, the framework will be tested to ensure that the program functions and that the service components responds correctly. These experiments will be rudimentary in nature like connection checks and VM creation and deletion.

- Check connectivity between the service components and the VM containing the AI
- Check that the service provided by the service components are after expectations
- Create and delete VMs.

3.4.2 The exhaustive strategy

The second stage of testing will be to introduce the first strategy to the framework. This particular strategy try to find the most optimal combination of content in a case where the amount of connections are causing a high response time. The strategy will then implement this combination as the result. The goals of testing the optimal strategy is to find out:

- If different content combinations do affect the response time
- The time it takes to find the optimal combination
- The reliability of this strategy
- How predictable this test is

3.4.3 The greedy strategy

The third stage will be the testing of the local optimum highest quality first, known in this paper as the greedy strategy. The greedy strategy will not necessarily be able to find the most optimal combination, but it will continue to scale down the content until the response time of the site is considered acceptable. When the strategy has found the acceptable combination, it will periodically try to scale the content up. This means that the greedy strategy will run continuously as a service instead of stopping after a solution has been found. The goals of testing the greedy strategy is to find out:
• If the strategy manages to find an acceptable combination
• The reliability of this strategy
• How predictable this test is

3.4.4 The panic strategy
The final strategy is the local optimum lowest quality first, known in this paper as the panic strategy. This strategy is an extension of the greedy strategy, with the difference that it will start at the lowest combination of content in stead of the highest. This strategy will in addition to content scaling be able to horizontally scale the infrastructure as well. This is to gain an understanding of how well content scaling work in conjunction with a traditional scaling method.

• If the strategy manages to find an acceptable combination
• How well the horizontal scaling work in conjunction with content scaling
• The reliability of this strategy
• How predictable this test is

3.5 Expected results
These experiments will show how well the framework perform under different scenarios. The data collected will prove how well content scaling works, both on its own, but also as a supplement to a more known scaling method. The data gathered by the framework will show how each change in the environment or in the framework affects the system as a whole.

It is hard to estimate how effective the content scaling will be in reducing the response time, but it probable that it will have some effect. The question is if it is enough to warrant more research on the topic of content scaling.

3.6 Measurements and Analysis
The data collected by the framework will written to log files during execution. Each consecutive trial will append the data to the existing log file. All log entries will be marked with a time stamp so it will be possible to differentiate between the different trials. The content of these log files will then be processed using Microsoft Excel to create plots which will make the data more readable.

This project will seek to understand the impact of the different measures set in motion by the framework. This makes the data gathered at the moment of a change in the network the most interesting. One example would be
when the web server component are content scaled. The graphs should reflect the scaling of the web sites, by showing a difference in response time.

3.7 Appraising properties

The introduction highlights some of the weaknesses of the more traditional scaling methods, such as the speed of the scaling and the problems occurring during high fluctuations in the amount of connections. This project will test the concept of content scaling in an effort to examine what possible benefits this method can bring to the problem of scaling. This is also reflected in the problem statement. There are three key terms that can be extracted out of the problem statement, which needs to be highlighted. Each of these will be a measurement of the level of achievement of this project. These terms are create a framework, content scaling strategies and quality of service.

3.7.1 Create a framework

Being able to properly test content scaling necessitates creating a framework which can properly test the method. Creating a framework allows for testing of different strategies and facilitates for a more stable test environment. The reason why a framework can be considered more stable, is that the experiments will be automated and created from VM images. This leaves less room for human errors and ensures that the servers being tested are identical. This will again result in experiments that are reproducible and more trustworthy data. If this is not accomplished, it is not possible to draw any conclusions, thus making the data extracted useless.

3.7.2 content scaling strategies

While it is critical to ensure a stable test framework, the content strategies themselves play a big role in how the method is used. Each strategy will use a different approach in how content scaling will be utilized to reduce the load on the site. Each of these strategies will try to play to hypothetical strength of content scaling. Whether these different strategies yields different results will be revealed in the result chapter.

3.7.3 Quality of service

The expression keep the quality of service is found in the problem statement. Quality of service is the overall performance of a service perceived by the users of that network[13]. In this paper the service provided will be the web page and images provided by the servers. Quality of Service can also imply the services provided internally on the network like DNS and ICMP, though this is beyond the scope of this paper.

The quality of service will be the term used for the experienced quality
of the service delivered by the framework. The content scaling strategies will strive to keep the quality of service as optimal as possible.

### 3.8 Possible challenges

Administrators has always relied on different tools and software to manage a network, albeit virtual or not. It is important that the framework presented in this project will not be more of a burden than a tool. Creating this framework necessitates careful planning to ensure that the data reported from it is accurate and that the result provided will relieve a potential administrators workload.

To achieve this there are some challenges that needs to be highlighted. The first is the amount of different technologies in this project. These varies from web technologies to virtualization technologies. Making all these different software products cooperate in a predictable fashion may provide some challenges. The second is the problem of complexity. When using this amount of technologies, it is important to always strive for keeping the end result as streamlined as possible. Wanting to change a property of the network through the framework should not require accurate knowledge of the technologies used in the network. It is important that the measurements gained from framework is accurately describing the current situation of the network. This is also important for the artificial intelligence, which is taking actions based on these measurements.

Lastly is the challenge to ensuring that if the program managing the framework where to ungracefully shut down, it will not affect the service components. Meaning that if the program shuts down while changing the web servers, these servers will not be compromised. It is also important to ensure that the framework will not work beyond it delegated resources in the virtualized environment, ensure that other users of the cloud is not affected by this project.
Chapter 4

Results I: Design

To successfully manage a site, an administrator needs to take several variables into account. These variables is not only limited to service metrics like server loads and CPU core temperatures. External variables like user experience and budget is also significant. If any of these metrics falls outside what is considered an acceptable level, the administrator needs to take action to rectify the cause of the issue.

An example could be that the site is suddenly experiencing a surge of traffic, causing the web servers to not being able to serve all of the incoming connections. The administrator gets notified of the situation by an alert, and needs to make decision. The problem in essence is that the amount of data required by users is larger than what the server can deliver. After considering this, the administrator decides to reduce the amount of data delivered to each users, ensuring that every users get served. He then takes action and deploys this new configuration and the site is back to normal.

4.1 Model

The administrator in this scenario represents the manager in the model shown below. The manager is the central "brain" that interprets the data given to it and performs action based on the data it receives, while striving to maintain a compromise between quality and performance. The manager gets the performance data from the users, in this project. Users are the testing tool used to simulate traffic and measure response time.

When the manager wishes to change a configuration, it will act upon a service component, and the service component will in turn execute the specific configurations to the machines themselves.

4.2 Requirements

Before discussing the model it is important to understand what is expected out of this system and to develop a model around these requirements.
1. Only one Manager

2. No parameter dependency

3. No subclass knowledge by Manager

4. No cross site action dependency

5. No domain knowledge by Manager

The first requirement is that the system will only have one Manager. Having more than one will add more complexity to the system and will make it more difficult to implement strategies.

No parameter dependency means that the system will not be tied to a specific manufacturer or operative system. The system will use its own parameters which will be feasible to implement for any system.

No subclass knowledge by Manager. The manager is supposed to work with different strategies that adjust the properties of the site on a high level. It is not the Managers task to actually perform this tasks, since this will be executed by threads. If the Manager had to both make decisions and execute the configurations, the system will be slow and unresponsive. To prevent this, the Manager is denied any knowledge of its own sub classes.

No cross site action dependency. The Manager will adjust the site based on a set of data it receives from the simulated user base and perform actions on these components exclusively. The Manager will only perform one action at a time, thus it is important that no action consists of several small steps. One example could be to adjust the number of web servers. The Manager needs to add a new server and then connect it to the loadbalancer. This is unfeasible since it contradicts the requirement of having no knowledge of its own subclasses.

No domain knowledge by manager. The Manager will also be oblivious of its own domain. This is also information that is not needed, since the Manager is only working with system components and not whit the actual servers and mechanisms inside the site.

4.2.1 The separated components model

In this model the Manager reads and operates each of the different service components separately. This model has only one manager and it operates each service component directly. Each service component functions as an abstraction of the actual servers. This results in that the Manager is able to configure the service components, without having any knowledge of the architecture itself. There are no parameter dependency either, due to the abstraction layer. Any actions taken by the Manager will be translated by each service components to actual commands and configurations. External variables will be the feedback of the users of the site. In the case of this
project, the external variable will be the tool used to apply pressure on the architecture during experiments.

There are however some problems with this model, the fact that the Manager has to work with all the different service components simultaneously will take a toll on the system and make it more complex. Though the main issue is that the model have cross site action dependency. If the manager decides to add another web server, it also needs to inform the loadbalancer service component to ensure that the server is added into its configuration. One could argue that a workaround could be to make the service components communicate with each other, but this would add another layer of complexity and will introduce the need for a synchronisation mechanism between the servers.

Figure 4.1: Figure showing the manager operating separated service components

4.2.2 The control panel model

A solution to this problem would be to add an intermediary entity. Imagine a control panel with gauges and levers. The gauges provide informa-
tion and the levers performs actions. This is what the control panel will do for the Manager in this system. This solution solves the problem of having no cross site action dependency. If the Manager wishes to expand the number of web servers on the site, it will simply pull the lever. The rest will be handled by the control panel. This satisfies both the requirement of having no knowledge about the architecture and no direct communication between the different service components. One added benefit of this model is the ability to aggregate information from different service components to create more meaningful data. One example could be to create a measure that shows the number of database requests compared to actual web site requests. If the site is starting to struggle to keep up with the requests, the Manager can use this metric to decide what to adjust. If there are many requests compared to database queries, it would be more effective to create more web server or to scale up the loadbalancer. If there are a lot of database requests, it would be more effective to scale down the files served by the database.
This model satisfies all of the requirements set in the start of this chapter, but the model has still a possible problem. Imagine the following scenario: The site receives more traffic than the web servers can process. This results in loss of traffic and in turn loss of revenue. This provokes a reaction from the Manager, which instructs the control panel to create a new web server. The control panel in turn initiates the process of creating a new server. The web servers for this particular site are so large that it takes an considerable amount of time before the server is ready for production.

This is poses as a problem. The Manager has no way of knowing if a machine is in production or not. The Manager will still observe that there has
been no change to the quality of service and will issue the same command again. This will result in a case where the Manager will issue the same command again.

VMs will continue to be created until the first starts to have an effect on the site. The end result is a site with a surplus of web servers. This also means that the site will have a delayed reaction to traffic fluctuation, risking dissatisfaction with their service.

4.2.3 Introducing locks

There are two ways of solving the problem of delayed VM creation time. The first solution is to provide a mechanism that predicts the effect of a single web server. This allows the Manager to create a number of new web servers which corresponds with the amount of traffic the site is currently receiving. This solution is quick since it allows for controlled creation of several web servers. There are however some drawbacks with this strategy.

The first and most devastating problem with this solution is that the either framework will be more complex or not corresponds with the set requirements. One of the requirements is "No domain knowledge by Manager". Making it possible for the AI to predict the number of required web servers requires it to understand some of the underlying architecture. One could argue that this can be circumvented by placing this particular mechanism inside the control panel. Though this will create another decision making component in the system and thus contradicting the requirement: "Only one Manager".

The second possible solution is to introduce a locking mechanism on the control panel. When an operation is executed on the control panel, this operation is prevented from being executed again before the first one is finished. This is done by having the "buttons" on the control panel change states. If a command is ready, its state is "ready", when pressed it will change its state to "executing". When the button has changed state to "executing", it is not available to the AI.
This solution will solve the problem of controlling the amount of commands executed. This will also keep the Manager from receiving knowledge of the underlying architecture. One drawback is that it will take more time to create an adequate number of web servers. Though this is considered as acceptable since horizontal scaling is a small part of the project.

### 4.2.4 Final model

This model combines the locking mechanism with the control panel model. This will be the final model which will be created in the implementation phase. Mind that this will only grant a quick overview of the functions and logic of the framework. This model have combined the locking mechanism together with the rest of the control panel model. Making a model that is both simple while still isolating the manager from the rest of the network by the control panel. This model also allows for the Manager to work completely independent of the control panel. This means that the Manager will not stop executing when the control panel executes commands. The manager will utilize threads to operate the control panel. The programming logic and how to achieve the functions in the model will be covered in the implementation chapter.
4.3 Strategies

Another big aspect of this project is not only the framework, but the strategies as well. Each strategy will be implemented as a part of the Manager in such a way that the framework does not need to be altered to make sure that the strategy will function. This section will go into more detail about the idea and purpose of each of these strategies.
4.3.1 Deciding factors

Before describing each strategy it is best to explain what the strategies are trying to achieve and how the strategies will evaluate their own performance. Each strategy have two factors to relate to. The first is the performance factor and the other is the quality factor. The performance factor is variables like response time and availability. Having a high response time or low availability are serious issues which can lead to long load time of the service, if any service at all. These will severely affect the users and can lead to a loss of revenue.

The other factor are the quality factor. The quality factor describes variables like image sizes, objects per page, the grade of interactivity etc. Having these variables degraded, while still vexing for the users, are not as bad as not being served at all. This trade off is what content scaling is trying to achieve. The strategies will try to find a balance between performance and quality. Balance in this case is serving a service of the highest possible quality, while still not affecting the performance.

To find a balance, the two factors will be graded on a numeric scale, one for each factor. The scale represent the distance the factor is away from an optimal state. Meaning that a higher number represents a less desirable state. One added benefit with having such a scale is that it is in fact scalable. Which means that the number of different states are unlimited, since the scale has no maximum value.

Note that the frameworks only way of reducing the performance value is to make changes to the quality of the service. This is also why the performance has a higher importance. If the quality was considered as important as the performance, it would be extremely hard to design proper strategies.

By using the two scales derived from the performance and quality factor, it is possible to create a chart where the X axis represent the quality and the Y axis represents the performance. The factors are now coordinates within a chart. The most desired state is now the coordinates 0,0, but everything below 2,2 can be considered optimal. Arranging the factors into a chart does also open up for the possibility for algorithms that orients itself by the
chart. Such a coordinate is known in this paper as a combination.

![Chart created by using the performance and quality factor](image)

Figure 4.6: An example of how an algorithm can orient itself by using the chart.

As mentioned, performance is considered to be the most important factor of the two. This is to ensure that the site will always be available. This is reflected in the figure above. After checking the outcome of four possible configurations, it opts for choosing the one resulting in a more optimal performance state, all tough at the expense of the quality.

### 4.3.2 Exhaustive strategy

The exhaustive strategy is the most rudimentary of the strategies. The core algorithm of the exhaustive algorithm is to test every possible configuration of the content and then decide on the combination that is decided to be the most optimal, thus the name exhaustive. Each configuration will be held for some time and then the performance and quality factor will be measured. The strategy will then pick the combination with the lowest possible performance value (Y value) and the lowest possible quality value (X value) below that performance value.
Figure 4.7: The exhaustive strategy will try every combination and pick the most optimal. In this example, configuration 5 is the most optimal.

The example above shows how the exhaustive strategy prioritizes combinations that have the lowest performance value, and the lowest quality value, in that order.

4.3.3 Greedy strategy

The greedy strategy does not test all of the combinations before making a decision. The greedy strategy will instead pick a combination that have a slightly higher quality value, and then wait for the result. If the performance value is still too high, it will lower the quality again until the performance value is acceptable. In other words, the greedy strategy is to continue to degrade the content until it is satisfied, hence the name greedy.
Figure 4.8: The greedy strategy will continue to degrade the quality until the performance value is acceptable. In this example iteration 4 is accepted.

The example above shows the strategy starts at iteration 1, and continues to degrade the content once per iteration until the performance is as optimal as possible. When the results are acceptable, the strategy will then periodically try to decrease the quality value. If this does not affect the performance, it will continue to decrease. If the performance is affected, the change will be reverted.

It is also worth noting that the greedy strategy will not necessarily find the most optimal combination, it will find the first acceptable combination. If there is no acceptable combination the strategy will simply run with the highest possible quality value.

4.3.4 Panic strategy

The last strategy to be implemented is the panic strategy. The panic strategy is similar to the greedy strategy in that it works by iterations, but there are some differences. While the greedy strategy starts to increase the quality value when the performance value has increased. The panic strategy will instead implement the lowest possible quality of the content and then
slowly increase the quality until the performance value again increases. It is almost as the framework is startled by the increase of the performance value and is scrambling to keep that value under control. Thus the name panic.

Figure 4.9: The panic strategy will immediately set the highest quality value possible and then continuously decrease the quality. In this example iteration 3 is accepted.

As seen in the figure above, when the performance value increases, the strategy goes for the maximum quality value. The strategy then steadily decreases the quality value until the performance is affected. In the example above, iteration 3 is accepted. Iteration 4 is not accepted and the quality is increased again.

The panic strategy will not always “panic”. The strategy will work in the same fashion as the greedy strategy when there is only a small increase of the performance value.

Another difference from the greedy strategy will be its ability to horizontally scale machines when needed. The panic strategy will horizontally scale when it panics or when it can’t get an acceptable performance value.
The panic strategy will try to remove the extra server when both quality and performance is again optimal.
Chapter 5

Results II: Implementation

This chapter will cover the implementation of the framework. The chapter will start by describing the environment that the framework will work in. This will be the setup of the different service components and how they interact with the framework. This will include the crawler, which is responsible for measuring the response time of a page request from the web servers. Finally the manager itself with its corresponding strategies will covered. The purpose of this chapter is to give the reader a deeper understanding of the framework and the strategies presented in the design chapter. The framework is implemented using python, but pseudo-code will be used to make the principals more understandable.

5.1 Implementation - Creating the environment

To test the effectiveness of content scaling. It is necessary to create an environment that simulates an actual site. Such an environment consists of several service components that interacts together to creates a service. The service simulated is a simple web page containing images and text. This service consists of three different service components.

- Loadbalancer
- Web server
- Image server

5.1.1 Loadbalancer

The loadbalancer used in this project is an Ubuntu 16.04 machine running HAProxy. It will listen on port 80 for the web server and port 8080 for the image server. That means that HTTP requests connecting to the port 80 will be served a web page and connections to the port 8080 will be served an image. The loadbalancer is configured this way to ease the implementation of the tool used to measure the response time.
There are two parts that needs to be defined for HAProxy to function properly. Frontend and backend. Frontend is used to distinguish incoming traffic from another. In this environment there is web-frontend and image-frontend.

In this configuration any http traffic toward port 80 is directed to the web-backend and any http traffic toward port 8080 is directed to the image-backend. These backends are the IP addresses to the servers, so any traffic will be redirected to its respective server.

The backend might change during the course of the experiments. One example would be during horizontal scaling where an extra server is added. To add another server the configuration file needs to be changed so that the newly spawned server will receive traffic from the loadbalancer. This is accomplished by two scripts which are called from the framework server. addConfig.py and delConfig.py. addConfig.py takes 3 arguments. Type of server to be added, reference name for HAProxy and finally the IP address.

When the manager wants to create a new web server. It will connect to the loadbalancer using ssh and start the script. An example would be:

```
ssh ubuntu@loadbalancer \ python addConfig.py web web2 10.10.10.12
```

The configuration file of the loadbalancer will now look like this:
delConfig.py does the complete opposite of addConfig.py, it removes a server from the backend to reduce costs of running the site.

5.1.2 Web server

The web server is the main service delivered to the user. This is in the sense the web page that the user will be served when connecting to the site. The web server runs Ubuntu 16.04 and runs Apache 2 for serving its web page. The server is listening for HTTP connections on port 80. The server does also have PHP installed, since this is a part of the page served.

Web page

The web page in this framework serves two major purposes. The first is to simulate load. It would be impractical to launch hundreds of connections to create a load that would affect the service. This could possibly introduce variables that would pollute the data extracted from the framework or cause the virtual environment to behave unexpectedly. The other is to serve a site with images and text that can be changed on demand.

Weight

Weight is the term given for simulating traffic load. When the web server receives a request, it will execute a computational operation which will strain the CPU of the server. This operation is two nested for-loops where the inner loop creates an array with 10 000 random numbers from 0 to 100 000, and then proceeds to sort the array. The outer loop decides how many times this operation should be repeated. This number is known as weight. The higher the weigh, the higher the computational load is for each request.

```bash
FOR i in $weight {
    FOR j in 100000 {
        create array
        fill array with random numbers between 0 and 10 000
    }
    sort array
}
```

Format

The second part of the web page is the block of content delivered to the user. One block consists of one image URL and a block of text. The number
of these blocks served per session is dictated by the format value. If this value is 10, the web page will serve the client with 10 image URLs and 10 specimens of the text. The format value is found in the file format.txt on the web server. This file can easily be changed by the Manager to reduce the number of repeats.

```bash
FOR i in $format
  serve text
  serve image URL
```

### 5.1.3 Image server

The final part of the service is the image server. This machine is similar to the web server regarding software and setup, the difference is that it serves images rather than a web page. In this framework the image server will only serve one image in three possible qualities. These qualities are:

1. large.jpg - 7.22 MB
2. medium.jpg - 2.27 MB
3. small.jpg - 167 KB

To effectively change what image are to be served to requests, there is a fourth file, a symbolic link called image.jpg. Clients will always request the image.jpg which will point to one of the real images. Changing the image quality is then performed with a simple `ls` command.

```bash
rm image.jpg
ls -l medium.jpg image.jpg
```

The first command removes the old symbolic link. The second command creates a new symbolic link to the medium image. Any requests for image.jpg will now serve the medium sized image.

### 5.2 Crawler

The original plan in this project was to incorporate an existing HTTP testing tool into the framework. The most important property of this tool was that it had to be able to not only check the response time of the site, but also being able to parse and visit all the links present on the site and then measure the loading time of the site as a whole.

After extensive research it turned out that no common tool had this ability, resulting in the sudden need of developing the tool from scratch. This tool ended up being Crawler. The name Crawler comes from its similarities of a typical web crawler. A crawler is a program that visits a site to map all
the links present on the site. The crawler then visits all the links found and starts the process over again. This kind of crawling is also known as web indexing, used to populate the database of a search engine[22].

This is also what Crawler is doing. Crawler inspects a web page, finds all the links present on that exact web page and proceeds to download each image URL it finds. Crawler is also multi threaded, meaning that it can simulate several users at the same time.

5.2.1 Program logic

There are two requirements that the tool needs to meet. The first is the ability to time the response time of the site as a whole and to be able to simulate traffic from several users. It is impractical to create one VM per simulated users and a nightmarish task to aggregate all that data in a sensible fashion. This is why Crawler relies on threads to measure the site performance. Each thread is known as a worker. Each worker is tasked with visiting the site, time the period between the first GET request and the time when every image has finally been loaded. The worker then post its result in its variable. This is the pseudo code of the worker object:

```python
class worker:
    ID-number = 0
    lastTime = -1

    Run when I am created(number in list):
        ID-number = number in list
        worker-thread = function work

    function work:
        while my ID-number are less than maxThreads:
            start = system time
            crawl the loadbalancer site
            end = system time
            lastTime = end - start
            delete myself
```

The worker is an object which can only perform one task which is to crawl the site. It will do so continuously as long as it is allowed to stay alive. The worker knows this by looking at the global variable maxThreads. maxThreads is the number of workers allowed to exist. If the worker was created as number 9 it will continue to run until maxThreads are changed to a number which is lower than 9. The worker will then stop and delete itself from memory.

The base framework of the program is just a basic encapsulation whose task is just to manage the workers and to provide those workers with necessary information. This information consists of two parts, each read from each own file. The first part is the IP-address of the loadbalancer. This IP address is renewed between each run of the Crawler and is part of a mecha-
anism to direct the workers to the correct IP address. The second part is the number of workers that are to send request at the same time. Both of these files can be changed while the Crawler is running.

```plaintext
GLOBAL list listOfWorkers
GLOBAL list averageTime
GLOBAL int maxThreads
GLOBAL string loadbalancerIP

while True:
    fileAmount = read file numberOfThreads.cfg
    IF maxThreads < fileAmount
        maxThreads = fileAmount
        Create more workers so amount equals maxThreads
    ELSE IF maxThreads > fileAmount):
        maxThreads = fileAmount
    ELSE
        while maxThreads == fileAmount
            IF maxThreads != 0
                <Calculate averages>
            ELSE
                loadbalancerIP = read file loadbalancerIP.cfg
    Referring to the code above. Crawler is continuously checking the file numberOfThreads.cfg to check if the number has changed. If this is the case, it will adjust the number of workers accordingly. Let us imagine that Crawler is running a session with 40 connections. When the file is changed to 20, Crawler will terminate 20 workers from its inventory, effectively reducing the number of simultaneous connections to 20.

If the number of connections is reduced to zero, Crawler will then terminate all of the workers and start to check the file loadbalancerIP.cfg for changes. When the IP address in the file is changes, so will the loadbalancerIP variable in the program. These two functions allows Crawler to behave like a service instead of a program. The advantage of doing the service oriented approach is that it can be altered as it runs instead of having to restart the program every time the number of connections needs to be changed.

5.2.2 Calculating averages

The main task of Crawler is to measure the average response time of the workers. This is done by collecting the last registered response time from each worker object and aggregate this value into an average which is placed in a list for later use. If Crawler is running a session with 40 workers, Crawler will add the response time from each worker into a total, which again is divided by 40. This value is then put into a list which are 10 items long. Crawler then takes the average of that list to get the final average. This is a complicated method, but it will ensure that no extreme value will severely affect the total response time.
WT = Worker time
AVG = Average of all workers

```
1 FUNCTION crawl(weblink)
```

Figure 5.1: Figure showing how Crawler calculates the response time with 10 workers

This solution is also scalable in the sense that it can calculate the average response time no matter the number of connections.

5.2.3 Crawling mechanism

The final module of the Crawler is the crawling mechanism. This is the part of the program that performs the actual crawling part of the program. Each worker creates a thread which continuously times the execution this function. Below is the pseudo code for this crawling mechanism.
The code opens the web page given to function as an argument. The page is then searched for any links tagged with 'img'. The crawler then downloads the image into /dev/null. This simulates that the webserver is loading all the images, though not perfectly. Usually a web page try to load all the images simultaneously, not one by one. Though this is by far the less time consuming way of solving the image issue.

5.3 Crawler scheduler

As seen in the prior section, Crawler is able to change its configuration during execution by a remote machine. This is done by the crawler scheduler. Crawler scheduler is a simple script which only task is to change the number of connections and to give Crawler the IP address of the loadbalancer. Crawler scheduler reads from the file crawler-scheduler.cfg to understand when to change the number of connections. This is an example of how this file could look.

This particular configuration file tells the scheduler to first launch 40 connections for 120 seconds, then reduce it to 20 for 60 seconds and then finally stop the Crawler by reducing it to 0. Below is the code which parses and executes the configuration file.

When the scheduler reaches the end of the file, the program will terminate. This is why the configuration file should always end on 0 so that the Crawler will stop sending traffic.

5.4 Manager

The manager is the program that is responsible for managing the site. As shown in the Design chapter. The manager consists of two main parts, the manager and the control panel. The manager has the role of carrying out the strategy it has been given, and the control panel module is responsible
for carrying out the operations given by the Manager module. This role based concept made it more sensible to use an object-oriented approach when creating this program. Especially when taking into account the need of representing servers in a sane way.

5.4.1 Logical structure

Since this particular program is object-oriented, it has several classes. Each class represents either a major part of the system like the Manager and the control panel or an actual server. Figure 5.2 visualises the different objects and how they relate to each other. The Manager contains the control panel. The control panel has a list containing all of the servers, each represented by an object. The web server class can either be a web server or image server depending on the type it is given upon creation.

![Diagram of Manager program objects and their relationships](image)

Figure 5.2: Figure showing the objects in the Manager program and how they relate.

Each function of each object is also shown in figure 5.2, but variables not involving objects and the initial and delete functions have been omitted to make the figure more readable. Though helpful, this figure is a far step from giving the reader a full understanding of the inner workings of this program. To achieve this, this section will explain the program step by step from starting the program until the invoking of managers thinkTank function. Which is the function that operates the strategy.
5.4.2 Manager initiation

When the Manager is started the first thing that is created is the Manager object itself. This object will then create everything else related to manage the framework. This is the __init__ function of the manager class. __init__ is the function of a class which is always executed when an object of that class is created, making it ideal to set up the rest of the logical environment.

```python
FUNCTION __init__ (maxWebservers):
    maxWebservers = maxWebservers
    self.ControlPanel = ControlPanel()
```

The only thing this initiation function does is to announce the maximum amount of servers the Manager is allowed to create and to create the control panel. The reason why the Managers initiation function is rather small is to meet the criteria of the separation between the Manager and the details of the actual network. It instead creates the control panel, which in turn interacts with the environment to create the site itself.

5.4.3 Control panel initiation

The initiation function of the control panel is larger and more complex. The reason is that this is the class that is responsible for managing the servers. As seen in the python code snippet below, there are three dictionaries that are initiated with the control panel. The dictionary named "servers" is where every server object will be placed. The two other dictionaries are the locking mechanisms described in the design chapter. ListOfOperations is the dictionary responsible for locking servers that are currently being configured. List of creations is the dictionary responsible for telling the Manager that servers are being created, prohibiting that multiple servers are being created before the effects has manifested itself on the site.

```python
FUNCTION __init__:
    currentWebservers = 0
    INITIATE dictionary servers
    INITIATE dictionary listOfOperations
    INITIATE dictionary listofCreations
    createLoadbalancer()
    addWebsevers(type = web)
    addWebsevers(type = image)
```

After the dictionaries has been initiated, the control panel carries on creating the bare minimum of the site. One loadbalancer, one web server and one image server. This is done by calling the create-function correlating to the server that are to be created. Only addWebsevers will be covered in this thesis, since both addWebsevers and createWebser work in a very similar fashion. Consult appendix A for the complete code.

```python
FUNCTION addWebsevers(typeOfServer):
    hostname = create hostname
```
When the control panel initiates the function addWebservers it is either with the argument "web" or "img". This argument decides what type of server that will be created. The control panel then changes the value of the key with the corresponding type in the dictionary listOfOperations from "Ready to "Executing". This is to prohibit the Manager from trying to configure VMs that are not yet in production. The state will be changed back to "Ready" when the object has been created.

5.4.4 Webserver initiation

A web server object is representing an actual VM present in the virtual infrastructure. When this object is initiated it will start communicating with the nova API. The nova API is the program's connection to the Openstack environment, making it possible to create and delete machines at will. The web server object is also responsible for configuring the content of the one server it is representing.

The program is not however creating VMs completely from scratch. It is using a snapshot of a VM, formally known as an image. The advantage with using an image is that it eliminates the need of installing packets needed to put the VM in production. This again saves time and also assures that the experiments conducted are all performed on theoretically identical machines.
When a web server is created, it will first use the nova API to create a VM known as an instance. It does take an unknown amount of time between the order for a VM is placed until it is actually ready for production. This is where the two spin locks come in. The first spin lock is constantly searching the Openstack database for the VM it has created. The goal of this is to find the IP address the VM has been given. This IP address is stored within the ipAddress variable of the object.

The second spin lock is a loop which uses netcat[14] to constantly prod the ssh port of the VM. When the server responds correctly to the prod, the server is deemed fit for production. The last step of the process is to notify the loadbalancer about the new web server. This is done by remotely activating addConfig.py described in section 5.1.

The program does also have the ability to constantly monitor the metrics of the server itself. This could be statistics like CPU load, memory available and number of connections. The daemon thread is the function with this ability. Though this was never used in the experiments due to time constraints, so this functionality will not be explained any further.

5.4.5 Thinktank

When the control panel has finished setting up the environment, the crawler scheduler is then initiated. This is done outside of the objects since the feedback loop is supposed to be an external functionality.

Now that the site has been fully set up, Managers thinktank function is then initiated. The thinktank is where the strategy is found, so the contents of the function will vary depending on which strategy is used. Though they all share some similarities. Every strategy involves getting feedback from the control panel, operating the control panel, and to calculate the quality and performance values.

```
FUNCTION thinkTank:
    timeLimits = [10.0,14.0,18.0]
    FOR combination of two digit value of '123':
        possibleConfigurations.append(combination)
```

This is the start of the thinktank. It starts by creating a list of its given time limits. The Manager will judge the sites performance by consulting this list. The for-loop is creating a tree of possible configurations of image quality and format. These configurations is decided by a two digit number. In this thesis there are three different images and three different format values. Where the number 1 means highest quality and 3 represent the lowest quality possible. This means that 3,3 is the lowest possible quality the site can deliver, while 1,1 is the highest. It might seem a bit unintuitive
that a lower number represent a higher quality, but note that doing it this way makes it much easier to add an even lower quality. The tree generated in the code above looks like the following figure.

Figure 5.3: Figure showing the tree created when Manager is mapping out all possible configurations. This example have three different format configurations and three different image quality configurations.

There is also another two digit value that is important to the manager and this is the current measured quality of the site. The first digit is known as the performance number. The performance number is calculated from the average response time from Crawler. The algorithm to accomplish this is simple. When the manager has gotten the response time from the crawler through the validataSitePerformance function, it will consult the timeLimits list. If the response time is lower than entry 0, the performance is 1, if the response time exceeds entry 0 but is lower than entry 1 the performance is 2 and so forth.

The second digit is the quality value. This value is calculated from the configuration value. The mathematical formula for this process is the following.

\[
\text{quality} = \frac{(\text{format quality} \times 2) + \text{image quality}}{2}
\]

In the case for this thesis, the highest configuration number is three. The quality can then vary from 1.5 which is both format and image quality set to 1. And the highest possible value is 4.5 when both format and image quality is set as 3. There is te reasons why the format value is weighted higher in this formula. The first is to create a diversity in the resulting numbers. If both where weighted equally almost every configuration would equal 2, which is not very informational. The second reason is that a lower format would be more vexing for a user than a lower image resolution. Weighting the formula this way makes the Manager more reluctant of changing the format value.
IF any of the servers has the state 'Executing':
    wait 5 seconds

    httpTimes = validateSitePerformance()

    i = 1
    performanceValue = 3
    FOR limit in timeLimits:
        IF httpTimes < limit:
            performanceValue = i
            break;
        i += 1

    qualityValue = ((current format quality + 2) + current image quality) / 2

The following three sections will go into further detail about each individual strategy, explaining how each strategy was implemented.

5.4.6 Exhaustive strategy

The exhaustive strategy is the simplest of the three strategies. This strategy creates a tree of all possible combinations of format and image quality and then proceeds to test out every combination to find the configuration that keeps performance at an acceptable level. While still having as high as a quality as possible.

Determining the number of possible configurations is done before the strategy is executed. As shown in the prior section this is turned in to a tree structure which are traversed by depth. Resulting in the following order of configurations to be tested.

Table 5.1: Table showing the order of how the configurations are tested

<table>
<thead>
<tr>
<th>Test order</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,1</td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>1,3</td>
</tr>
<tr>
<td>4</td>
<td>2,1</td>
</tr>
<tr>
<td>5</td>
<td>2,2</td>
</tr>
<tr>
<td>6</td>
<td>2,3</td>
</tr>
<tr>
<td>7</td>
<td>3,1</td>
</tr>
<tr>
<td>8</td>
<td>3,2</td>
</tr>
<tr>
<td>9</td>
<td>3,3</td>
</tr>
</tbody>
</table>

To keep the Manager from having any knowledge of the actual network, the Manager will only issue the configuration numbers to the control panel via threads. The control panel will interpret these numbers to actual commands.
1. initiate thread with target ControlPanel.scalewebsevers
2. and with arguments combination[0] and 'web'
3. initiate thread with target ControlPanel.scalewebsevers
4. and arguments combination[1] and 'image'

The manager will then wait a set amount of seconds before measuring the result of the configuration, before a new one is implemented. The program exits by implementing the configuration that is deemed to be the most optimal.

### 5.4.7 Greedy strategy

The greedy strategy is a quicker, but a less precise strategy compared to the exhaustive strategy. The exhaustive strategy checks every strategy regardless if the performance quality is higher than 1. The greedy strategy will perform a depth search of the configuration tree until the performance quality is 1. When the Manager achieves this, it will then try to increase the quality and wait for the results. If the performance quality does not increase it will continue to slowly increase the quality. Though if the performance is affected, the Manager will revert the configuration, and try again later.

```plaintext
1. IF performance > 1.0:
   2. move the configuration pointer upwards
3. ELSE IF performance == 1.0:
   4. IF user quality higher than 2:
      5. move the configuration pointer downwards
   7. ELSE:
      8. //Quality is optimal and response time is acceptable
         9. Do nothing
```

The manager orients itself by using the configuration pointer. As stated, all the configuration is stored in a tree structure, and this structure is stored in a list. The configuration pointer will traverse this list by index. If the manager wants to lower the performance quality, it will move the configuration pointer up one index, to a lower quality index.
5.4.8 Panic strategy

The last strategy is the panic strategy. This strategy in essence the same as the greedy strategy with two major differences. The first being its ability to horizontally scale web servers as seemed fit. This means that it not only can reduce response time by changing the content provided, but also by creating more servers. The other difference is its ability to panic.

In the greedy strategy, the manager is using the configuration pointer to slowly change the content until the performance value is set to 1. When the Manager panics it will immediately change its configuration to the lowest possible while simultaneously spawning a new server. This somewhat rash and panic-like decision is what gave the strategy its name.

Panic is not always invoked when the performance of the site is falling. The trigger for panic mode is: If the the newly registered response time is twice as high as the prior registered response time, then panic. The idea behind this is to try to catch a steep increase in connections before it actually hits. A sort of preemptive effort to reduce the amount of lost connections.

```
  IF performance > 1:
      numberOfIdealStates = 0
      IF (new response time > last response time * 2):
          configurationPointer = highest possible index
          IF controlpanel is able to create more VMs:
              IF last created server was 'web':
                  serverToCreate = 'image'
                  tell control panel to create a web server
              ELSE:
                  serverToCreate = 'image'
                  tell control panel to create an image server
```
This is the core of the panic strategy. If performance is over 1 and the new response time is twice as high as the last recorded response time. The Manager will move the configuration pointer to the highest possible index, and create a new server. The manager will differ between what type of server to create depending on what type was created last. This is to keep an even number of web servers and image servers.

There is also a number which is reset when the performance exceeds 1. This is the variable numberOfIdealStates. This variable is Managers method of counting the number of times the performance has been in an acceptable range and the configuration is at the optimal state of 1,1.

\[
\begin{align*}
1 & \text{IF performance == 1 AND configuration == (1,1):} \\
2 & \quad \text{numberOfIdealStates} = \text{numberOfIdealStates} + 1 \\
3 & \quad \text{IF numberOfIdealStates} \geq 5 : \\
4 & \quad \quad \text{Delete VM} \\
5 & \quad \quad \text{numberOfIdealStates} = 0
\end{align*}
\]

When the state has been ideal 5 times in a row, the Manager will then reduce the number of servers by 1 and reset the variable. This makes sure that the Manager will not constantly create and delete servers in the cases when the performance is registered around the required response time.
Chapter 6

Results III: Measurements and analysis

This chapter will cover and subsequently analyse the results of all the experiments conducted during this project. The chapter will start by covering the functional testing of the framework. Then proceed to cover the data extracted from the framework during each of the tested strategies.

6.1 Functional testing

Before attempting to run experiments on the framework, a series of smaller functional tests were performed. The purpose of these tests were to ensure that the framework would be able to compile and produce predictable results. These three tests were described in Functional testing in the chapter 2 Approach.

• Check connectivity between the service components and the VM containing the Manager
• Ensure that the service provided by the service components are working as intended
• Create and delete VMs

These tests were fairly general, since the framework where yet to be designed. The tests specified in this section where the implementation of these functional tests divided three categories of software testing: Unit testing, Integration testing and finally system testing.

6.1.1 Unit testing

The first step of the functional testing where the unit testing. Unit testing is the method of splitting the program in to smaller sized modules which are more manageable to test. The unit tests were conducted during the implementation phase. These tests where performed by creating the module in an independent script and subsequently adjust that module until the results are passable.
Unit tests

- Run a unit test of the module responsible for creating VMs
- Run a unit test of the module responsible for deleting VMs
- Locally add a web server entry in the HaProxy configuration file by invoking the addConfig.py script on loadbalancer
- Locally remove web server entry in the HaProxy configuration file by invoking the removeConfig.py script on the loadbalancer
- Locally change the configuration of web- and image server using the terminal
- Run a mock-up of the Manager, to ensure consistent thread behaviour
- Run a mock up of the crawler, to ensure consistent thread behaviour

The framework are also using a range of Linux system commands. These where also tested by first executing them from the terminal before implementing them in python.

6.1.2 Integration testing

Integration testing is the method of combining the modules into a single autonomous system. This part of the testing consists of locating and ridding the system of software deficiencies that may appear when attempting to merge them. This step does also include all functionality that requires communication between different VMs. The networking tool curl[15] where frequently used during this process due to its ability to specify HTTP queries from terminal.

Integration tests

- Start an SSH session with every VM in the network
- Connect to port 80 from crawler to loadbalancer, to ensure connectivity to web server
- Connect to port 8080 from crawler to loadbalancer, to ensure connectivity to image server loadbalancer
- Curl to web server from crawler, using different URLs to ensure consistent replies
- Curl to image server from crawler, to ensure that server provides the correct image quality
- Short test run of the crawler, ensuring consistent output
- Remotely reconfigure the loadbalancer, by invoking the scripts on the loadbalancer
• Execute crawler-scheduler.py, ensuring that the crawler can be controlled remotely
• Test a crawler schedule with varying amount of connections

6.1.3 System testing
Finally the framework was tested as a single unit. Basically doing a full test run of the Manager with each strategy.

System tests
• Test run of framework equipped with exhaustive strategy
• Test run of framework equipped with greedy strategy
• Test run of framework equipped with the panic strategy
• Review the format of the data extracted from the framework

6.2 Strategy 1: Exhaustive approach
After the functional testing where completed the framework where ready for the experiment regarding the effects of content scaling. The first strategy to be tested where the exhaustive approach. This strategy differs from the other two since it will not run continuously. It will try every combination of configurations and settle with the one that offers the best results. This means that there will be no point in varying the number of connections during a trial, since this will only confuse the algorithm.

6.2.1 Flat: 40 connections
This experiment consist of 5 trials. Each trial have similar specifications, meaning that the simulated traffic and the thresholds of the Manager are perfectly similar. Note that the duration of the exhaustive tests are only superficial, since the result will not change after the algorithm has decided on the best configuration. This procedure takes about 10 minutes

Experiment specifications
• Number of experiments: 5
• Seconds between each configuration: 60 seconds
• Crawler schedule:
  1. 40 connections : 1200 seconds (20 minutes)
• Performance thresholds:
  – 1: Response time < 10
Experiment results
As seen in the charts below. All of the trials follow a trend. Each trial starts with a very high response time. This proves that the starting configuration, which is the most resource demanding, is overloaded. This is interpreted as an unacceptable level for the Manager, which then starts to iterate through its configurations.

After the initial peak, the response time is severely lowered into more acceptable levels. One important trend to note is how the response time increases and falls with the image quality. This points to the fact that the image server is of higher importance related to response time than expected.

![Chart comparing all of the results from the flat exhaustive experiment with 40 connections](image)

Figure 6.1: Chart comparing all of the results from the flat exhaustive experiment with 40 connections

Overlaying every trial in a plot where time has been replaced with numbers of measurements makes the similarities between the trials even clearer. Every trial follow a similar line with an initial high response time, but is drastically reduced as soon as the content scaling takes effect. Each trial then follows the same trend of having the response time increase in unison with the image quality.

It is also worth pointing out that the fifth trial deviates slightly from the others. This trial has on average a higher response time than the others, but it do have the same tendency to rise and fall with the image quality. The reason behind the deviation is probably due to the Crawler, which has shown itself to be unreliable.
Figure 6.2: Spaghetti plot comparing each trial from the exhaustive experiments with 40 connections

**First trial**
All the trials are behaving similarly, but it there are some points that are easier to explain using a single trial. This graph shows how the response time, registered by the crawler represented in orange, are affected by the different configurations, each represented by its own colour.

Figure 6.3: Figure comparing all of the results from the flat exhaustive experiments with 40 connections

This data clearly shows that differing the configuration of the servers
affect the overall response time of the web requests. It is evident that the image size greatly affects the overall performance, this is visualized by the repeating zig-zag pattern in the graph. The response time is sinking in accord with the image size. When the image is at the lowest quality (3). The overall response time is around 10 seconds independent of the format. In this particular trial, the preferred configuration was with format 1 and image 3.

Table 6.1: Table showing the Manager decisions during the first trial with the exhaustive strategy

<table>
<thead>
<tr>
<th>Time</th>
<th>Measured performance</th>
<th>Measured quality</th>
<th>Format</th>
<th>Image quality</th>
<th>Avg response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>41:11,2</td>
<td>3</td>
<td>1,5</td>
<td>1</td>
<td>1</td>
<td>69,92</td>
</tr>
<tr>
<td>42:26,6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>13,76</td>
</tr>
<tr>
<td>43:37,6</td>
<td>1</td>
<td>2,5</td>
<td>1</td>
<td>3</td>
<td>8,35</td>
</tr>
<tr>
<td>44:54,4</td>
<td>3</td>
<td>2,5</td>
<td>2</td>
<td>1</td>
<td>17,26</td>
</tr>
<tr>
<td>46:09,3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>12,49</td>
</tr>
<tr>
<td>47:26,2</td>
<td>1</td>
<td>3,5</td>
<td>2</td>
<td>3</td>
<td>9,70</td>
</tr>
<tr>
<td>48:37,0</td>
<td>3</td>
<td>3,5</td>
<td>3</td>
<td>1</td>
<td>17,63</td>
</tr>
<tr>
<td>49:51,4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>13,71</td>
</tr>
<tr>
<td>51:00,5</td>
<td>2</td>
<td>4,5</td>
<td>3</td>
<td>3</td>
<td>10,14</td>
</tr>
</tbody>
</table>

As seen in the table, there is two configurations that meet the performance requirement of 10 seconds or lower. These are configuration 1.3 and configuration 2.3. However since 1.3 represent a configuration with less downgrades, it is chosen.

Figure 6.1 and 6.2 only shows the data recorded until the last configuration. Figure 6.3 shows the data recorded after the last configuration was tested. As seen in the chart, after having a stable response time for about 10 minutes after the last configuration, the response time starts to increase. This is caused by a defect at the Crawler. After about 20 minutes of run time, the data extracted from the Crawler starts to become very unreliable. Though this issue seems to be minimal at the start of each trial.

6.3 Strategy 2: Greedy approach

The second strategy is the greedy approach. This is a strategy that differs from the exhaustive by its inability to find the optimal configurations. The greedy strategy will adjust the configuration of the site until it has reached an perceived optimal. This means the the Manager will continue to downgrade the service until either the performance requirement is met, or that the quality cannot be downgraded any further.

6.3.1 Flat: 40 connections

This experiment consists of 5 trials. Each trial is the set have been executed with the same properties. Meaning that performance and quality thresholds are identical as well as the simulated load on the site.

Experiment specifications
- Number of experiments: 5
- Seconds between each configuration: 60 seconds
- Crawler schedule:
  1. 40 connections : 900 seconds (15 minutes)
- Performance thresholds:
  - 1: Response time < 10
  - 2: Response time > 10 AND Response time < 14
  - 3: Response time > 14

### 6.3.2 Experiment results

Examining share the same trends as the exhaustive set. Each trial alternates between two configurations in an effort to keep the response time within an acceptable level. These alternations may seem large, the reason behind this is that the system is alternating between the image size, which had a larger impact than first thought. Trial 3 and 5 differs from the set. This is probably due to the crawlers inconsistency.

![Greedy: 40 connections, complete experiment](image)

**Figure 6.4:** Figure comparing all of the results from the flat exhaustive experiment with 40 connections

The spaghetti chart strengthens the assertions taken from figure 6.4. The response time starts at a very high number, but this is quickly subdued by the content scaling. After the initial peak the strategy alternates between configuration 1,3 and 1,2 except for the third and fifth trial. These trials have a bit higher average response time, which again is probably due to the faults of the Crawler.
Figure 6.5: Spaghetti plot comparing each trial from the greedy experiments with 40 connections

6.3.3 Second trial

Again the response time starts at a very high rate, but an adjustment of the image size improves it greatly. After the 2nd adjustment the response time dips below the 10 seconds requirement. The Manager will notice this and subsequently try to upgrade the image size. This results in an unacceptable response time of 11.73 seconds causing the Manager to downgrade the image again. This alternation between image quality will continue until the amount of traffic drops, or to the end of that particular run.
Figure 6.6: Figure comparing all of the results from the flat exhaustive experiment with 40 connections

This table shows the data the Manager reads during the trial and also what actions it decides to perform on the site. According to the table the Manager immediately starts to scale the content from the ideal state 1,1 to the next which is 1,2. The response time is still unacceptable, so it decides to scale down yet again. The response time is now 9,26 which is acceptable. The Manager then alternates between these two configurations as long as the number of connections do not change.

Table 6.2: Table showing the Manager decisions during the second trial with the greedy strategy

<table>
<thead>
<tr>
<th>Time</th>
<th>Performance</th>
<th>Quality</th>
<th>Avg responsetime</th>
<th>Current config</th>
<th>Next config</th>
</tr>
</thead>
<tbody>
<tr>
<td>27:10,5</td>
<td>3</td>
<td>1,5</td>
<td>61,92</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>28:24,6</td>
<td>2</td>
<td>2</td>
<td>13,68</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>29:31,9</td>
<td>1</td>
<td>2,5</td>
<td>9,26</td>
<td>1,3</td>
<td>1,2</td>
</tr>
<tr>
<td>30:42,2</td>
<td>2</td>
<td>2</td>
<td>11,73</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>31:49,5</td>
<td>1</td>
<td>2,5</td>
<td>8,52</td>
<td>1,3</td>
<td>1,2</td>
</tr>
<tr>
<td>32:59,7</td>
<td>2</td>
<td>2</td>
<td>11,78</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>34:08,3</td>
<td>1</td>
<td>2,5</td>
<td>8,6</td>
<td>1,3</td>
<td>1,2</td>
</tr>
<tr>
<td>35:22,2</td>
<td>2</td>
<td>2</td>
<td>11,01</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>36:27,7</td>
<td>1</td>
<td>2,5</td>
<td>8,88</td>
<td>1,3</td>
<td>1,2</td>
</tr>
<tr>
<td>37:37,6</td>
<td>2</td>
<td>2</td>
<td>12,24</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>38:46,6</td>
<td>1</td>
<td>2,5</td>
<td>8,83</td>
<td>1,3</td>
<td>1,2</td>
</tr>
<tr>
<td>39:57,1</td>
<td>2</td>
<td>2</td>
<td>12,95</td>
<td>1,2</td>
<td>1,3</td>
</tr>
</tbody>
</table>

6.4 Strategy 3: Panic approach

The third strategy is the panicking approach. This strategy is an extension of the greedy strategy. The strategy will still find the local top in the sense
of configuration, but it also has the ability to horizontally scale the site. This means that if the performance requirement is not met with the lowest quality settings, it will start to create new web servers to accommodate the traffic.

This strategy is also able to quickly accommodate a spike in traffic. If the current measured response time is twice the size or larger. The Manager will then instantly adjust the quality settings to the lowest possible settings and start creating web servers. One could say that the manager is caught by surprise and scrambles to get an overview of the situation. Thus the name panic. When the web servers are created the Manager will then adjust the quality upwards until the performance threshold is broken.

6.4.1 Flat: 40 connections

This experiment consist of 5 separate trials. Each of these trials where run with identical specifications, meaning that the simulated traffic and performance thresholds are perfectly similar. There is however a new specification in these and that is the VM spawn delay. This is implemented to simulate a busy cloud, where creating a new VM may take some time. The crawler does also run with 2 connections for 60 seconds before increasing to 40 seconds. This is to force the Manager into panic mode.

Experiment specifications

- Number of experiments: 5
- Seconds between each configuration: 20 seconds
- VM spawn time delay: 10 seconds
- Crawler schedule:
  1. 2 connections : 60 seconds
  2. 40 connections : 900 seconds (15 minutes)

- Performance thresholds:
  - 1: Response time < 10
  - 2: Response time > 10 AND Response time < 14
  - 3: Response time > 14

6.4.2 Experiment results

Comparing the results from the Panic experiment again shows that the trials are very similar to each other. The local top present at the start of the graph is the 60 seconds of 2 connections. The small time window of 2 connections is there to ensure that the Manager invokes the panic strategy.
When panic has been invoked, the Manager then quickly creates more servers. This severely reduces the maximum peak of the response time. The highest response time of the trials hovers between 30 and 35 seconds here, where the other experiments had as high as 75.

Every trial in this experiment are very similar, no trial is deviating from the norm in the same fashion as observed in the other experiments. The only difference between this experiment and the other is that these trials where not executed right after each other, but with a longer period of down time between each trial. Though this should not have any effect since the whole framework are created from scratch for every trial.

Figure 6.7: Spaghetti plot comparing each trial from the panic experiments with 40 connections

6.4.3 Second trial

As seen in this chart, the traffic starts at an acceptable level, with two connections the response time is below 5 seconds. When the crawler increases the connections to 40, the response time spikes accordingly. Immediately the Manager panics and adjusts everything down to its lowest settings while simultaneously starting the creation another web server. This results in a much lower peak in response time which is around 30 seconds, contrary to the other strategies which had a peak at around 70 seconds. Then we have another smaller peak at around 25:00,0 which is probably an attempt to scale up the image size, which pushes the response time over an acceptable level causing it to be reduced again.
Figure 6.8: Figure comparing all of the results from the flat exhaustive experiments with 40 connections

This table shows the Managers decisions up to 28:57. There are some interesting decisions in this table. The first is where panic is invoked. The first row shows the first measurement gathered by the Manager and the response time is at that time at 5.84 seconds. The next measurement have a response time of 31.62 seconds which is well over the double required to invoke panic. The configuration are then immediately changed from 1,1 to 3,3. The Manager does also start to horizontally scale the web servers at this time.

It takes roughly 60 seconds before the Manager registers its new servers, though the servers themselves are in production a short while before they are registered. This is because the thread creating the server are running checks to ensure that the server creation was successful.

A peculiarity is the jump at 27:22 from 3,1 to 3,3. This looks like another panic by the Manager, even though the response time is not twice the size of the last registered response time. It is uncertain what has caused this, since this is the only occurrence in this data set.

66
Table 6.3: Table showing the Manager decisions during the second trial with the panic strategy

<table>
<thead>
<tr>
<th>Time</th>
<th>Performance</th>
<th>Quality</th>
<th>Response time</th>
<th>Nr of web</th>
<th>Nr of img</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>22:32.8</td>
<td>1</td>
<td>1.5</td>
<td>5.84</td>
<td>web=1</td>
<td>image=1</td>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>23:06.9</td>
<td>3</td>
<td>1.5</td>
<td>31.62</td>
<td>web=1</td>
<td>image=1</td>
<td>1,1</td>
<td>3,3</td>
</tr>
<tr>
<td>23:39.5</td>
<td>3</td>
<td>4.5</td>
<td>14.95</td>
<td>web=1</td>
<td>image=1</td>
<td>3,3</td>
<td>3,3</td>
</tr>
<tr>
<td>24:11.8</td>
<td>1</td>
<td>4.5</td>
<td>9.87</td>
<td>web=1</td>
<td>image=1</td>
<td>3,3</td>
<td>3,2</td>
</tr>
<tr>
<td>24:45.4</td>
<td>1</td>
<td>4</td>
<td>8.86</td>
<td>web=2</td>
<td>image=1</td>
<td>3,2</td>
<td>3,1</td>
</tr>
<tr>
<td>25:14.2</td>
<td>2</td>
<td>3.5</td>
<td>11.55</td>
<td>web=2</td>
<td>image=2</td>
<td>3,1</td>
<td>3,3</td>
</tr>
<tr>
<td>25:46.4</td>
<td>3</td>
<td>4.5</td>
<td>14.34</td>
<td>web=2</td>
<td>image=2</td>
<td>3,3</td>
<td>3,3</td>
</tr>
<tr>
<td>26:18.4</td>
<td>1</td>
<td>4.5</td>
<td>8.63</td>
<td>web=2</td>
<td>image=2</td>
<td>3,3</td>
<td>3,2</td>
</tr>
<tr>
<td>26:50.5</td>
<td>1</td>
<td>4</td>
<td>9.06</td>
<td>web=2</td>
<td>image=2</td>
<td>3,2</td>
<td>3,1</td>
</tr>
<tr>
<td>27:22.7</td>
<td>2</td>
<td>3.5</td>
<td>10.38</td>
<td>web=2</td>
<td>image=2</td>
<td>3,1</td>
<td>3,3</td>
</tr>
<tr>
<td>27:54.2</td>
<td>2</td>
<td>4.5</td>
<td>10.47</td>
<td>web=2</td>
<td>image=2</td>
<td>3,3</td>
<td>3,3</td>
</tr>
<tr>
<td>28:25.6</td>
<td>1</td>
<td>4.5</td>
<td>9.24</td>
<td>web=2</td>
<td>image=2</td>
<td>3,3</td>
<td>3,2</td>
</tr>
<tr>
<td>28:57.1</td>
<td>1</td>
<td>4</td>
<td>9.71</td>
<td>web=2</td>
<td>image=2</td>
<td>3,2</td>
<td>3,1</td>
</tr>
</tbody>
</table>
Chapter 7

Discussion

This chapter will discuss the aspects of this project, trying to reflect upon how the problem was approached and how the final results fits together with the problem statement as well as prior research within the same field. This chapter will also attempt to predict what impact this will have on the IT industry and how it is managed today.

7.1 Background and approach

This project was based upon an interesting tactic utilized by administrators during the 9/11 tragedy in 2001. News sites experienced such a high load, that the web servers became overloaded. Administrators tackled this issue by reducing the quality of the web page that received the most traffic. This did prevent the pages from going down.

The problem statement revolves around how this method of scaling can be used today to keep quality of service within an acceptable margin. Even though there where little to no research conducted in regards to content scaling in particular. Lots of research has been conducted in the field of scaling, this information became very useful throughout the course of this project. Being able to use common terms within scaling and virtualization also made it easier to write the report at the end of this project.

7.2 Design and implementation

When designing the model of the framework in this project, there where many factors to take into consideration. Many of the design choices where based upon assumptions alone due to the lack of documentation and prior research on the issue of content scaling. Though much research has been performed on scaling in itself as a concept making it possible to use the same concepts like service components and quality of service as a starting point. This section will describe the most significant design decisions and how these choices affected the implementation phase.
7.2.1 Manager

One of the more significant design choices was to ensure that it should be simple to implement different strategies for testing, without having to edit major parts of the code. This is one of the reasons why the Manager program was split into two parts. The control panel and the Manager itself. By having the control panel operating everything that are specific to the site, the Manager can operate on a conceptual level. Interpreting quality and performance as arbitrary numbers instead of detailed information like image sizes, makes it much easier to create strategies. This turned out to be a great decision during the implementation phase. Creating the different strategies where achieved by simply making a copy of the bare Manager and start to implement the thinktank.

Splitting the Manager and control panel could have been done in two ways, the first was by creating the frame work in an object oriented fashion. This was the solution that was ultimately chosen. The advantage of an object oriented approach is that the program works within a single process. This means that global variables and general synchronising between the the Manager and the control panel is easier. The major draw back of doing object oriented programming is that the code may seem more complex and less intuitive than a script. This was however not an issue during this project because of my experience with object oriented programming.

The other approach that could be used to separate the two, could be by using different files. The control panel would be located in its own file and the Manager in another, creating to completely separate processes. One could argue that this is a more true form of separation, though this could also have been a time consuming endeavor. The reason is that even though Manager and the control panel are supposed to be separated, they still need to communicate. By having separate processes means that they have to communicate through commands, loop back interfaces or files. All three are time consuming to implement and adds another layer of mechanisms to be tested.

Another mechanism presented in the design chapter was the locking system. This locking system served the purpose of making sure that the Manager did not start to configure servers that was already being configured. This turned out to be completely redundant since the time between each implementation was far longer than the time it took to reconfigure a server. Though this locking mechanism was not for naught. An identical mechanism was created for the horizontal scaling mechanism in the panic strategy. This mechanism made sure that the Manager could not create several servers at once. It had to create one server and then wait for that server to be set in production. Meaning that Manager had to wait an amount of time between server creations.

It was decided early in the design process that threading had to be uti-
lized to make sure that the Manager part of the framework would not stop executing while the control panel was operating the site. It was thought that strategies would run completely independent from what is currently happening in the network. While this is theoretically true, it did matter little in practice. The reason behind this is that every time Manager decides to operate the control panel, it still have to wait to see the effects of the changes. Though one advantage that still remains is that the threads will finish executing even if the Main thread dies. Making sure that no VMs are half configured even if the process was suddenly aborted.

The last design choice that was made in the design phase of the project was the independent daemons reporting the system status of the servers. Making it possible for the Manager to read the CPU load, memory usage, numbers of connections etc. This was supposed to give the Manager another set of metrics. Being able to read both the feedback from users (performance and quality) and the server specific metrics would give the manager more to work with. This in turn could make it possible for the manager to make more logical decisions.

An example of this could be that the response time starts to climb, affecting the performance score. The current prototype would simply iterate through configurations until an optimal has been found. If the Manager had access to for example memory usage, it could have identified that the image servers where running at full memory capacity. Which in turn would mean that any changes to the web servers would be futile. This would make the Manager more efficient in its scaling process. It would also mean that the framework would have more tools to offer to anyone wanting to implement their own strategy. Regrettably this mechanism had to be abandoned during the implementation phase due to time constraints. The daemons where tested and where deemed functional, but there was no time to find an acceptable method to present these metrics in an conceptual level like quality and performance.
7.3 Crawler

At the start of this project it was assumed that a suited testing tool for the web site already existed. This proved to be a mistake. The HTTP test tools that exists today are meant to test the servers capacities and response time, but this is only for a single web page. Meaning that these tools cannot measure the load time of a complete site with images. There would be no point in testing the effectiveness of content scaling when there is no method of detecting the changes.
This is why Crawler had to be created. Crawler is able to measure the time from the initial request until the final image has been downloaded. Crawler is able to run several connections at once and it is able to log the response time. Though it do have two major problems.

The first problem is that it is only partly able to simulate a complete page request. Meaning that crawler does not actually open a browser and loads the page. It instead opens the index site, locates all the image links and proceeds to download them one by one. A typical session would download all of the images more or less simultaneously. This may affect the average response time, making the response time longer than a "normal" web request. Though this does not render the data less valuable since the effects of the content scaling is still clearly visible in the charts.

The other major issue with the Crawler is how the response time starts to increase independently of the sites configuration. The reason behind this is probably a memory leak issue, where the threads created does not die or somehow keeps using more memory space as they continue to run. This is the reason why no long term experiments could be run. The Crawler would simply crash, though it does seem to be stable at the first 30 minutes of run time. Regrettably there was not enough time to locate and correct the defect that caused this memory leak. The time schedule was already skewed because a testing tool had to be implemented, leaving less time for experiments.

7.4 Measurements and analysis

The original plan in this project was to conduct a wide range of experiments with a varying amount of connections. This would have shown how the framework would handle the stress of a fluctuating amount of connections. This was not feasible due to the development time of the Crawler and due to its inability to run for longer periods of time.

The problems with the Crawler also made it hard to run a experiment showing how the panic strategy scaled down the number of servers in the framework. This strategy requires 5 "perfect" readings where both the quality and performance is ideal. Though this is challenging when the Crawler increases the response time. This resulted in tests where the framework where not scaled down because it never reached 5 consecutive ideal readings.

7.5 Effects of content scaling

When looking at the results as a whole, it is evident that content scaling do have an effect on the response time. There is a clear trend that shows that changing the amount and the quality of the content is a deciding factor in
the site's ability to serve multiple clients. Though the experiments done in this thesis is not enough to determine exactly how much effect it has. Partly because of the Crawler shortcomings, but also because there have not been done experiments that compare content scaling to other more traditional forms of scaling.

Though it is safe to conclude that content scaling have an effect. Especially image sizes affected the overall response time greatly. This was contrary to the initial beliefs where it was deemed that the numbers of images would have a greater effect than the image sizes themselves. But the format seemed to not affect the response time in any major way. This could be caused by the vast difference in the image sizes. The difference between 7,22 MB and 2,27 MB might be so much that it drowns out the more subtle changes by the format. Looking back on this, it would have made more sense to create the configuration tree with the format as the leafs and not the other way around. This would have changed the response time line in the chart from the zigzag shape to a more gradient one, which would have been easier to interpret and compare with each other.

7.5.1 Advantages of content scaling

A clear strength of content scaling is the speed that the changes can be implemented. This is something that is worth accentuating. Running a heavy site with a wide range of different complex service components can make traditional scaling a time consuming process. The time it takes to boot up a VM and also the time it takes to configure the machine correctly by using a central configuration manager of a remote repository will add up. This is where content scaling can be used as an immediate measure. Content scaling has been shown to take effect immediately and will alleviate the load on the site. The content of the site could be scaled down while the new machines are being created and configured, when these machines are done, the content can be scaled back up again. This gives the site a redundancy that can function as a fallback mechanic in the cases where traditional scaling is not fast enough.

Another advantage of content scaling is the fact that is will not cost any money to implement it. If an administrator have a budget of 10 virtual machines, the administrator can use traditional scaling techniques up to that budget limit and then start to scale the content. This gives the administrator another tool to use when the budget for active VMs are exhausted.

7.5.2 The weaknesses of content scaling

Content scaling has in itself no major technical weaknesses. Content scaling is in theory a fast and reliable way of reducing the load of a server without losing connections. Though it important to take the users impression of the site into perspective. A person who uses the site might think that the site is
unreliable or of poor quality if the image quality and page design is rapidly changing without any reason. This can again cause poor conversion rates or a bad reputation between users. This could possibly be mitigated by informing the user with a text box or similar about the cause for the sudden change in page layout, but this might still be viewed as a nuisance.

Another weakness of content scaling is that the administrators of the architecture needs intimate knowledge about their system to properly implement content scaling. The administrators have to know what content should be scaled and to what degree. The administrators also needs to understand service component and how each of these reacts to changes in the content. This might seem as a given, but keep in mind that IT consultants is often responsible for implementing parts of an architecture. The software used by the consultant might be proprietary and the implementation process might be poorly documented. It is important to keep these scenarios in mind if one should want to expand upon this project.

7.6 Affected fields

A question worth asking is how the IT industry as a whole will be affected if content scaling became a wide spread scaling technique. Starting with the obvious, IT administrators do get a new tool to use when traditional scaling strategies are too expensive or not possible for different reasons. This could make it easier to IT administrators to keep the up time to an acceptable level. So it is reasonable to predict that content scaling will help IT administrators to do their work, but how will this affect the other fields in IT?

7.6.1 Web design

Web design is something that comes to mind. As mentioned as one of the weaknesses of content scaling, using content scaling means that users will receive a poorer web site then they are used to. This is in direct opposite of what web designers work towards. A web designer wishes to create a site that is both user friendly and has a design that appeals to its users. Sudden changes in the format or the image quality might alter the site in a way that will cause unwanted side effects.

So if content scaling where to be implemented in an actual enterprise, this would have to be done in cooperation with the web designers. Not to just only agree upon what quality is acceptable from a users standpoint, but also to make sure that the sites would still be user friendly. This could result in three completely different page layouts that will be swapped depending on the pressure the site receives. And lastly all of the different page layouts needs to be made sure to follow standards and regulations in the country of operation. For instance in Norway there is a regulation that states that every web site will accommodate users with poor eye sight.
or other disabilities[19]. This means that not only has the font and design be readable and have an intuitive layout, but all text to speech programs needs to be able to interpret the site, no matter the configuration.

7.6.2 Security

Content scaling does not create any potential security risks apart from having another server with administrative privileges in the network. Though this is absolutely minimal considering that the Manager would most likely be placed in the DMZ and not together with the more crucial servers. One interesting aspect of the content scaling system is that it could potentially be used to detect a DDOS attack. It is not unfeasible for the content scaling framework to start to invoke DDOS mitigating measures when the amount of connections have met a certain threshold. An example of such a measure could be that if the server quota is used up and the content is at the lowest configuration the Manager will initiate TCP cookies on the firewall.

This could offer companies a automated solution to mitigate the possible damage of a DDOS attack. Though this has in no way been tested in this project and needs to be researched further to confirm if this is even possible.

7.7 Future work

This framework aims to present content scaling as a low cost alternative for IT administrators in addition to traditional scaling techniques. This project has managed to propose a solution, though there are many aspect of this proposal that needs further research.

The framework created in this project has proven that content scaling have an effect, but due to faults with the Crawler it is uncertain how much. More research is needed to develop a crawler that more accurate in emulating an actual HTTP GET of the page and all the images in the site. It is also important to create a tool which can run for a longer amount of time so that longer experiments can be conducted.

The strategies created in this project is proficient in showing that content scaling have an effect, but they do not take any intelligent decisions. It would be interesting to see this framework function with an artificial intelligence that can read the information from the control panel and take decision based upon that information. Especially by using the daemons mentioned earlier in the chapter in addition to the response time feedback.
Chapter 8

Conclusion

The goal of this project was to investigate how content scaling can be applied to uphold quality of service in a virtual environment in a way that allows to use different strategies to handle a high amount of connections.

The problem statement was addressed by developing a framework that was able to create, configure and measure service components and react according to the amount of stress these components where receiving. A testing tool also had to be created to be able to measure the effects of the content scaling performed by the framework.

The results and experiences gained through this projects show that content scaling do have an effect on the performance of the service delivered by the site. Each strategy makes the site behave differently when the load hit a critical threshold, but each strategy is effective in mitigating the response time.

The framework was able to show that content scaling was able to severely reduce the response time of a experiencing an overload of connections. Content scaling alone was able to reduce the response time from over 70 seconds to below 10 seconds in the time span of about 10 minutes. The strategy which used horizontal scaling in addition to the content scaling managed to reduce the initial response time from 70 seconds to 30 seconds. This shows that content scaling works even better in conjunction with horizontal scaling.

One problem in this project was the lack of a proper testing tool. This was solved by making a new one, but this tool is suboptimal. This is the reason that it is only safe to conclude that content scaling have an effect, but it is uncertain how much because of the contaminating factor of the tool. This is challenge that could be addressed in future research.

Content scaling do however show promising results, being able to severely reduce response time. More research is however needed to gain a full understanding of the potential gains to justify eventual new costs of a content
scaling system.
Bibliography


Appendices
This is the manager program. Note that this code has the function "thinktank" empty. This function is where the different strategies are found. The code of each of these strategies can be found in their respective appendixes.

```python
import re
import logging
import subprocess
import threading
import datetime
import itertools
from novaclient import client
from time import sleep
from collections import defaultdict

maxWebservers = 6
nova = client.Client(2, 's237682', 'number shake piece', 's237682_project', 'https://cloud.cs.hioa.no:5000/v2.0/

logging.basicConfig(level=logging.DEBUG,
    format='%(asctime)s [%(levelname)s] (%(threadName)s-10s) %(message)s',
)

def runbash(command):
    bashcommand = subprocess.call(command, shell=True)

def getbash(command):
    bashcommand = subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
    (bashOutput, err) = bashcommand.communicate()
    return bashOutput

class Manager:
    preferableUsabilityScore = 0
    preferablePerformanceScore = 0
    usabilityScore = 0
    performanceScore = 0
```
pulse = 0
maxWebservers = 0
totalServers = 0
ControlPanel = ''
triedCombinations = []
currentMetrics = []
loadbalancerOperations = []

def __init__(self, preferablePerformanceScore, maxWebservers):
    self.preferablePerformanceScore = prefPerformanceScore
    self.preferablePerformanceScore = prefPerformanceScore
    self.loadbalancerOperations = [ 'A', 'B', 'C' ]
    self.usabilityScore = 0
    self.performanceScore = 0
    self.pulse = 0
    self.ControlPanel = ControlPanel()
    triedCombinations = []

def __del__(self):
    print "exiting"
    # logging.debug('Exiting manager')

def thinkTank(self):
    # STRATEGY IS TO BE CODED HERE!!!!

def validateSitePerformance(self):
    currentMetrics = self.ControlPanel.aggregateMetrics()
    # logging.debug(currentMetrics)
    return getbash('ssh ubuntu@10.1.13.149 < cat currentstats.log')

class ControlPanel:
totalServers = 0
currentImageservers = 0
currentWebservers = 0

servers = defaultdict(list)
serverStatus = defaultdict(list)
listOfOperations = {}
listOfCreations = {}

aggregatedMetrics = {}
currentGrades = {}

isOperationReady = {}
machineImages = {}

def __init__(self):
    self.totalServers = 0
    self.currentImageservers = 0
    self.currentWebservers = 0
    # servers = defaultdict(list)
    servers = { 'web':[], 'image':[], 'loadbalancer':[] }
    aggregatedMetrics = { 'web':0, 'image':0, 'loadbalancer':0 }
    self.listOfOperations = { 'web': 'Ready', 'image': 'Ready', "→
----
{'loadbalancer': 'Ready'}

self.listOfCreations = [{'web': 'Ready', 'image': 'Ready', 'loadbalancer': 'Ready'}]

self.createLoadbalancer()
self.addWebservers('web')
self.addWebservers('image')

def createWebserver(self, serverType):
    if (self.totalServers < maxWebservers):
        self.listOfCreations[serverType] = 'Executing'
sleep(10)
        hostname = 'webserver-~' + serverType + ' ' + str(len(serverType))
        self.servers[serverType].append(Webserver(serverType, hostname))
        if serverType == 'web':
            self.webservers += 1
        else:
            self.imagetervers += 1
        self.listOfCreations[serverType] = 'Ready'
        self.totalServers += 1
    else:
        print 'Too many servers'

def createLoadbalancer(self):
    hostname = 'loadbalancer-~' + str(len(self.servers['loadbalancer']))
    self.servers['loadbalancer'].append(Loadbalancer(hostname))

def aggregateMetrics(self):
#logging.debug(self.serverStatus)
    for type in self.aggregatedMetrics:
        self.aggregatedMetrics[type] = 0
    for type in self.serverStatus:
        self.currentGrades[type] = self.serverStatus[type][0][1]
        for serverNumber in self.serverStatus[type]:
            logging.debug(serverNumber)
            self.aggregatedMetrics[type] = int(serverNumber[0])
            self.aggregatedMetrics[type] /= len(self.servers[serverType])
    return self.aggregatedMetrics
#logMetrics(self.aggregatedMetrics)

def logMetrics(self, dataBlob):
    print dataBlob + 'Written to file!\n'

def scaleWebservers(self, grade, serverType):
    self.listOfOperations[serverType] = 'Executing'
    for server in self.servers[serverType]:
        server.applyChanges(grade)
    self.listOfOperations[serverType] = 'Ready'

def scaleLoadbalancer(self, grade):
    self.listOfOperations['loadbalancer'] = 'Executing'
    for loadbalancer in self.servers['loadbalancer']:
        loadbalancer.applyChanges(grade)
def addWebservers(self, typeOfServer):
    hostname = 'webserver-' + str(typeOfServer) + '-' + str(len(self.servers[typeOfServer]))

    if (typeOfServer == 'web'):
        self.listOfOperations['web'] = 'Executing'
        self.servers['web'].append(Webserver(typeOfServer, hostname))
        self.listOfOperations['web'] = 'Ready'
        self.currentImageservers += 1
        self.totalServers += 1
    elif (typeOfServer == 'image'):
        self.listOfOperations['image'] = 'Executing'
        self.servers['image'].append(Webserver(typeOfServer, hostname))
        self.listOfOperations['image'] = 'Ready'
        self.totalServers += 1
        self.currentImageservers += 1

class Webserver(ControlPanel):
    ipAddress = '
    serverType = '
    hostname = '
    currentGrade = 0
    loginSsh = '

    def __init__(self, serverType, hostname):
        self.hostname = hostname
        self.serverType = serverType
        self.currentGrade = 1
        ControlPanel.serverStatus[serverType].append((0, self.currentGrade, self.hostname))
        logging.debug(ControlPanel.serverStatus[serverType])
        currentScale[2] = 1
        image = nova.images.find(name='server')
        flavor = nova.flavors.find(name='m1.small')
        net = nova.networks.find(label='nsa_master_net')
        nics = [{'net-id': 'babe241e-682c-467d-8a3a-31d3580e727c', 'v4-fixed-ip': ''}]
        instance = nova.servers.create(name=self.hostname,
            image=image,
            flavor=flavor, key_name='framework_key',
            nics=nics)

        while(1 == 1):
            novaList = "nova --os-username s237682 --os-password 'number shake piece' --os-project-name s237682_project --os-auth-url 'https://cloud.cs.hioa.no:5000/v2.0/' list"
            result = getbash(novaList + " | grep " + hostname + " | awk '{ print $12}'").split(=')
            if re.match('\\d+\d+|\d+\d+', result[-1]):
                self.ipAddress = result[-1].strip()
                logging.debug('The server is waiting at IP {0}. format(self.ipAddress))
                break
sleep(1)

#sleep(10)
while ('0' not in getbash('nc ' + self.ipAddress + ' 22 < /dev/null; echo $?').split('\n')[1]):
    print "Machine is still booting..."
    sleep(5)
loginSsh = 'ssh -o StrictHostKeyChecking=no ubuntu@loadbalancer'
changeConfigOfLoadbalancer = getbash(loginSsh + ' sudo ' +
    'python changeConfig.py ' + self.serverType + ' ' + self.hostname[-1] + ' ' + self.ipAddress + ' :80 ')
restartLoadbalancer = getbash(loginSsh + ' sudo service haproxy restart ')
reporterDeamon = threading.Thread(target=self.reportStatus)
reporterDeamon.setDaemon(True)
reporterDeamon.start()
def __del__(self):
    listOfServers = nova.servers.list()
    serverExist = False
    for s in listOfServers:
        if s.name == self.hostname:
            serverExist = True
            break
    if serverExist == False:
        logging.debug('Server does not exist ')
    else:
        logging.debug('deleting server ')
    loginSsh = 'ssh -o StrictHostKeyChecking=no ubuntu@loadbalancer'
    changeConfigOfLoadbalancer = getbash(loginSsh + ' sudo ' +
        'python deleteConf.py ' + self.serverType + ' ' + self.hostname[-1] + ' ' +
        self.ipAddress + ' :80 ')
nova.servers.delete(s)
#removeFromLoadbalancer()
logging.debug('server deleted ')

def applyChanges(self, grade):
    loginSsh = 'ssh -o StrictHostKeyChecking=no ubuntu@' + self.ipAddress
    if (self.serverType == 'web '):
        if (grade == 1):
            format = 50
        elif (grade == 2):
            format = 25
        else:
            format = 10
        #echo 2 | ssh ubuntu@10.1.16.46 " cat > /var/www/html/" + format .txt"
        print getbash('echo ' + str(format) + ' | ' + loginSsh + ' " cat > /var/www/html/format .txt " ')
    elif (self.serverType == 'image '):
        getbash('touch sshImage .txt ')
        print getbash('echo ' + str(grade) + ' | ' + loginSsh + ' " cat > /var/www/html/" ')

print getbash(loginSsh + ' \
rm /var/www/html/image.jpg')
print getbash(loginSsh + ' \
sudo ln -s /var/www/html/' + str(grade) + '.jpg /var/www/html/image.jpg')

else:
    logging.debug('serverType not found')
self.currentGrade = grade

def reportStatus(self):
    # while ('0' not in getbash('nc ' + self.ipAddress + ' 22< /dev/null; echo $?').split('
')[1]):
        # logging.debug("Machine is still booting...")
    # sleep(5)
    while 1 == 1:
        result = self.getServerMetrics()
        i = 0
        for serverTuple in ControlPanel.serverStatus[self.serverType]:
            if serverTuple[2] == self.hostname:
                ControlPanel.serverStatus[self.serverType][i] = (result, self.currentGrade, self.hostname)
            i += 1
        sleep(5)

def getServerMetrics(self):
    if (int(self.currentGrade) == 1):
        return 0
    elif (int(self.currentGrade) == 2):
        return 10
    elif (int(self.currentGrade) == 3):
        return 20
    else:
        return 50

    #return getbash(self.loginSsh + ' \
netstat -an | grep ' + str(self.ipAddress) + ':80 | wc -l')).strip()

class Loadbalancer(ControlPanel):
    ipAddress = ''
    hostname = ''
    VCLFileName = ''
    possibleOperations = []
    configuration = ''
    def __init__(self, hostname):
        self.configuration = 'A'
        self.hostname = hostname
        ControlPanel.serverStatus['loadbalancer'].append((0, self.configuration, self.hostname))
        logging.debug(ControlPanel.serverStatus['loadbalancer'])
        image = nova.images.find(name='loadbalancer')
        flavor = nova.flavors.find(name='m1.small')
        net = nova.networks.find(label='nsa_master_net')
        nics = [ [ 'net-id': 'babe241e-682c-467d-8a3a-31d3580e727c', 'v4-fixed-ip': '' ]]
        instance = nova.servers.create(name=self.hostname, image=image, flavor=flavor, key_name='framework_key',
sleep(5)

while (1 == 1):
    novaList = " nova --os-username s237682 --os-password "
    'number shake piece' --os-project-name 's237682_project' --os-auth-url 'https://cloud.cs.
    hioa.no:5000/v2.0/' list"
    result = getbash(novaList + " | grep + hostname + "
    | awk '{ print $12}'").split('="')
    if re.match(\d+\.\d+\.\d+\.\d+", result[-1]):
        self.ipAddress = result[-1].strip()
        logging.debug("The server is waiting at IP " +
            self.configuration.format(self.ipAddress, self."
        break
    sleep(1)
    while ("0" not in getbash('nc + self.ipAddress + ' 22 < /dev/null; echo $?').split(\n')[-1]):
        print "Machine is still booting..."
        sleep(5)
    runbash("echo " + self.ipAddress + " loadbalancer > /etc/hosts")
    #sleep(10)
    loginSsh = 'ssh -o StrictHostKeyChecking=no ubuntu@' +
        str(self.ipAddress)
    restartOK = getbash(loginSsh + ' \sudo service haproxy :
        restart')
    logging.debug(restartOK)
    reporterDeamon = threading.Thread(target=self."
        reportStatus)
    reporterDeamon.setDaemon(True)
    reporterDeamon.start()

def __del__(self):
    listOfServers = nova.servers.list()
    serverExist = False
    for s in listOfServers:
        if s.name == self.hostname:
            serverExist = True
            break
    if serverExist == False:
        logging.debug( 'Server does not exist'
    else:
        logging.debug( 'deleting server'
    nova.servers.delete(s)
    #removeFromLoadbalancer()
    logging.debug( 'server deleted '

def reportStatus(self):
    while (1 == 1):
        result = self.getServerMetrics()
        #print 'run bash command to fetch status\n'
        #print 'log status to file\n'
        i = 0
        for serverTuple in ControlPanel.serverStatus['loadbalancer']:
            if serverTuple[2] == self.hostname:
                ControlPanel.serverStatus['loadbalancer'][i] = (result, self.configuration, self."

89
hostname

```
324  i += 1
325  sleep(1)
326
327  def getServerMetrics(self):
328      if self.configuration == 'C':
329          return 0
330      elif self.configuration == 'B':
331          return 10
332      elif self.configuration == 'A':
333          return 20
334      # return getbash(self.loginSsh + ' \'netstat -an | grep '↩
335      #   + str(self.ipAddress) + ':80 | wc -l\').strip()
336
337  def applyChanges(self, command):
338      self.configuration = command
339      command = '<bash command>(grade)
340      print 'execute bash command to configure the loadbalancer'
341
342  runbash( 'ssh-keygen -f "/root/.ssh/known_hosts" -R loadbalancer'↩
343           )
344  monkey = Manager(0, 0, 5)
345  subprocess.Popen([ 'python', 'crawler-schedule.py' ])
346  sleep(90)
347  monkey.thinkTank()
```

90
Appendix B

Exhaustive strategy

This is the code of the exhaustive strategy. The code for the rest of the manager can be found in appendix A.

```python
def thinkTank(self):
    bestTime = 99.99
    timeLimits = [10.0, 14.0, 18.0]
    configurationResults = []
    # 1st digit = web 2nd digit = image
    for combination in itertools.product('123', repeat = 2):
        # GRADE 123,132,213,231,312,321
        for operation in self.loadbalancerOperations:
            currentConfiguration = combination[0] + combination[1]
            self.triedCombinations.append(currentConfiguration)
        logging.debug('inner loop')
        webserverOperatorWeb = threading.Thread(target=
            ControlPanel.scaleWebservers, args=(self.
            ControlPanel, combination[0], 'web',))
        webserverOperatorImage = threading.Thread(target=
            ControlPanel.scaleWebservers, args=(self.
            ControlPanel, combination[1], 'image',))
        # loadbalancerOperator = threading.Thread(target=
        # ControlPanel.scaleLoadbalancer, args=(self.
        # ControlPanel, operation,))
        webserverOperatorWeb.start()
        webserverOperatorImage.start()
        # loadbalancerOperator.start()
        liste = [True]
        while True in liste:
            # sleep(120)
            i = 0
            liste = [False, False, False]
            for item in self.ControlPanel.listOfOperations:
                if self.ControlPanel.listOfOperations[item] == 'Executing':
                    liste[i] = True
            logging.debug(item + '−servers are still being configured. Keep waiting')
            sleep(5)
sleep(60)
```
httpTimes = self.validateSitePerformance().strip().split(',
print httpTimes
with open('Metrics.log', 'a') as crawlerMetrics:
i = 1
performanceValue = 3
for limit in timeLimits:
    if float(httpTimes[2]) < float(limit):
        performanceValue = i
        break;
i += 1
qualityValue = ((float(combination[0]) * 2) + float(combination[1])) / 2
configurationResults.append((performanceValue, qualityValue,
currentConfiguration[0], currentConfiguration[1],
httpTimes[1], httpTimes[2]))
    # bestTime = httpTimes[2]
    # crawlerMetrics.write('*
    # bestConfiguration = currentConfiguration
    crawlerMetrics.write(str(datetime.datetime.utcnow()) + ',
    ' + str(configurationResults[-1]) + '\n')
    #logging.debug(str(currentConfiguration))
bestConfiguration = configurationResults[0]
for item in configurationResults:
    if int(bestConfiguration[0]) > int(item[0]):
        bestConfiguration = item
    if int(bestConfiguration[0]) == int(item[0]):
        if float(bestConfiguration[1]) > float(item[1]):
            bestConfiguration = item
logging.debug('The best web configuration is: ' + str(bestConfiguration[2]) + '；The best image configuration is:
    ' + str(bestConfiguration[3]))
with open('Metrics.log', 'a') as crawlerMetrics:
crawlerMetrics.write('The best web configuration is: ' + str(bestConfiguration[2]) + '；The best image
    configuration is: ' + str(bestConfiguration[3]))
webserverOperatorWeb = threading.Thread(target=ControlPanel.scaleWebservers, args=(self, ControlPanel, bestConfiguration[2], 'web',))
webserverOperatorImage = threading.Thread(target=ControlPanel.scaleWebservers, args=(self, ControlPanel, bestConfiguration[3], 'image',))
Appendix C

Greedy strategy

This is the code of the greedy strategy. The code for the rest of the manager can be found in appendix A.

```python
def thinkTank(self):
    timeLimits = [10.0, 14.0, 18.0]
    configurationResults = []
    possibleConfigurations = []
    # 1st digit = web 2nd digit = image
    for combination in itertools.product('123', repeat=2):
        possibleConfigurations.append(combination)
    print(possibleConfigurations)
    print(possibleConfigurations[0])
    currentConfig = possibleConfigurations[0]
    configurationPointer = 0
    while True:
        currentConfiguration = possibleConfigurations[configurationPointer]
        webserverOperatorWeb = threading.Thread(target=self.ControlPanel.scaleWebservers,
                                                  args=(self.ControlPanel, currentConfiguration[0], 'web',))
        webserverOperatorWeb.start()
        webserverOperatorImage = threading.Thread(target=self.ControlPanel.scaleWebservers,
                                                   args=(self.ControlPanel, currentConfiguration[1], 'image',))
        webserverOperatorImage.start()
        i = 0
        liste = [False, False, False]
        for item in self.ControlPanel.listOfOperations:
            if (self.ControlPanel.listOfOperations[item] == 'Executing'):
                liste[i] = True
                logging.debug(item + '−servers are still being configured. Keep waiting')
                sleep(5)
                sleep(60)
    httpTimes = self.validateSitePerformance().strip().split(',')
```
i = 1
performanceValue = 3
for limit in timeLimits:
    if float(httpTimes[2]) < float(limit):
        performanceValue = i
        break;
    i += 1
qualityValue = ((float(currentConfiguration[0]) * ← 2) + float(currentConfiguration[1])) / 2
currentIndex = (performanceValue, qualityValue, ← currentConfiguration[0],
                currentConfiguration[1], httpTimes[1], httpTimes[2])
    if float(currentResult[0]) > 1.0:
        if int(configurationPointer) < len(possibleConfigurations) - 1:
            configurationPointer += 1
            print "ADDING"
        else:
            logging.debug('cannot go any lower! D: ')
    elif float(currentResult[0]) == 1.0:
        if float(currentResult[1]) > 2.0:
            print "SUBTRACTING"
            print currentResult[1]
            configurationPointer -= 1
        else:
            logging.debug('State is ideal')
descisionFile = open('results.log', 'a')
descisionFile.write(str(datetime.datetime.utcnow()) + ' ←
                  ' + str(currentResult) + ' : ' + str(currentResult[2]) + ' , ' + str(currentResult[3]) + ' −−> ' + str(possibleConfigurations[int(configurationPointer)]) + '\n')
Appendix D

Panic strategy

This is the code of the panic strategy. The code for the rest of the manager can be found in appendix A.

```python
def thinkTank(self):
    bestTime = 99.99
    # 10.0, 14.0, 18.0
    timeLimits = [10.0, 14.0, 18.0]
    configurationResults = []
    possibleConfigurations = []
    # 1st digit = web 2nd digit = image
    for combination in itertools.product('123', repeat = 2):
        possibleConfigurations.append(combination)
    print possibleConfigurations
    print possibleConfigurations[0]

    currentConfig = possibleConfigurations[0]
    configurationPointer = 0
    createWebserver = threading.Thread(target=ControlPanel.createWebserver,
        args=(self.ControlPanel, 'web',))
    createImageserver = threading.Thread(target=ControlPanel.createWebserver,
        args=(self.ControlPanel, 'image',))
    serverToCreate = 'web'
    lastHttpTimes = 0.0
    numberOfIdealStates = 0

    while True:
        print ControlPanel.listOfCreations

        currentConfiguration = possibleConfigurations[configurationPointer]
        webserverOperatorWeb = threading.Thread(target=ControlPanel.scaleWebservers,
            args=(self.ControlPanel, currentConfiguration[0], 'web',))
        webserverOperatorImage = threading.Thread(target=ControlPanel.scaleWebservers,
            args=(self.ControlPanel, currentConfiguration[1], 'image',))
        webserverOperatorWeb.start()
webserverOperatorImage.start()

time = 0

for item in self.ControlPanel.listOfOperations:
    if (self.ControlPanel.listOfOperations[item] == 'Executing'):
        time[i] = True
        logging.debug('webserverOperatorImage.start()')
        sleep(5)
        sleep(20)

httpTimes = self.validateSitePerformance().strip().split(' , ')

i = 1

for limit in timeLimits:
    if float(httpTimes[2]) < float(limit):
        performanceValue = i
        break;

    i += 1
    qualityValue = ((float(currentConfiguration[0]) * 2) + float(currentConfiguration[1])) / 2

    currentResult = (performanceValue, qualityValue, currentConfiguration[0], currentConfiguration[1], httpTimes[1], httpTimes[2], 'web' + str(len(self.ControlPanel.servers[ 'web'])) , 'image' + str(len(self.ControlPanel.servers[ 'image'])))

    if int(currentResult[0]) > 1:
        logging.debug('resetting NumberOfIdealStates')
        numberOfIdealStates = 0
        print str(numberOfIdealStates) + '\n'
        if (httpTimes[2] > (float(lastHttpTimes) + 2)):
            configurationPointer = 8
            if self.ControlPanel.listOfCreations[serverToCreate] == 'Ready':
                if serverToCreate == 'web':
                    serverToCreate = 'image'
                    createWebserver.start()
                else:
                    serverToCreate = 'image'
                    createImageserver.start()
            elif configurationPointer < len(possibleConfigurations) - 1:
                configurationPointer += 1
                print 'ADDING'
            else:
                logging.debug('cannot go any lower! D: ')
                print ControlPanel.listOfCreations
                if ControlPanel.listOfCreations[serverToCreate] == 'Ready':
                    if serverToCreate == 'web':
                        else:
createWebserver.start()
serverToCreate = 'image'
else:
    serverToCreate = 'image'
createImageserver.start()

    elif int(currentResult[0]) == 1:
        if float(currentResult[1]) > 1.5:
            print "SUBTRACTING"
            print currentResult[1]
            configurationPointer -= 1
        else:
            logging.debug('State is ideal')
            numberOfIdealStates += 1
            print str(numberOfIdealStates) + '\n'

            logging.debug('numberOfIdealStates is:' + str(numberOfIdealStates))
            if int(numberOfIdealStates) >= 5:
                print str(numberOfIdealStates) + 'IN IF TO DELETE SERVER
'

if ((serverToCreate == 'image') & (len(self.ControlPanel.servers['web']) != 1)):
    logging.debug('popping web')
    self.ControlPanel.servers['web'].pop
    self.ControlPanel.servers['web'][−1]
    servertocreate = 'web'

elif ((serverToCreate == 'image') & (len(self.ControlPanel.servers['image']) != 1)):
    logging.debug('popping image')
    self.ControlPanel.servers['image'].pop
    serverToCreate = 'image'

else:
    logging.debug('Minimum requirement of servers')
    del self.ControlPanel.servers['image'][−1]
    logging.debug('resetting numberOfIdealStates')
    print str(numberOfIdealStates) + '\n'

numberOfIdealStates = 0
descisionFile = open('descisionFile.log', 'a')
descisionFile.write(str(datetime.datetime.utcnow()) + ' ' + str(currentResult[2]) + ' ' + str(currentResult[3]) + ' ' + str(possibleConfigurations[int(configurationPointer)]) + '\n')

lastHttpTimes = httpTimes[0]
Appendix E

Crawler-schedule.py

The code of the crawler-schedule.py. This python script is used to initiate the Crawler program located on another server.

```python
from time import sleep
import subprocess

def runbash(command):
    bashcommand = subprocess.call(command, shell=True)

def getbash(command):
    bashcommand = subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
    (bashOutput, err) = bashcommand.communicate()
    return bashOutput

loadbalancerIP = getbash('cat /etc/hosts | grep loadbalancer')
loadbalancerIP = loadbalancerIP.split(' ')[0].strip()
runbash('echo ' + loadbalancerIP + ' | ssh ubuntu@10.1.13.149 " cat > /home/ubuntu/loadbalancerIP.cfg"

with open('crawler-schedule.cfg', 'r') as f:
    for line in f:
        if not line.startswith('#'):
            data = line.split(':')
            amount = data[0].strip()
            duration = data[1].strip()
            print str(amount) + ':' + str(duration)
            #echo 2 | ssh ubuntu@10.1.13.149 " cat > /home/ubuntu/numberOfThreads.cfg"
            runbash('echo ' + str(amount) + ' | ssh ubuntu@10.1.13.149 " cat > /home/ubuntu/numberOfThreads.cfg"

            sleep(float(duration))
    else:
        pass
f.close()
```

Appendix F

Crawler.py

This is the code for the Crawler. Crawler was the tool developed to test the web servers controlled by the manager.

```python
#!/usr/bin/python

from bs4 import BeautifulSoup
import datetime
import urllib2
import re
import subprocess
import time
import threading
import logging
import sys

##GLOBALS##
loadbalancerFile = open('loadbalancerIP.cfg', 'r')
loadbalancerIP = loadbalancerFile.read().strip()
loadbalancerFile.close()

print loadbalancerIP

currentAmountOfThreads = 0
fileAmount = 0
maxThreads = 0
listPointer = 0

listOfTimeAverages = [0,0,0,0,0,0,0,0,0,0]
threads = []

##FUNCTIONS##
def runbash(command):
    bashcommand = subprocess.call(command, shell=True)

def getbash(command):
    bashcommand = subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
    (bashOutput, err) = bashcommand.communicate()
    return bashOutput

def crawl(weblink, depth):
    #Open page
```
html_page = urllib2.urlopen(weblink)

# Create soup object
soup = BeautifulSoup(html_page, 'html.parser')

# For each link in page
for link in soup.findAll('a'):
    strippedLink = re.search('\[(.*?)]', str(link)).group(1)

for link in soup.findAll('img'):
    strippedLink = re.search('\[(.*?)]', str(link)).group(1)
    dummy = getbash('wget -O /dev/null -o /dev/null ' + strippedLink)

##CLASS##
class worker:
    number = 0
    lastTime = -1

    def __init__(self, numb):
        self.number = numb
        w = threading.Thread(target=self.work)
        w.start()

    def work(self):
        # while True:
        while(self.number <= maxThreads):
            start = time.time()
            # logging.debug('Crawling ' + str(self.number))
            end = time.time()
            self.lastTime = end - start
            del threads[-1]

while True:
    print 'Initiating runtime with ' + str(fileAmount) + ' connections\n'
    if (maxThreads < fileAmount):
        holder = maxThreads
        maxThreads = fileAmount
        for i in range(holder, fileAmount):
            # print i
            threads.append(worker(i+1))
    elif (maxThreads > fileAmount):
        maxThreads = fileAmount
    else:
        while (maxThreads == fileAmount):
            total = 0
            lastFiveAvgs = 0
            lastTenAvgs = 0
            # print fileAmount
            with open('NumberOfThreads.cfg', 'r') as f:
                time.sleep(3)
                fileAmount = int(f.read())
            if fileAmount < 0:
                sys.exit(0)
            # print threads
            for workingman in threads:
                lastTime = workingman.lastTime
```python
if lastTime != -1:
    total += lastTime
if maxThreads != 0:
    total /= len(threads)
listOfTimeAverages[listPointer] = total
for i in range(listPointer - 4, listPointer + 1):
    if listPointer < 0:
        listPointer += 10
lastFiveAvgs += listOfTimeAverages[i]
lastFiveAvgs /= 5
for i in listOfTimeAverages:
    lastTenAvgs += i
lastTenAvgs /= 10
# print str(total) + ',' + str(lastFiveAvgs) + ',' + str(lastTenAvgs)
timefile = open('avgtime.log', 'a')
timefile.write(str(datetime.datetime.utcnow()) + ',' + str(round(lastFiveAvgs, 2)) + ',' + str(round(lastTenAvgs, 2)) + '
')
timefile.close()
statsfile = open('currentstats.log', 'w')
statsfile.write(str(round(total, 2)) + ',' + str(round(lastFiveAvgs, 2)) + ',' + str(round(lastTenAvgs, 2)) + '
')
statsfile.close()
if listPointer == 9:
    listPointer = 0
else:
    listPointer += 1
    # logging.debug('The average is:' + str(total))
else:
    loadbalancerFile = open('loadbalancerIP.cfg', 'r')
    loadbalancerIP = loadbalancerFile.read().strip()
    loadbalancerFile.close()
```
Appendix G

changeConfig.py

This is the python script for adding entries the HAProxy configuration file.

```python
import fileinput
import sys
import subprocess

def getbash(command):
    bashcommand = subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
    (bashOutput, err) = bashcommand.communicate()
    return bashOutput

processing_text = False
for line in fileinput.input('/etc/haproxy/haproxy.cfg', inplace=1):
    if line.startswith('backend ' + sys.argv[1] + ' −backend '):
        processing_text = True
        print line,

    if processing_text == True:
        if 'balance' in line:
            print 'YES',
        elif ('server' in line):
            print line,
        else:
            print '\n',
            processing_text = False

    else:
        print line,

getbash('service haproxy restart')
```
Appendix H

delConfig.py

This is the python script for removing entries in the HAProxy configuration file.

```python
import fileinput
import sys
import subprocess

def getbash(command):
    bashcommand = subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
    (bashOutput, err) = bashcommand.communicate()
    return bashOutput

processing_text = False
for line in fileinput.input('/etc/haproxy/haproxy.cfg', inplace=True):
    if line.startswith(' backend ' + sys.argv[1] + ' --backend '):
        processing_text = True
        print line,

    if processing_text == True:
        if 'balance' in line:
            print 'YES',
        elif ('server' in line):
            print line,
        else:
            print '\n',
            processing_text = False

    else:
        print line,

getbash('service haproxy restart')
```
Appendix I

Index.php for webserver

```php
<?php
//echo "format : " . $_GET["format"] . "<br>
$format = (integer) file_get_contents("format.txt");
//echo "format: $format<br>
//echo "weight: " . $_GET["weight"] . "<br>
//echo "imageIP: ". $_GET["ip"] . "<br>
```

109
mollis interdum sapien. Phasellus pharetra erat vitae eros ↔
faucibus, vitae varius lacus dictum. Vivamus scelerisque ↔
odio at ornare venenatis. Aliquam eu tincidunt nulla, varius↔
vestibulum purus. Nulla pellentesque sem quis quam cursus, ↵
ac pellentesque risus cursus. Suspendisse a maximus velit, ↔
nec consequat erat. Integer magna est, cursus in sodales et,↵
aliquam a lorem. Integer quis ipsum ligula.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. ↔
Curabitur eu arcu mi. Duis ac lorem consectetur justo ↔
bibendum maximus. Interdum et malesuada fames ac ante ipsum ↔
primis in faucibus. In ac tellus eu orci egestas pulvinar ac↔
sit amet ante. Curabitur sollicitudin mi vestibulum velit →
interdum, vitae dapibus dolor ultrices. Cum sociis natoque ↔
penatibus et magnis dis parturient montes, nascetur ↵
ridiculus mus. Donec convallis bibendum felis, a egestas ↔
metus tempus non. Nunc varius orci nisi, eget pharetra orci ↔
mollis vitae. Donec commodo ultrices purus, interdum posuere↔
est iaculis sed. Vestibulum ante ipsum primis in faucibus ↔
orci luctus et ultrices posuere cubilia Curae; Proin at ↔
libero nisi. Integer in nunc varius, tincidunt justo vel, ↔
iaculis sapien. Vestibulum fermentum malesuada lectus, ut ↔
tristique metus cursus eu. Sed ac felis iaculis, rhoncus ↔
ante vitae, pellentesque magna. Integer nec venenatis enim, ↵
eu hendrerit ex. Mauris eget purus tortor.

```php
for ( $i = 0; $i <= $_GET['weight']; $i++ ) {
    $myarray = array();
    for ( $j = 0; $j <= 100000; $j++){
        array_push($myarray, rand(0,100000));
    }    
asort($myarray);
}

for ( $i = 0; $i <= $format; $i++ ){
    echo "".$text . "$text." . "$text\n";
    echo "<br/>
";
    echo "<img src='http://$_GET[ip]/image.jpg' http://$_GET↔
[ip]/image.jpg' />
";<br/>
";
    echo "<a src='http://$_GET[ip]/image_quality.html' http↔
://$_GET[ip]/image_quality.html' />
";<br/>
";
}

?>
</BODY>
</HTML>
```